Demand Interprocedural Dataflow Analysis

Susan Horwitz, Thomas Reps, and Mooly Sagiv[†]
University of Wisconsin

Abstract

An exhaustive dataflow-analysis algorithm associates with each point in a program a set of "dataflow facts" that are guaranteed to hold whenever that point is reached during program execution. By contrast, a *demand* dataflow-analysis algorithm determines whether a single given dataflow fact holds at a single given point.

This paper presents a new demand algorithm for interprocedural dataflow analysis. The algorithm has four important properties:

- It provides precise (meet-over-all-interprocedurally-valid-paths) solutions to a large class of problems.
- It has a polynomial worst-case cost for both a single demand and a sequence of all possible demands.
- The worst-case total cost of the sequence of all possible demands is no worse than the worst-case cost of a single run of the current best exhaustive algorithm.
- Experimental results show that in many situations (*e.g.*, when only a small number of demands are made, or when most demands are answered *yes*) the demand algorithm is superior to the current best exhaustive algorithm.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.3.4 [**Programming Languages**]: Processors − *compilers, optimization*; E.1 [**Data Structures**] − *graphs*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems − *computations on discrete structures*; G.2.2 [**Discrete Mathematics**]: Graph Theory − *graph algorithms*

General Terms: Algorithms, Experimentation, Theory

Additional Key Words and Phrases: demand dataflow analysis, distributive dataflow framework, graph reachability, interprocedural dataflow analysis, interprocedurally realizable path, interprocedurally valid path, meet-over-all-valid-paths solution

---

[†]On leave from IBM Scientific Center, Haifa, Israel.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants CCR-8958530 and CCR-9100424, by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937), by the Air Force Office of Scientific Research under grant AFOSR-91-0308, and by a grant from Xerox Corporate Research.

Part of this work was done while the authors were visiting the University of Copenhagen.

A preliminary version of this paper appeared in SIGSOFT 95: Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering (Washington DC, October 10-13, 1995) [15]

Authors' address: Computer Sciences Department; Univ. of Wisconsin; 1210 West Dayton Street; Madison, WI 53706; USA.
Electronic mail: {horwitz, reps, sagiv}@cs.wisc.edu.

1. Introduction

An exhaustive dataflow-analysis algorithm associates with each point in a program a set of "dataflow facts" that are guaranteed to hold whenever that point is reached during program execution. This information can be used in a variety of software engineering tools (for example, to provide feedback to the programmer about possible errors such as the use of an uninitialized variable, or to determine whether a restructuring transformation is meaning-preserving) or can be used by an optimizing compiler (in choosing valid optimizing transformations).

It is not always necessary to compute complete dataflow information at all program points. A *demand* dataflow-analysis algorithm determines whether a given dataflow fact holds at a given point [2,10,22,20,21,11]. Demand analysis can sometimes be preferable to exhaustive analysis for the following reasons:

- ***Narrowing the focus to specific points of interest***. Software-engineering tools that use dataflow analysis often require information only at a certain set of program points. Similarly, in program optimization, most of the gains are obtained from making improvements at a program's "hot spots"—in particular, its innermost loops. However, current tools typically include a phase during which an exhaustive interprocedural dataflow-analysis algorithm is used. There is good reason to believe that the use of a demand algorithm will greatly reduce the amount of extraneous information computed.
- ***Narrowing the focus to specific dataflow facts of interest***. Even when dataflow information is desired for every program point p, the full set of dataflow facts at p may not be required. For example, it is probably only useful to determine whether the variables *used* at p might be uninitialized, rather than determining that information for all of the variables in the procedure.
- ***Reducing work in preliminary phases***. In problems that can be decomposed into separate phases, not all of the information from one phase may be required by subsequent phases. For example, the MayMod problem is to determine, for each call site, which variables may be modified during the call [4,8]. This problem can be decomposed into two phases: computing side effects disregarding aliases (the so-called DMod problem), and computing alias information [4,9,8]. Given a demand (*e.g.*, "What is the MayMod set for a given call site c?"), a demand algorithm has the potential to reduce drastically the amount of work spent in earlier phases by propagating only relevant demands (*e.g.*, "What are the alias pairs (x, y) such that x is in DMod(c)"?).
- ***Sidestepping incremental-updating problems***. A transformation performed at one point in the program can affect previously computed dataflow information at other points in the program. In many cases, the old information at such points is no longer "safe"; the dataflow information needs to be updated before it is possible to perform further transformations at such points. Incremental dataflow analysis could be used to maintain complete information at all program points; however, updating all invalidated information can be very expensive. An alternative is to demand only the dataflow information needed to validate a proposed transformation; each demand would be solved using the current program, so the

answer would be up-to-date.

- ***Demand analysis as a user-level operation***. It is desirable to have program-development tools in which the user can ask questions interactively about various aspects of a program [19,27,18,14]. Such tools are particularly useful when debugging, when trying to understand complicated code, or when trying to transform a program to execute efficiently on a parallel machine. Because it is unlikely that a programmer will ask questions about all program points, solving just the user's sequence of demands is likely to be significantly less costly than an exhaustive analysis.

Of course, determining whether a given fact holds at a given point may require determining whether other, related facts hold at other points (and those other facts may not be "facts of interest" in the sense of the second bullet-point above). It is desirable, however, for a demand dataflow-analysis algorithm to minimize the amount of such auxiliary information computed. Certainly the worst-case cost of a demand dataflow-analysis algorithm (for one demand) should be no worse than the worst-case cost of the best exhaustive algorithm. Furthermore, it is desirable that the information computed in response to one demand be reusable, so as to minimize the cost of a *sequence* of demands; we call algorithms that are able to reuse information in this way ***caching demand algorithms***. Ideally, the worst-case total cost of the sequence of demands that produces complete dataflow information should be no worse than the worst-case cost of a single run of the best possible exhaustive algorithm; we call this the ***same-worst-case-cost property***. Since no non-trivial lower bounds (other than undecidability results) are currently known for dataflow analysis, it is not possible to determine whether a demand algorithm has the same-worst-case-cost property; however, it is possible to determine whether a demand algorithm has this property with respect to a particular exhaustive algorithm.

This paper presents a new caching demand algorithm for interprocedural dataflow analysis. The new algorithm, which is an improved version of the one reported in [15], has four important properties:

- It provides precise (meet-over-all-interprocedurally-valid-paths) solutions to a large class of problems.
- It has a polynomial worst-case cost for both a single demand and a sequence of all possible demands.
- It has the same-worst-case-cost property with respect to the exhaustive algorithm given in [23], which is currently the best exhaustive algorithm for the class of dataflow-analysis problems that can be handled precisely by our demand algorithm.
- Experimental results show that in many situations (*e.g.*, when only a small number of demands are made, or when most demands are answered *yes*) the demand algorithm is superior to the current best exhaustive algorithm.

The remainder of the paper is organized as follows: Section 2 provides background material. First, the class of dataflow-analysis problems that can be handled by our algorithm is defined. Second, we show how to transform a dataflow-analysis problem in this class into a special kind of graph-reachability problem.

Section 3 presents our new algorithm, which solves demands for dataflow-analysis information by solving equivalent graph-reachability demands. Experimental results on C programs are reported in Section 4. Section 5 discusses related work.

2. Background

2.1. The IFDS Dataflow Framework

The algorithm given in Section 3 can be used to solve any interprocedural dataflow-analysis problem in which the dataflow facts form a finite set D, and the dataflow functions (which are of type $2^D \rightarrow 2^D$) distribute over the meet operator (either union or intersection). We call this class of problems the *interprocedural, finite, distributive, subset problems*, or ***IFDS problems***, for short. The IFDS problems include all *locally separable* problems—the interprocedural versions of classical "bit-vector" or "gen-kill" problems (*e.g.*, reaching definitions, available expressions, and live variables)—as well as non-locally-separable problems such as truly-live variables [13], copy-constant propagation [12, pp. 660], and possibly-uninitialized variables. The IFDS framework was defined in [23], where we presented an efficient exhaustive algorithm for solving IFDS problems. That definition is summarized below.

The IFDS framework is a variant of Sharir and Pnueli's "functional approach" to interprocedural dataflow analysis [25], with an extension similar to the one given by Knoop and Steffen in order to handle programs in which recursive procedures have local variables and parameters [17]. These frameworks generalize Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [16] to the "meet-over-all-valid-paths" solution of an *inter*procedural dataflow-analysis problem.

In Kildall's framework, an instance of a dataflow-analysis problem consists of a bounded lower semilattice (the dataflow information) with meet operator $\sqcap$, a flowgraph (representing the program), and an assignment of dataflow functions to the edges of the flowgraph. If all of the dataflow functions are distributive, Kildall's algorithm computes the meet-over-all-paths solution to the problem instance. Similarly, in the IFDS framework, an instance of a dataflow-analysis problem (or IFDS problem, for short) consists of the following:

- A finite set D (the dataflow information).
- A meet operator $\sqcap$. In general, the meet operator can be either union or intersection. For the purposes of this paper we will assume that the meet operator is union. (It is not hard to show that IFDS problems in which the meet operator is intersection can always be handled by transforming them to a complementary union problem.)
- A ***supergraph*** G^* (a collection of flowgraphs, one for each procedure). In supergraph G^*, a procedure call is represented by two nodes, a ***call*** node and a ***return-site*** node. In addition to the ordinary intraprocedural edges that connect the nodes of the individual flowgraphs, for each procedure call—represented by call-node c and return-site node r—G^* has three edges: an intraprocedural ***call-to-***

return-site edge from c to r; an interprocedural ***call-to-start*** edge from c to the start node of the called procedure; an interprocedural ***exit-to-return-site*** edge from the exit node of the called procedure to r.

(The call-to-return-site edges are included so that the IFDS framework can handle programs with local variables and parameters; the dataflow functions on call-to-return-site and exit-to-return-site edges permit the information about local variables and value parameters that holds at the call site to be combined with the information about global variables and reference parameters that holds at the end of the called procedure.)

- An assignment of distributive dataflow functions (of type $2^D \rightarrow 2^D$) to the edges of the supergraph.

Given an instance of an IFDS problem, a dataflow fact $d \in D$, and a flowgraph node n, the demand algorithm given in Section 3 determines whether fact d is in the meet-over-all-valid-paths solution at node n. The distinction between meet-over-all-paths and meet-over-all-*valid*-paths is necessary to capture the idea that not all paths in G^* represent potential execution paths. A ***valid path*** is one that respects the fact that a procedure always returns to the site of the most recent call. To understand the algorithm of Section 3, it is useful to distinguish further between a ***same-level valid path*** (a path in G^* that starts and ends in the same procedure, and in which every call has a corresponding return) and a ***valid path*** (a path that may include one or more unmatched calls).

Example. Figure 1 shows an example program and its supergraph G^*. In G^*, the path

$$start_{main} \rightarrow n1 \rightarrow n2 \rightarrow start_p \rightarrow n4 \rightarrow exit_p \rightarrow n3$$

is a (same-level) valid path; the path

$$start_{main} \rightarrow n1 \rightarrow n2 \rightarrow start_p \rightarrow n4$$

is a non-same-level valid path (because the call-to-start edge $n2 \rightarrow start_p$ has no matching exit-to-return-site edge); the path

$$start_{main} \rightarrow n1 \rightarrow n2 \rightarrow start_p \rightarrow n4 \rightarrow exit_p \rightarrow n8$$

is not a valid path because the exit-to-return-site edge $exit_p \rightarrow n8$ does not correspond to the preceding call-to-start edge $n2 \rightarrow start_p$.

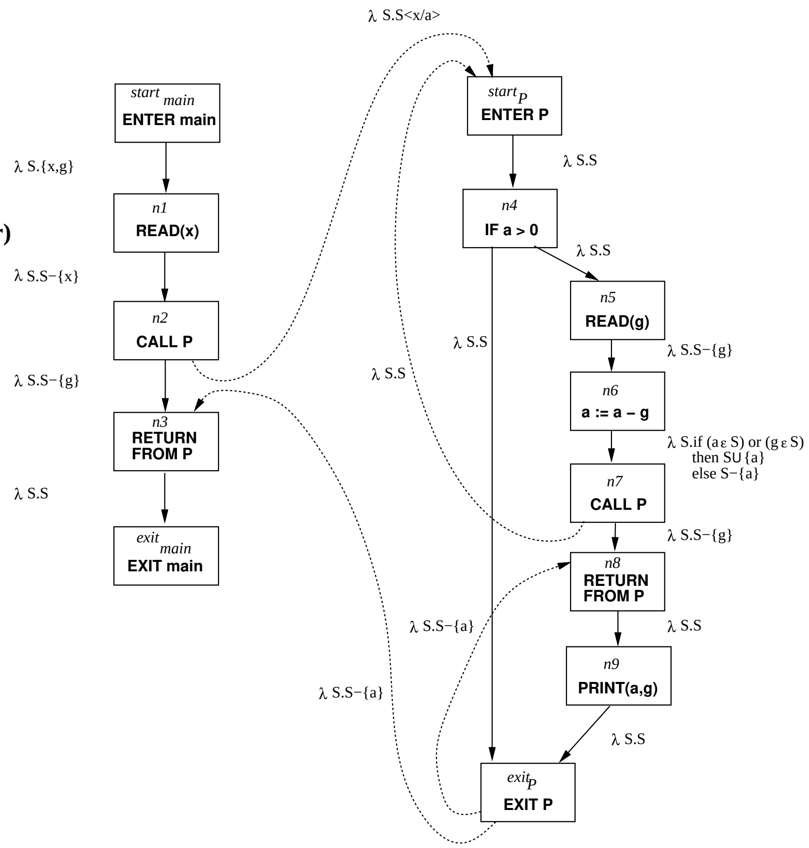
In Figure 1, the supergraph is annotated with the dataflow functions for the “possibly-uninitialized variables” problem. The “possibly-uninitialized variables” problem is to determine, for each node n in G^*, the set of program variables that may be uninitialized just before execution reaches n. A variable x is possibly uninitialized at n either if there is an x-definition-free valid path to n or if there is a valid path to n on which the last definition of x uses some variable y that itself is possibly uninitialized. For example, the dataflow function associated with edge $n6 \rightarrow n7$ shown in Figure 1 adds a to the set of possibly-uninitialized variables after node $n6$ if either a or g is in the set of possibly-uninitialized variables before node $n6$. □

declare *g*: **integer**

program *main*
begin
  declare *x*: **integer**
  read(*x*)
  call *P* (*x*)
end

procedure *P* (**value** *a* : **integer**)
begin
  if (*a* > 0) **then**
    read(*g*)
    $a := a - g$
    call *P* (*a*)
    print(*a*, *g*)
  fi
end

(a) Example program



(b) Its supergraph G^*

Figure 1. An example program and its supergraph G^*. The supergraph is annotated with the dataflow functions for the “possibly-uninitialized variables” problem. The notation S<x/a> denotes the set *S* with *x* renamed to *a*.

The IFDS framework can be used for languages with a variety of features (including procedure calls, parameters, global and local variables, and pointers). Encoding a problem in the IFDS framework may in some cases involve a loss of precision; for example, in languages with pointers there may be a loss of precision for problem instances in which there is aliasing. Once a problem has been encoded in the IFDS framework, the demand algorithm presented in this paper provides (with no further loss of precision) an efficient way to determine whether a particular dataflow fact is in the meet-over-all-valid-paths solution to the problem.

2.2. From Dataflow-Analysis Problems to Realizable-Path Reachability Problems

In this section, we show how to convert IFDS problems to “realizable-path” graph-reachability problems. This is done by transforming an instance of an IFDS problem (a supergraph G^* in which each edge has an associated distributive function in $2^D \rightarrow 2^D$) into an ***exploded supergraph*** $G^\#$, in which each node $\langle n, d \rangle$

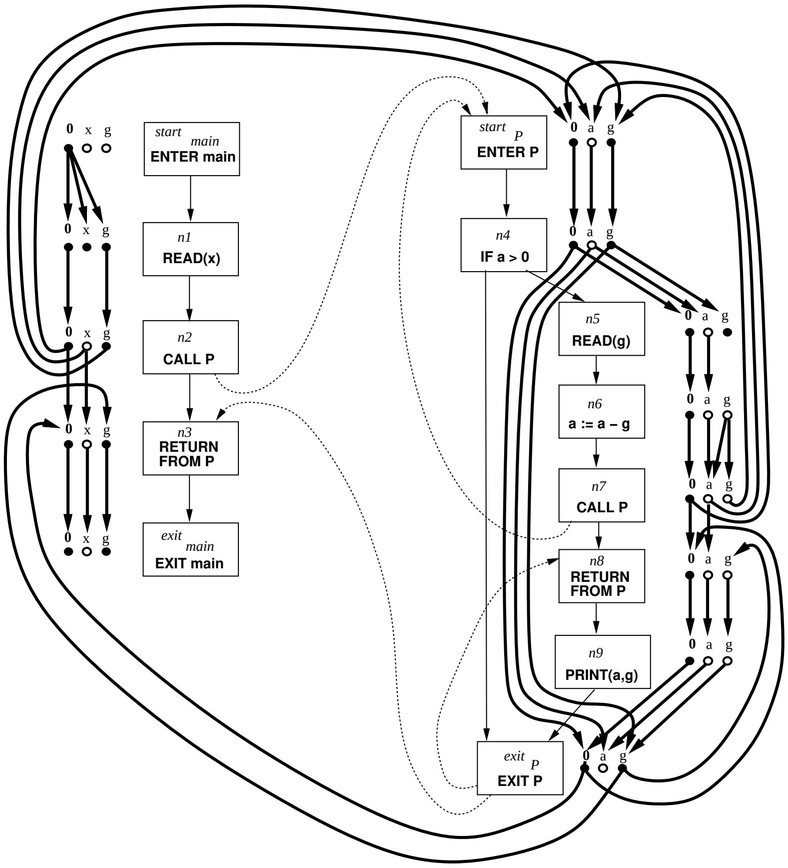


Figure 2. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1. Closed circles represent nodes of $G^{\#}$ that are reachable along realizable paths from $\langle start_{main}, \mathbf{0} \rangle$. Open circles represent nodes not reachable along such paths.

represents dataflow fact $d \in D$ at supergraph node n, and each edge represents a dependence between dataflow facts at different supergraph nodes.

The key insight behind this "explosion" is that a distributive function in $2^D \to 2^D$ can be represented using a graph with $2(D+1)$ nodes [23]; this graph is called the function's *representation relation*. Thus, each node n in G^* is "exploded" into $D + 1$ nodes in $G^{\#}$, and each edge $m \to n$ in G^* is "exploded" into a set of edges that represent the function associated with $m \to n$. In particular:

(i) For every node n in G^*, there is a node $\langle n, \mathbf{0} \rangle$ in $G^{\#}$.

(ii) For every node n in G^*, and every dataflow fact $d \in D$, there is a node $\langle n, d \rangle$ in $G^{\#}$.

Given function f associated with edge $m \to n$ of G^*:

(i) There is an edge in $G^{\#}$ from node $\langle m, \mathbf{0} \rangle$ to node $\langle n, d \rangle$ for every $d \in f(\emptyset)$.

(ii) There is an edge in $G^{\#}$ from node $\langle m, d_1 \rangle$ to node $\langle n, d_2 \rangle$ for every d_1, d_2 such that $d_2 \in f(\{ d_1 \})$ and $d_2 \notin f(\emptyset)$.

(iii) There is an edge in $G^{\#}$ from node $\langle m, \mathbf{0} \rangle$ to node $\langle n, \mathbf{0} \rangle$.

This collection of edges forms f's representation relation. The intuition behind this definition of the representation relation is that the edges represent the behavior of f "point-wise". In particular, (i) captures the fact that d is in the result of an application of f no matter what the input is; (ii) captures the fact that d_2 is only in the output set resulting from an application of f if d_1 is in the input set; (iii) is included for technical reasons so that the relational composition of the representation relations of functions f and g corresponds to the functional composition $g \circ f$ [20].

A ***realizable path*** in $G^{\#}$ is one that corresponds to a *valid path* in G^*. (Similarly, a ***same-level realizable path*** is one that corresponds to a *same-level valid path*.) That is, realizable paths respect the fact that a procedure always returns to the site of the most recent call. In [20] we have shown that dataflow fact d holds at supergraph node n iff there is a realizable path in $G^{\#}$ from node $\langle start_{main}, \mathbf{0} \rangle$ (which represents the fact that no dataflow facts hold at the start of procedure *main*) to node $\langle n, d \rangle$.

Example. The exploded supergraph that corresponds to the instance of the "possibly-uninitialized variables" problem shown in Figure 1 is shown in Figure 2. The dataflow functions are replaced by their representation relations. For example, the function $\lambda S. \{x, g\}$ on the edge out of node $start_{main}$ is represented by these three edges:

$$\langle start_{main}, \mathbf{0} \rangle \rightarrow \langle n1, \mathbf{0} \rangle$$
$$\langle start_{main}, \mathbf{0} \rangle \rightarrow \langle n1, x \rangle$$
$$\langle start_{main}, \mathbf{0} \rangle \rightarrow \langle n1, g \rangle$$

In Figure 2, closed circles represent nodes that are reachable along realizable paths from $\langle start_{main}, \mathbf{0} \rangle$. Open circles represent nodes not reachable along realizable paths. (For example, note that nodes $\langle n8, g \rangle$ and $\langle n9, g \rangle$ are reachable only along non-realizable paths from $\langle start_{main}, \mathbf{0} \rangle$.) As stated above, this information indicates the nodes' values in the meet-over-all-valid-paths solution to the dataflow-analysis problem. For instance, the meet-over-all-valid-paths solution at node $exit_p$ is the set $\{g\}$. (That is, variable g is the only possibly-uninitialized variable just before execution reaches the exit node of procedure p.) In Figure 2, this information can be obtained by determining that there is a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle exit_p, g \rangle$, but not from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle exit_p, a \rangle$. □

3. A Demand Algorithm for IFDS Problems

In this section, we show how to solve demand IFDS problems by solving equivalent realizable-path reachability demands. The algorithm, called the ***Demand-Tabulation Algorithm***, is presented in Figure 3. The top-level function of the algorithm is called IsMemberOfSolution. The call IsMemberOfSolution($\langle \bar{n}, \bar{d} \rangle$) returns *true* iff there is a realizable path from node $\langle start_{main}, \mathbf{0} \rangle$ to node $\langle \bar{n}, \bar{d} \rangle$ in $G^{\#}$. Such a path exists iff

```
declare
   G# = (N#, E#):                        global exploded supergraph
   PathEdge, SummaryEdge:                global edge set, initially empty                  /* These sets are preserved across calls */
   ReachableNodes:                       global node set, initially {⟨n, 0⟩ | n ∈ N*}      /* This set is preserved across calls */
   VisitedNodes:                         global node set, initially empty                  /* This set is preserved across calls */

function IsMemberOfSolution(⟨n̄, d̄⟩: exploded supergraph node) returns boolean
declare en: exploded supergraph node
begin
en = BackwardDFS(⟨n̄, d̄⟩)
if en = Failure then
   return(false)
else
   UpdateReachableNodes(en)
   return(true)
fi
end

function BackwardDFS(⟨n̄, d̄⟩) returns exploded supergraph node or Failure
declare EdgeWorkList: edge set;  NodeStack: node stack
begin
   push ⟨n̄, d̄⟩ onto NodeStack
   while NodeStack is not empty do
      pop a node ⟨n, d⟩ from NodeStack
      if ⟨n, d⟩ ∈ ReachableNodes then
         return(⟨n, d⟩)
      else if ⟨n, d⟩ ∉ VisitedNodes then
         insert ⟨n, d⟩ into VisitedNodes
         switch n
            case n is a return-site node :
               let c be the call node that corresponds to n, and let p be the procedure called at c
                  EdgeWorkList := ∅
                  for each d′ such that ⟨exit_p, d′⟩ → ⟨n, d⟩ ∈ E# do Propagate(⟨exit_p, d′⟩ → ⟨exit_p, d′⟩, EdgeWorkList) od
                  BackwardTabulateSLRPs(EdgeWorkList)
                  for each d′ such that ⟨c, d′⟩ → ⟨n, d⟩ ∈ (E# ∪ SummaryEdge) and ⟨c, d⟩ ∉ VisitedNodes do push ⟨c, d′⟩ onto NodeStack od
               end let
            end case
            case n is the start node of procedure p :
               for each c ∈ Callers(p) do
                  for each d′ such that ⟨c, d′⟩ → ⟨n, d⟩ ∈ E# and ⟨c, d′⟩ ∉ VisitedNodes do push ⟨c, d′⟩ onto NodeStack od
               od
            end case
            default :
               for each ⟨m, d′⟩ such that ⟨m, d′⟩ → ⟨n, d⟩ ∈ E# and ⟨m, d′⟩ ∉ VisitedNodes do push ⟨m, d′⟩ onto NodeStack od
            end case
         end switch
      fi
   od
   return(Failure)
end

procedure UpdateReachableNodes(⟨n_0, d_0⟩: exploded supergraph node)
declare NodeWorkList: node set = ∅
declare ⟨n, d⟩, ⟨m, d′⟩: exploded supergraph node
begin
   insert ⟨n_0, d_0⟩ into NodeWorkList
   while NodeWorkList ≠ ∅ do
      Select and remove an exploded supergraph node ⟨n, d⟩ from NodeWorkList
      insert ⟨n, d⟩ into ReachableNodes
      remove ⟨n, d⟩ from VisitedNodes
      for each ⟨m, d′⟩ such that (⟨n, d⟩ → ⟨m, d′⟩ ∈ (E# ∪ SummaryEdge)) and ⟨n, d⟩ → ⟨m, d′⟩ is not an exit-to-return-site edge, and
                                  (⟨m, d′⟩ ∈ VisitedNodes) and (⟨m, d′⟩ ∉ ReachableNodes) do
         Insert ⟨m, d′⟩ into NodeWorkList
      od
   od
end
```

Figure 3. The Demand-Tabulation Algorithm determines whether dataflow fact $\bar{d}$ holds at flowgraph node $\bar{n}$. Procedures BackwardTabulateSLRPs and Propagate are given in Figure 4.

the meet-over-all-valid-paths solution to the dataflow-analysis problem at node $\bar{n}$ of G^* includes dataflow fact $\bar{d}$.

IsMemberOfSolution consists of a backward phase (performed by function BackwardDFS) followed by a forward phase (performed by procedure UpdateReachableNodes). BackwardDFS performs a backward depth-first search of $G^\#$ starting from "demand" node $\langle \bar{n}, \bar{d} \rangle$, to determine whether the demand node can be reached via a realizable path from node $\langle start_{main}, \mathbf{0} \rangle$. UpdateReachableNodes is called only when BackwardDFS is successful. The purpose of UpdateReachableNodes is to update two sets, ReachableNodes and VisitedNodes, that are maintained across calls to IsMemberOfSolution in order to prevent repeating work done on a previous call.

The ReachableNodes and VisitedNodes sets are used and maintained as follows:

ReachableNodes

An exploded-graph node $\langle n, d \rangle$ is placed in set ReachableNodes when it has been determined that there is a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle n, d \rangle$. Before the first call on IsMemberOfSolution is performed, ReachableNodes is initialized to $\{ \langle n, \mathbf{0} \rangle \}$ for all supergraph nodes n.

As soon as function BackwardDFS encounters a node $\langle n, d \rangle$ that is in ReachableNodes, the backward depth-first search is terminated, and node $\langle n, d \rangle$ is returned (line [12]). (The fact that $\langle n, d \rangle$ is reachable via a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ together with the fact that BackwardDFS only visits nodes from which there is a realizable path to the "demand" node $\langle \bar{n}, \bar{d} \rangle$, means that there is a realizable path from $\langle start_{main}, \mathbf{0} \rangle$ to $\langle \bar{n}, \bar{d} \rangle$.)

Procedure UpdateReachableNodes starts at the node $\langle n, d \rangle$ returned by BackwardDFS, and performs a forward traversal of $G^\#$, following only the edges that were traversed backwards by BackwardDFS. All of the nodes that it encounters are reachable from $\langle start_{main}, \mathbf{0} \rangle$ via a realizable path, so they are added to ReachableNodes. (This update of ReachableNodes is part of what makes the Demand-Tabulation Algorithm a caching algorithm. The algorithm would still be correct if UpdateReachableNodes followed all edges, but it would increase the time required for a single demand.)

VisitedNodes

Between invocations of IsMemberOfSolution, the exploded-graph nodes in set VisitedNodes are those for which it has been determined that there is *no* realizable path from $\langle start_{main}, \mathbf{0} \rangle$. During an invocation of IsMemberOfSolution, nodes visited for the first time are added to this set (line [14]); those determined to be reachable from $\langle start_{main}, \mathbf{0} \rangle$ by a realizable path are transferred from this set to the ReachableNodes set by procedure UpdateReachableNodes (lines [39] and [40]). Note that when BackwardDFS returns *Failure*, none of the nodes added to VisitedNodes is reachable from $\langle start_{main}, \mathbf{0} \rangle$, so it is not necessary for IsMemberOfSolution to call UpdateReachableNodes.

An interesting aspect of BackwardDFS is how it ensures that only nodes from which there is a realizable path to demand node $\langle \bar{n}, \bar{d} \rangle$ are visited. This is accomplished by the call to BackwardTabulateSLRPs at line [20], which occurs when the node $\langle n, d \rangle$ popped from the stack corresponds to a return-site node (*i.e.*, n is a return-site node in G^*). The purpose of BackwardTabulateSLRPs is to find ***summary edges***, which represent transitive dependences due to procedure calls: A summary edge of the form $\langle c, d_1 \rangle \rightarrow \langle r, d_2 \rangle$ (where c is a call node and r is the matching return-site node) represents a same-level realizable path from $\langle c, d_1 \rangle$ to $\langle r, d_2 \rangle$. Summary edges are recorded in the (global) set named SummaryEdge. After calling BackwardTabulateSLRPs, BackwardDFS can continue its backward traversal across the newly discovered summary edges (line [21]).

BackwardDFS calls several auxiliary subprograms: function Callers(p) returns the set of call nodes that

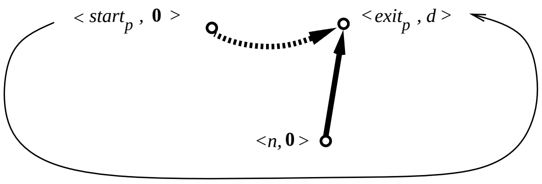
```
declare G# = (N#, E#):                global exploded supergraph
declare PathEdge, SummaryEdge: global edge set, initially empty      /* These sets are preserved across calls */
procedure BackwardTabulateSLRPs(EdgeWorkList: edge set)
begin
  while EdgeWorkList ≠ ∅ do
    Select and remove an edge ⟨n, d1⟩ → ⟨exit_p, d⟩ from EdgeWorkList
    if (⟨n, d1⟩ is ⟨start_p, 0⟩) or (⟨start_p, 0⟩ → ⟨exit_p, d⟩ ∉ PathEdge) then
      switch n
        case n a return-site node :
          let c be the call node that corresponds to n, and q be the procedure called at c
            for each d2 such that ⟨exit_q, d2⟩ → ⟨n, d1⟩ ∈ E# do Propagate(⟨exit_q, d2⟩ → ⟨exit_q, d2⟩, EdgeWorkList) od
            for each d2 such that ⟨c, d2⟩ → ⟨n, d1⟩ ∈ (E# ∪ SummaryEdge) do Propagate(⟨c, d2⟩ → ⟨exit_p, d⟩, EdgeWorkList) od
          end let
        end case
        case n the start node of procedure p :
          for each c ∈ Callers(p) do
            let q be c's procedure, and r be the return-site node that corresponds to c
              for each d3, d4 such that ⟨c, d4⟩ → ⟨n, d1⟩ ∈ E# and ⟨exit_p, d⟩ → ⟨r, d3⟩ ∈ E# do
                if ⟨c, d4⟩ → ⟨r, d3⟩ ∉ SummaryEdge then
                  Insert ⟨c, d4⟩ → ⟨r, d3⟩ into SummaryEdge
                  for each d2 such that ⟨r, d3⟩ → ⟨exit_q, d2⟩ ∈ PathEdge do Propagate(⟨c, d4⟩ → ⟨exit_q, d2⟩, EdgeWorkList) od
                fi
              od
            end let
          od
        end case
        default :
          for each m, d2 such that ⟨m, d2⟩ → ⟨n, d1⟩ ∈ E# do Propagate(⟨m, d2⟩ → ⟨exit_p, d⟩, EdgeWorkList) od
        end case
      end switch
    fi
  od
end
procedure Propagate(⟨n, d1⟩ → ⟨exit_p, d⟩: edge, EdgeWorkList: edge set)
begin
  if d1 is 0 then
    n := start_p
  fi
  if ⟨n, d1⟩ → ⟨exit_p, d⟩ ∉ PathEdge then
    Insert ⟨n, d1⟩ → ⟨exit_p, d⟩ into PathEdge
    Insert ⟨n, d1⟩ → ⟨exit_p, d⟩ into EdgeWorkList
  fi
end
```

Figure 4. Procedure BackwardTabulateSLRPs finds summary edges and records them in set SummaryEdge.

represent calls on p; procedures Propagate and BackwardTabulateSLRPs are shown in Figure 4. As discussed above, the purpose of BackwardTabulateSLRPs is to find summary edges, and to record them in the set named SummaryEdge. In order to do this, BackwardTabulateSLRPs finds ***path edges*** (which represent same-level realizable paths in $G^{\#}$) whose targets are nodes of the form $\langle exit_p, d \rangle$ (*i.e.*, nodes of $G^{\#}$ that correspond to exit nodes of G^{*}). It records all such path edges in the (global) set named PathEdge.

Procedure BackwardTabulateSLRPs is a worklist algorithm that starts with an initial worklist containing a set of zero-length path edges (edges of the form $\langle exit_p, d \rangle \rightarrow \langle exit_p, d \rangle$); on each iteration of the main loop it deduces the existence of additional path edges and summary edges.

In terms of leading quickly to a "yes" answer to a demand, the best thing that can happen in BackwardTabulateSLRPs is to discover a path edge whose source is a "**0**" node (*i.e.*, a path edge of the form $\langle n, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$). In this case, the answer to the current demand is guaranteed to be "yes." However, BackwardTabulateSLRPs cannot simply quit, because it is vital that the PathEdge and SummaryEdge sets be left in a consistent state to ensure that subsequent calls to IsMemberOfSolution return the correct answer. In particular, BackwardTabulateSLRPs must finish finding all path edges whose targets are some node *other* than $\langle exit_p, d \rangle$. On the other hand, there is no need to process any more path edges to $\langle exit_p, d \rangle$. Therefore, on discovering such an edge, BackwardTabulateSLRPs inserts the path edge $\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$ into PathEdge and into the worklist (lines [30], [33] and [34]). This situation is illustrated below. The solid bold arrow represents the path edge whose source is a "**0**" node, and the dotted bold arrow represents the new path edge that is inserted into PathEdge and the worklist.



Furthermore, when a path edge is taken off the worklist (line [2]) it is processed only if it is itself of the form $\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$, or if that path edge has not yet been discovered.

The configurations that are used by BackwardTabulateSLRPs to deduce the existence of path edges and summary edges are depicted in Figure 5. The first two diagrams of Figure 5 correspond to the case where n is a return-site node; the next two diagrams correspond to the case where n is a start node; and the final diagram corresponds to the default case. In Figure 5, the bold dotted arrows represent edges that are inserted into sets PathEdge and SummaryEdge if they were not previously in those sets.

Example. When IsMemberOfSolution is called with the exploded supergraph node $\langle n9, g \rangle$ from the example shown in Figure 2, (*i.e.*, the demand "Might g be uninitialized at node $n9$?" is made), the following steps are performed (all line numbers refer to lines in Figure 3):

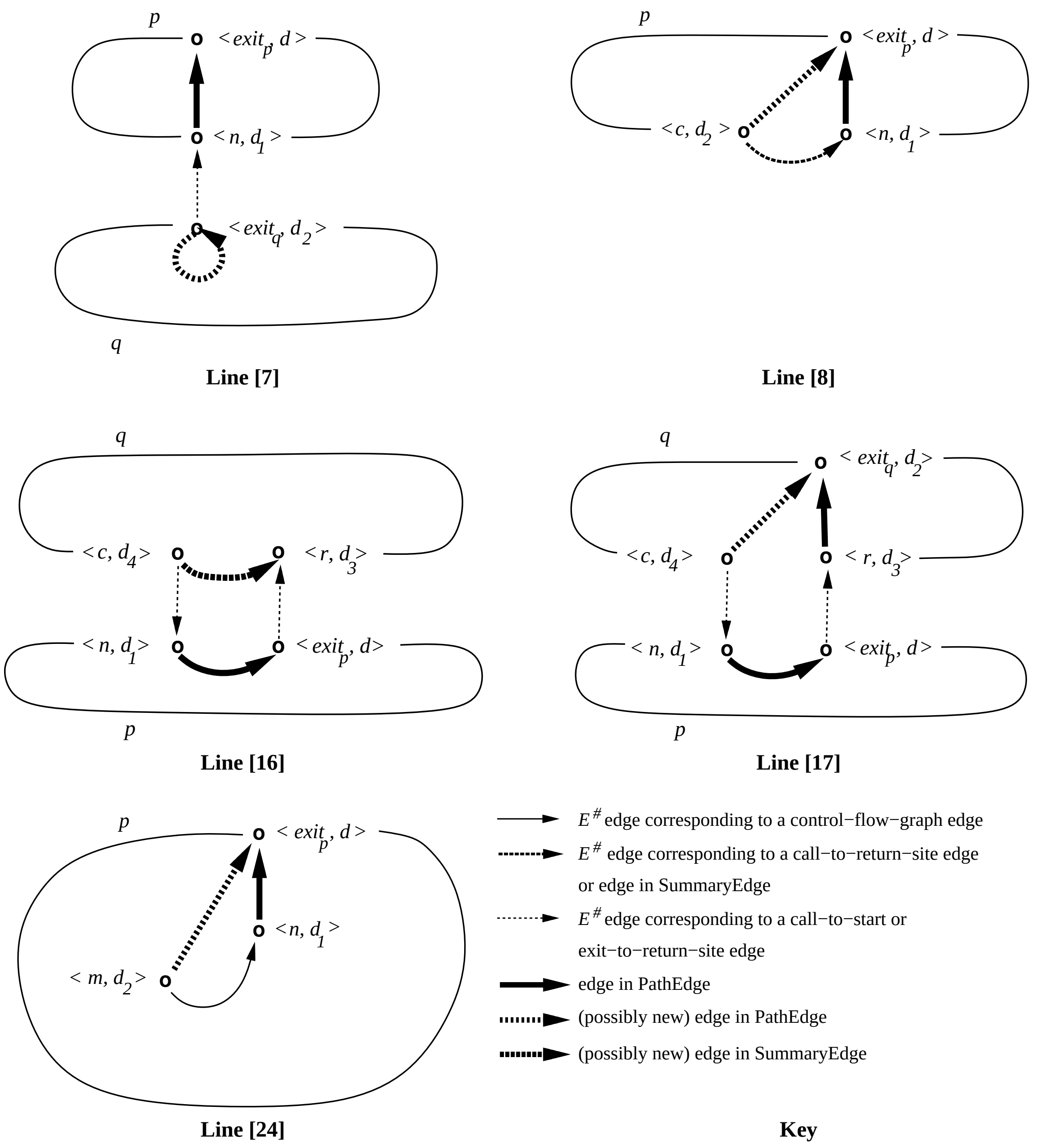


Figure 5. These five diagrams show how procedure BackwardTabulateSLRPs deduces the existence of new path and summary edges.

1. BackwardDFS is called with node $\langle n9, g \rangle$.
2, Node $\langle n9, g \rangle$ is pushed onto NodeStack at line [8], and then popped off (into $\langle n, d \rangle$) at line [10].
3. Node $\langle n9, g \rangle$ is inserted into VisitedNodes at line [14]. The default case of the switch (line [29]) is taken, and node $\langle n8, g \rangle$ is pushed onto NodeStack.

4. Node $\langle n8, g\rangle$ is popped from NodeStack; *n8* is a return-site node, so the case on line [16] is selected, and BackwardTabulateSLRPs is called for the first time. This causes summary edge $\langle n7, g\rangle \rightarrow \langle n8, g\rangle$ to be inserted into SummaryEdge (and causes several other edges to be inserted into PathEdge).
5. Node $\langle n7, g\rangle$ is pushed onto NodeStack at line [21].
6. Node $\langle n7, g\rangle$ is popped from NodeStack; the default case is taken, and node $\langle n6, g\rangle$ is pushed onto NodeStack.
7. Node $\langle n6, g\rangle$ is popped from NodeStack; the default case is taken, but there are no edges that satisfy the for-loop condition (line [30]).
8. NodeStack is now empty, so BackwardDFS returns *Failure*, and IsMemberOfSolution returns *false*. □

3.1. Cost Of The Demand-Tabulation Algorithm

In the worst case, a single demand will cause the Demand-Tabulation Algorithm to traverse every edge of the exploded supergraph as well as every summary edge. To express this cost in terms of the size of the (unexploded) supergraph, we will use the following parameters:

N	the number of nodes in supergraph G^*
E	the number of edges in supergraph G^*
Call	the number of call nodes in supergraph G^*
D	the size of set D

The maximum number of exploded supergraph and summary edges (and thus, the worst-case running time of the Demand-Tabulation Algorithm) varies depending on what class of dataflow-analysis problems is being solved. We have already mentioned the locally separable problems; it is also useful to define the class of *h*-sparse problems:

Definition 3.1. A problem is ***h-sparse*** if all problem instances have the following property: For each function f on an ordinary intraprocedural edge or a call-to-return-site edge of G^*, the number of edges in $G^\#$ that represent function f, excluding edges that emanate from the **0** node, is at most hD. □

In general, when the nodes of G^* represent individual statements and predicates (rather than basic blocks), and when there is no aliasing, we expect most distributive problems to be *h*-sparse (with $h \ll D$): Each statement changes only a small portion of the execution state, and accesses only a small portion of the state as well. Therefore, the dataflow functions, which are abstractions of the statements' semantics, should be "close to" the identity function. The identity function is represented using $D + 1$ edges; thus, the number of edges needed to represent each dataflow function should be roughly D.

Example. When the nodes of G^* represent individual statements and predicates, and there is no aliasing, every instance of the possibly-uninitialized variables problem is 2-sparse. The only non-identity dataflow functions are those associated with assignment statements. The outdegree of every non-**0** node in the representation of such a function is at most two: a variable's initialization status can affect itself and at most one other variable, namely the variable assigned to.

□

Another parameter that affects the running time of the Demand-Tabulation Algorithm is the "bandwidth" for the transmission of dataflow information between procedures. In particular, the times given here rely on the fact that it is always possible to construct $G^{\#}$ so the maximum outdegree of a non-**0** node in a call-to-start edge's representation relation, and the maximum indegree of a non-**0** node in an exit-to-return-site edge's representation relation are both 1. (See the Appendix of [23] for a more complete discussion of this issue.) The implementation reported in Section 4 constructs $G^{\#}$ so that these properties hold.

The table in Figure 6 summarizes how the Demand-Tabulation Algorithm behaves for six different classes of problems. In each case, the time given is the worst-case time for a single demand. The details of the analysis of the running time of the Demand-Tabulation Algorithm can be found in [20].

The most efficient exhaustive algorithm known for the class of IFDS problems is the one given in [23]. Its worst-case running times are almost identical to the times given above; the only difference is that for an intraprocedural, locally separable problem, the bound for the exhaustive algorithm is $O(ED)$, while the bound for the Demand-Tabulation Algorithm is $O(E)$. The similarity in the worst-case running times of the two algorithms reflects the fact that (theoretically) a dataflow fact at one point might depend on all other facts at all other points. In practice, however, we have found that the Demand-Tabulation Algorithm (applied to a single demand) is much faster than the exhaustive algorithm (see Figure 8).

3.2. The Same-Worst-Case-Cost Property

We have designed the Demand-Tabulation Algorithm so that it has the same-worst-case-cost property with respect to the exhaustive algorithm of [23]. In particular, a call to IsMemberOfSolution can re-use the sets ReachableNodes, VisitedNodes, PathEdge, and SummaryEdge, whose values are preserved across calls. When the Demand-Tabulation Algorithm is used with a request sequence that places demands on all nodes of $G^{\#}$, BackwardDFS and UpdateReachableNodes will each traverse a given edge in $G^{\#}$ at most once during the processing of the request sequence. BackwardTabulateSLRPs will traverse a given summary edge or an edge of $G^{\#}$ in procedure p at most D times: once for each node of the form $\langle exit_p, d \rangle$. (The

Class of functions	Graph-theoretic characterization of the dataflow functions' properties	Asymptotic running time	
		Intraprocedural problems	Interprocedural problems
Distributive	Up to $O(D^2)$ edges/representation-relation	$O(ED^2)$	$O(ED^3)$
h-sparse	At most $O(hD)$ edges/representation-relation	$O(hED)$	$O(Call\ D^3 + hED^2)$
(Locally) separable	Component-wise dependences	$O(E)$	$O(ED)$

Figure 6. Asymptotic running time of the Demand-Tabulation Algorithm (for answering a single demand) for six different classes of dataflow-analysis problems.

information accumulated in sets PathEdge and SummaryEdge prevents procedure BackwardTabulateSLRPs from performing additional work, and the information accumulated in ReachableNodes and VisitedNodes prevents BackwardDFS and UpdateReachableNodes from performing additional work.) In general, this is bounded by $O(E\,D^3)$, which is the same amount of work that could be performed in the worst case by the exhaustive algorithm given in [23]. Thus, the Demand-Tabulation Algorithm has the same-worst-case-cost property with respect to the exhaustive algorithm.

While this is an important property, it does not, of course, mean that the Demand-Tabulation Algorithm will always outperform the exhaustive algorithm. There will be problem instances for which the exhaustive algorithm will not achieve its worst-case cost, and it may outperform the Demand-Tabulation Algorithm in those cases (see Figure 9).

4. Experimental Results

4.1. Background to the Experiments

We have carried out two experiments to compare the performance of the Demand-Tabulation Algorithm to that of the exhaustive algorithm of [23], and two further experiments to study the trade-off between the benefit and overhead of the caching performed by the Demand-Tabulation Algorithm. In all of our reported results, running times reflect the trimmed mean of five data points (*i.e.*, all experiments were run five times, and the average running times were computed after discarding the high and low values).

Three different analysis algorithms were used in the study: (1) the Demand-Tabulation Algorithm, as described above, (2) a non-caching version of the Demand-Tabulation Algorithm (that returns *true* as soon as it visits a node of the form $\langle n, \mathbf{0} \rangle$, reinitializes the set VisitedNodes to $\emptyset$ after each invocation of IsMemberOfSolution, does not maintain the set ReachableNodes, but *does* preserve the sets PathEdge and SummaryEdge across calls to IsMemberOfSolution),[1] and (3) the exhaustive algorithm reported in [23].

The three algorithms described above were implemented in C and used with a front end that analyzes a C program and generates the corresponding exploded supergraph for five dataflow-analysis problems:

Possibly-Uninitialized Variables

This is the problem that we have used as our running example.

Simple Uninitialized Variables

This is the locally separable version of the possibly-uninitialized variables problem, in which a variable is considered to be initialized whenever it is the target of an assignment, regardless of whether the

---

[1]The non-caching algorithm does not have the same-worst-case-cost property with respect to the exhaustive algorithm. In the worst case, on a request sequence that places demands on all nodes of $G^{\#}$, the non-caching algorithm could perform as much as $\Omega(N\,E\,D^3)$ work, which is worse than the $O(E\,D^3)$ bound on the work performed by the exhaustive algorithm.

right-hand-side expression includes possibly-uninitialized variables. (So every *simple* uninitialized variable is also *possibly* uninitialized, but not vice versa.)

Live Variables

This is the standard, locally separable problem in which variable x is considered to be live at supergraph node n iff there is a path from n to the end of the program on which x is used before being defined. It is useful to identify assignments to non-live variables: Programming tools might flag them as indicating possible logical errors, and optimizing compilers can remove them.

Truly-Live Variables

This is a non-locally-separable (and more accurate) version of the live-variables problem in which variable x is considered to be truly live at supergraph node n iff there is a path from n to the end of the program on which x is used in a truly-live context before being defined, where a truly-live context means: in a predicate, or in a call to a library routine, or in an expression whose value is assigned to a truly-live variable [13]. Because it is non-locally-separable, the truly-live variables problem is in some sense a harder problem than the live-variables problem, and its results are more accurate (every truly-live variable is also live, but not vice versa). Some assignments to variables that are live but not truly live can be discovered by repeatedly solving the live-variables problem and removing assignments to non-live variables until no more assignments to non-live variables are found. However, there are two potential disadvantages to solving the live variables problem rather than the truly-live variables problem: the problem needs to be solved repeatedly, and in the presence of cycles in the control-flow graph, there may be assignments to non-truly-live variables that are never discovered.

Constant Predicates

This is a non-locally-separable problem that seeks to determine, for every predicate that consists of a single identifier, whether that predicate is guaranteed to have a constant value (either *true*—non-zero—or *false*—zero). To do this, it performs a simple kind of copy-constant propagation, tracking, for every scalar variable x, whether x might be non-zero, zero, or $\perp$ (an unknown value). Given a predicate that consists of just the identifier x, if the dataflow fact $<x$, non-zero$>$ holds at that points, while neither fact $<x$, zero$>$, nor fact $<x, \perp>$ holds at that point, then the predicate is guaranteed to be *true* (and similarly, it is possible to determine when the predicate is guaranteed to be *false*).

In our experiments, procedure calls via pointers to procedures, and aliasing due to pointers were handled by our C front end as follows:

- For all of the dataflow-analysis problems, every call via a pointer was considered to be a possible call to every procedure of an appropriate type that was passed as a parameter or whose value was assigned to a variable somewhere in the program.
- For all of the dataflow-analysis problems, every memory write via a pointer was considered to be a possible write of every piece of heap-allocated storage and of every variable to which the “address-of” operator (&) was applied somewhere in the program.

- For the live variables, truly-live variables, and constant-predicates problems, every memory read via a pointer was considered to be a possible read of every piece of heap-allocated storage and of every variable to which the "address-of" operator was applied somewhere in the program.
- For the possibly-uninitialized variables problem, memory reads were considered to read only the value of the pointer itself. This is because the results of this analysis are suitable for providing feedback to the programmer rather than for guiding an optimizing compiler; it is more important to avoid overwhelming the programmer by reporting hundreds of possibly-uninitialized variables than to be sure that absolutely every possibly-uninitialized variable has been reported. (For the simple uninitialized-variables problem, reads via pointers are irrelevant, since a variable that is the target of an assignment is considered to be initialized regardless of which variables are used to compute the assigned value.)

Tests were carried out on a Sun SPARCstation 20 Model 71 with 64 MB of RAM. The study used 44 C programs—some standard UNIX utilities and some programs from the SPEC integer benchmark suite [1]. For each program, the table in Figure 7 gives the number of lines of preprocessed source code (with blank lines removed), the parameters that characterize the size of the control-flow graphs (number of procedures, number of call sites, number of control-flow graph nodes), and, for each of the five dataflow-analysis problems, the number of "interesting" program-point/dataflow-fact pairs (see below), and the number of these pairs that are in the meet-over-all-valid-paths solution (*i.e.*, if demands are made for all "interesting" program-point/dataflow-fact pairs, this is the number of demands that would be answered *yes*). The idea behind the notion of "interesting" program-point/dataflow-fact pairs is that the full set of dataflow facts at all points may not be required, and hence demands need only be issued for certain pairs. In particular, for the two versions of the "uninitialized-variables" problem, every use of a scalar variable x gives rise to the demand "might x be uninitialized here?"; for the two versions of the "live variables" problem, every assignment to a scalar variable x gives rise to the demand "is x live here?"; and for the "constant predicates" problem, every instance of a predicate that consists only of the identifier x gives rise to three demands: "might x be zero here?", "might x be non-zero here?", and "might x be $\perp$ here?".

4.2. Experiments

Experiment 1: Single demand vs. Exhaustive

Our first two experiments compared the Demand-Tabulation Algorithm with the exhaustive algorithm. Our first experiment reflects what might happen when dataflow analysis is used in the context of a tool that intersperses demands and program modifications (so if an exhaustive algorithm is used, it must be re-run whenever a demand is made following a modification). In this case, it is reasonable to compare the time required by the exhaustive algorithm with the time required by the demand algorithm to answer a single demand. Therefore, for this study, we recorded the following data for each dataflow-analysis problem and

Example	Lines of source code	CFG statistics			uninit-vars statistics			live-vars statistics			const-preds statistics	
		P	*Call*	*N*	*# demands*	*# yes* *possibly uninit*	*# yes* *simple uninit*	*# demands*	*# yes* *truly live*	*# yes* *live*	*# demands*	*# yes*
xref	68	7	12	193	65	13	0	88	79	81	42	14
queens	111	3	3	392	97	0	0	180	165	165	51	22
hash	146	5	8	216	49	1	1	90	74	76	12	4
misr	216	5	7	387	127	22	0	176	158	158	9	3
exptree	238	12	16	430	171	0	0	177	154	158	54	21
dry	241	13	16	324	106	6	5	116	83	96	6	2
chomp	262	20	43	894	247	0	0	361	317	319	54	28
diff.diffh	303	13	48	644	219	4	4	320	277	277	57	22
genetic	336	16	31	678	245	28	0	290	232	236	21	18
allroots	427	6	18	502	132	0	0	220	182	184	6	2
ul	451	13	34	1354	310	2	1	578	514	514	195	73
compress	657	14	27	1732	428	0	0	772	679	682	165	67
stanford	665	46	78	1422	625	6	3	597	498	523	51	24
clinpack	695	11	42	1251	509	37	1	579	481	525	12	4
travel	725	14	22	546	316	11	5	290	265	267	75	40
lex315	747	16	101	1204	235	2	2	448	435	437	138	80
sim	748	14	46	2245	1825	583	6	1219	1166	1167	411	159
mway	806	20	41	1822	732	147	12	795	672	689	63	27
pokerd	1099	24	83	2195	558	74	6	978	855	857	192	77
ansitape	1222	35	107	2324	412	0	0	1006	842	843	168	75
loader	1255	29	78	2137	540	60	5	892	884	884	165	72
gcc.main	1285	30	96	2385	827	8	1	1127	1016	1016	309	143
voronoi	1394	46	103	1415	696	3	3	686	577	610	168	69
ratfor	1531	51	265	2749	847	232	1	1245	941	946	411	142
livc	1674	85	202	4456	1403	0	0	2050	1603	1612	24	10
struct.beauty	1701	32	211	2758	708	18	6	1273	1060	1066	327	127
diff.diff	1761	40	125	4065	1178	75	4	1839	1538	1540	465	213
xmodem	1809	25	152	3415	743	21	10	1487	1232	1232	543	328
compiler	1908	37	348	3687	613	126	0	1539	1171	1175	714	448
learn.learn	1954	33	76	3827	589	9	0	1472	1169	1169	309	152
gnugo	1963	27	87	3423	1147	120	39	1409	1144	1146	624	260
triangle	1968	17	41	3427	3217	104	29	1765	1610	1622	582	215
football	2075	57	256	6018	2232	36	12	2936	2889	2889	687	329
dixie	2439	35	85	2906	693	19	8	1255	1130	1130	222	89
eqntott	2470	60	214	4701	1674	291	14	2096	1892	1895	405	185
twig	2555	75	221	4347	1278	16	1	2049	1813	1817	312	121
cdecl	2577	31	202	3420	730	2	0	1520	1346	1383	333	160
lex	2645	61	328	6666	2577	201	4	3192	2748	2800	861	363
patch	2746	53	263	5213	1569	216	41	2592	2433	2440	771	391
assembler	2994	51	246	5217	1185	28	2	2229	2153	2165	516	185
unzip	3261	39	125	3583	1231	185	16	1712	1556	1562	462	229
tbl	3462	82	313	6495	2147	135	15	2964	2562	2562	1119	588
gcc.cpp	4079	72	310	7545	3538	292	5	3687	3247	3285	1425	634
simulator	4239	98	407	5859	1317	310	9	2214	2178	2178	996	463

Figure 7. Test program information.

for each test program: (1) the time used by the exhaustive algorithm to find the meet-over-all-valid-paths solution; (2) the average time used by the Demand-Tabulation Algorithm to answer a single demand (using 20 randomly selected demands). This data is summarized in Figure 8.

Although there are a few cases where the Demand-Tabulation Algorithm is slower than the exhaustive algorithm (for the two versions of the “uninitialized-variables” problem), they are all on very small test files, where the running times are trivial anyway. It seems clear that overall, the Demand-Tabulation

Experiment 1: Single Demand vs Exhaustive

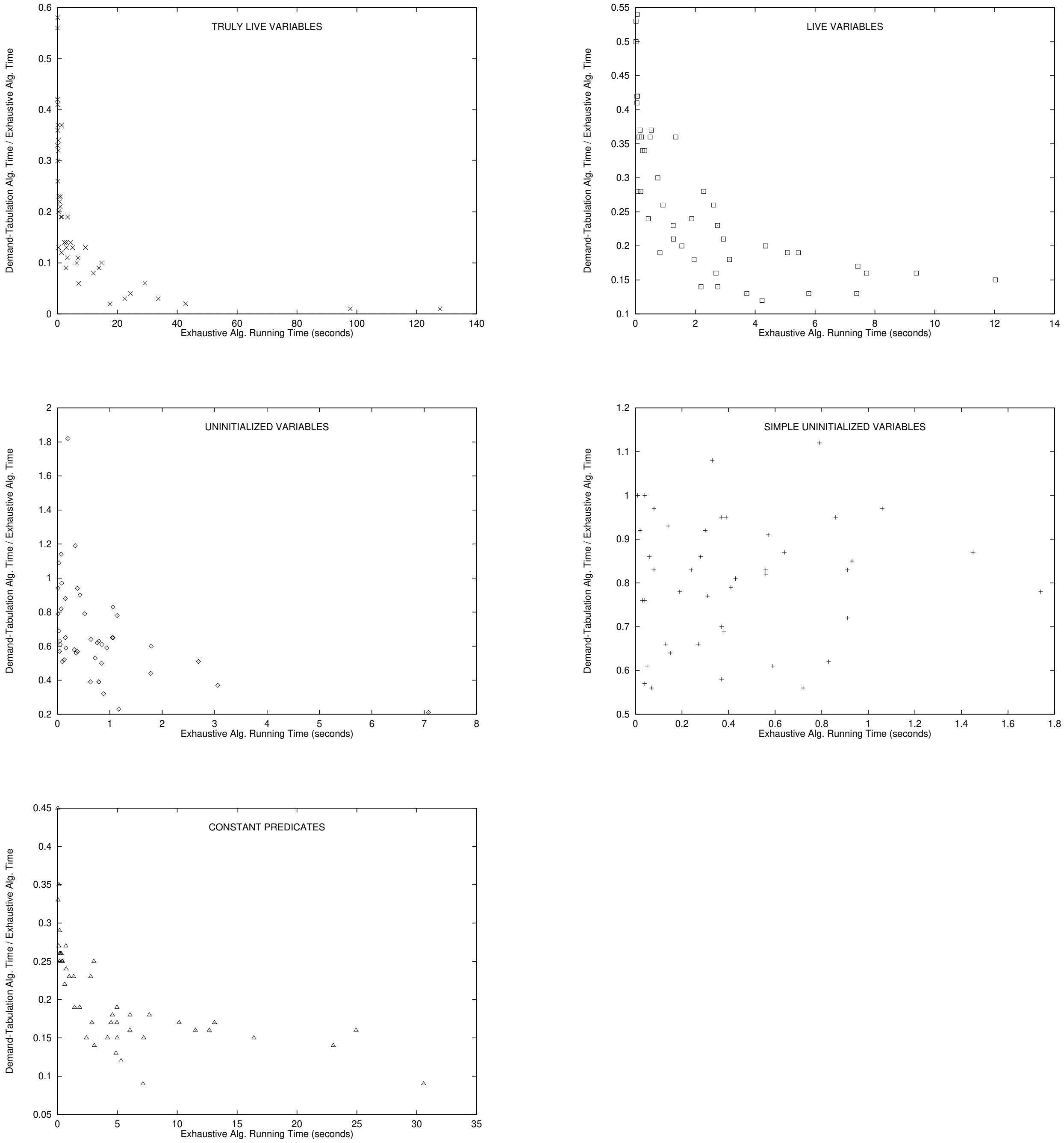


Figure 8. First comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm. The exhaustive algorithm is used to find the entire meet-over-all-valid-paths solution, and the Demand-Tabulation Algorithm is used to answer a single demand.

Algorithm is preferable to the exhaustive algorithm when the goal is to answer a single demand.

Experiment 2: Sequence of demands vs. Exhaustive

Our second comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm reflects what happens when “complete” dataflow information is desired (*i.e.*, when it is desired to know, for all “interesting” program-point/dataflow-fact pairs, whether that pair is in the meet-over-all-valid-paths solution). Therefore, for this study we recorded the time used by the Demand-Tabulation Algorithm to answer the sequence of all “interesting” demands, for each dataflow-analysis problem and for each test program, and compared those times to the times required by the exhaustive algorithm. This data is summarized in Figure 9.

The Demand-Tabulation Algorithm outperforms the exhaustive algorithm in all cases for the *live-variables* and the *constant-predicates* problems. There are a few cases where the situation is reversed for the *truly-live variables* problem, but these are only for programs with trivial running times. However, for the two versions of the *uninitialized-variables* problem, the Demand-Tabulation Algorithm is often significantly slower than the exhaustive algorithm.

We believe that the reason the Demand-Tabulation Algorithm outperforms the exhaustive algorithm in this experiment for the two *live-variables* problems but not for the two *uninitialized-variables* problems has to do with the percentage of demands with “yes” answers. In the case of the *live-variables* problems, most of the demands (“is x live at this assignment?”) lead to a *yes* answer, while in the case of the *uninitialized-variables* problems, most of the demands (“might x be uninitialized at this use?”) lead to a *no* answer. Demands answered *no* correspond to unreachable exploded supergraph nodes, so the exhaustive algorithm does not visit those nodes or any of their predecessors; however, the demand algorithm starts at those nodes and visits *all* predecessors, eventually discovering that none of them is in the ReachableNodes set. (The problem for which the Demand-Tabulation Algorithm is most clearly superior to the exhaustive algorithm in this experiment is the *constant-predicates* problem. While this may be partly due to the percentage of demands with *yes* answers, it is also due to the fact that, since demands are only made for instances of identifier predicates, the *number* of demands is very small relative to the size of the exploded graph. This means that the Demand-Tabulation Algorithm is likely to visit many fewer nodes than the exhaustive algorithm.)

The graph in Figure 10 plots the percentage of demands that are answered *yes* versus the ratio of the running times of the two algorithms for the five dataflow-analysis problems.

Based on the results of our first two experiments, we hypothesize that when the goal is to answer demands at most program points, and it is expected that most demands will be answered *no*, the exhaustive algorithm will be the algorithm of choice. However, when the expected number of demands is small (for example, in an interactive tool, or in a restructuring tool that is likely to demand dataflow information only

Experiment 2: Sequence of Demands vs Exhaustive

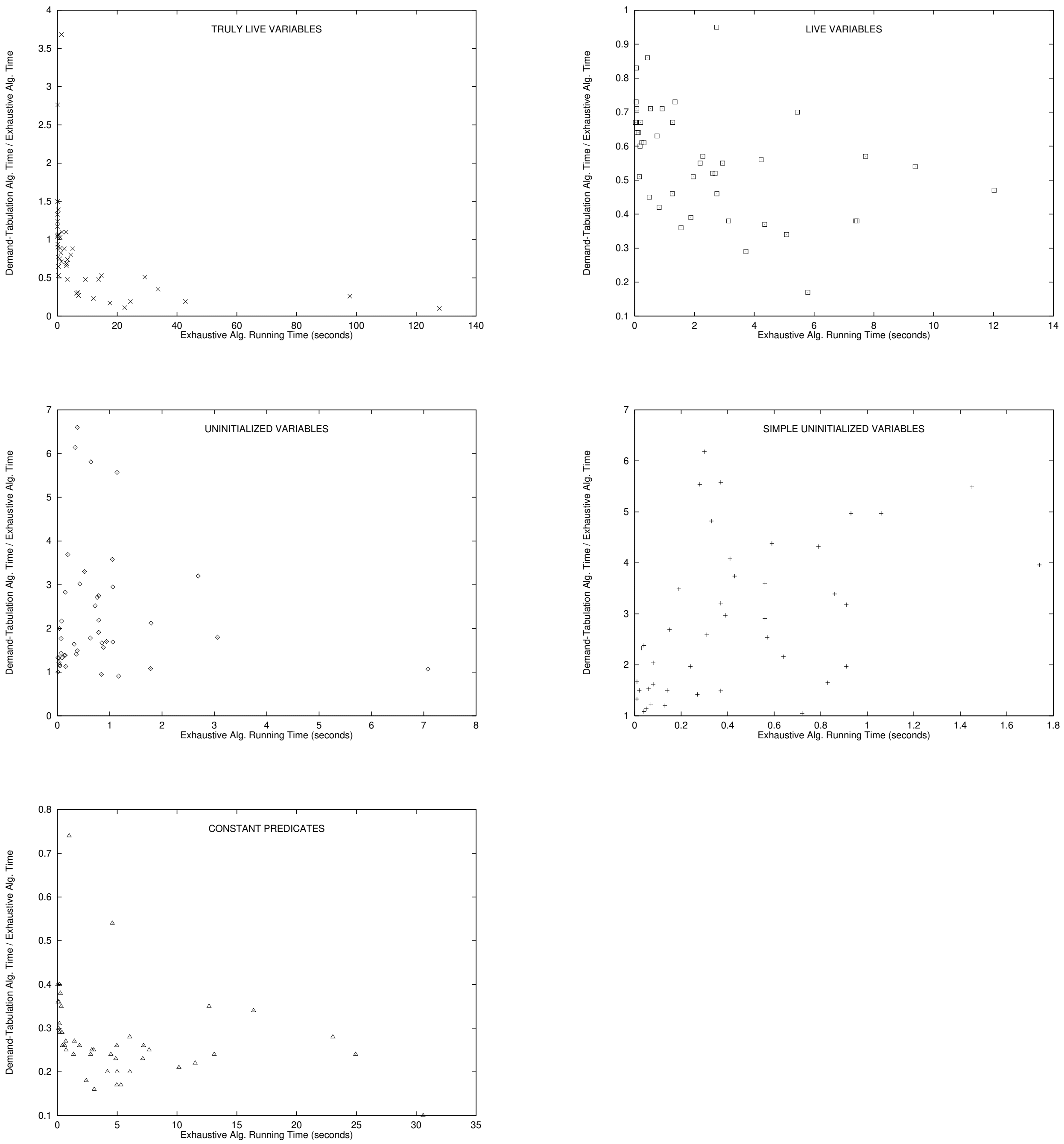


Figure 9. Second comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm. The exhaustive algorithm is used to find the entire meet-over-all-valid-paths solution, and the Demand-Tabulation Algorithm is used to answer all “interesting” demands.

Experiment 2: Time Ratios vs. Percentage of "yes" Answers

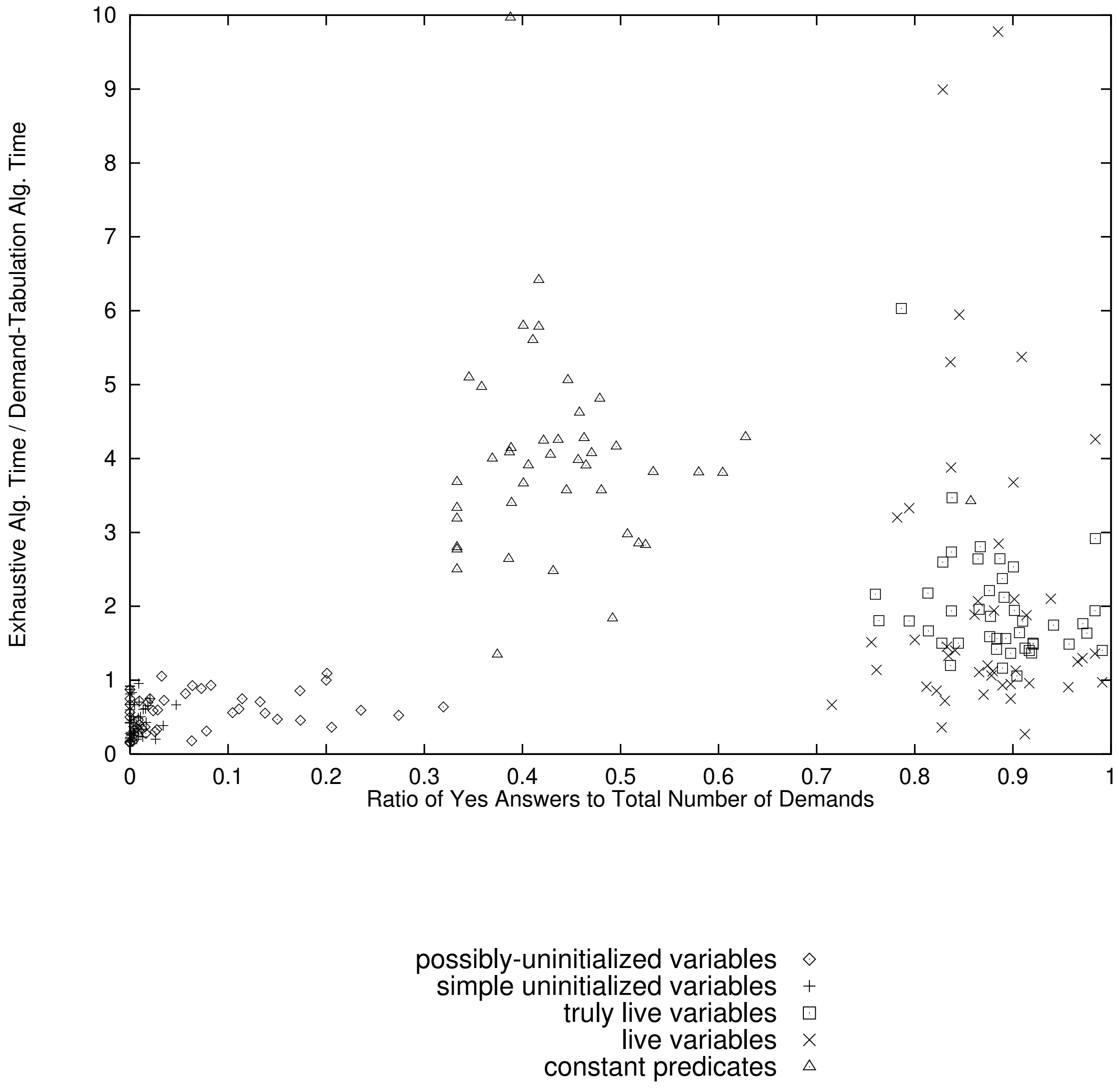


Figure 10. When most demands are answered *yes*, the Demand-Tabulation Algorithm is likely to outperform the exhaustive algorithm; the situation is reversed when most demands are answered *no*.

for a small part of a program before performing a transformation, or for a problem like the *constant-predicates* problem), or it is expected that most demands will be answered *yes*, then the Demand-Tabulation Algorithm will be the algorithm of choice.

Experiment 3: Caching vs Non-caching demand (single demand)

The goal of our third and fourth experiments was to study the tradeoff between the benefit and overhead of caching, first on a single demand and then on a sequence of demands.

For our third experiment we applied the non-caching demand algorithm to the same 20 randomly selected demands used in Experiment 1 (starting the algorithm from scratch for each demand as was done for the Demand-Tabulation Algorithm), and we computed the average running time for a single demand. The results of this experiment are shown in Figure 11. Note that the maximum overhead for caching is 20–25%.

Experiment 4: Caching vs Non-caching demand (sequence of demands)

For our final experiment, we applied the non-caching demand algorithm to the same sequences of all "interesting" demands to which the Demand-Tabulation Algorithm was applied in Experiment 2. The results of this experiment are shown in Figure 12.

For a sequence of demands, the benefits of caching outweigh its overhead in all cases for the *truly-live variables* and the *possibly-uninitialized variables* problems, and in all but one case for the *simple uninitialized-variables* problem (the exception is for the test program *hash*, where the ratio of the running times is 1.3). For the *constant-predicates* problem, caching is a win in only half of the cases, and for the *live-variables* problem, in only about one-third of the cases. However, for both of these problems, it seems that the caching algorithm is still the algorithm of choice; at worst, in one case, the caching algorithm runs 49% slower than the non-caching algorithm, while in six cases, the non-caching algorithm runs at least 50% slower than the caching algorithm (and at worst, is 450% slower).

5. Relation to Previous Work

Until very recently, work on demand-driven dataflow analysis only considered the *intra*procedural case (*cf*. [2]) and work on *inter*procedural dataflow analysis only considered the exhaustive case (*cf*. [25,7,6,17]). Because in intraprocedural dataflow analysis *all* paths in the control-flow graph are assumed to be valid execution paths, the work on demand-driven intraprocedural dataflow analysis does not extend to the interprocedural case, where the notion of *realizable* paths is important.

One approach to obtaining demand algorithms for interprocedural dataflow-analysis problems was described by Reps [22,21]. Reps presented a way in which algorithms that solve demand versions of interprocedural analysis problems can be obtained automatically from their exhaustive counterparts (expressed as logic programs) by making use of the "magic-sets transformation", a general transformation developed in the logic-programming and deductive-database communities for creating efficient demand versions of (bottom-up) logic programs [24,3,5,26]. Reps illustrated this approach by showing how to obtain a demand algorithm for the interprocedural locally separable problems. Subsequent work by Reps, Sagiv, and

Experiment 3: Caching Demand vs Non-Caching Demand (Single Demand)

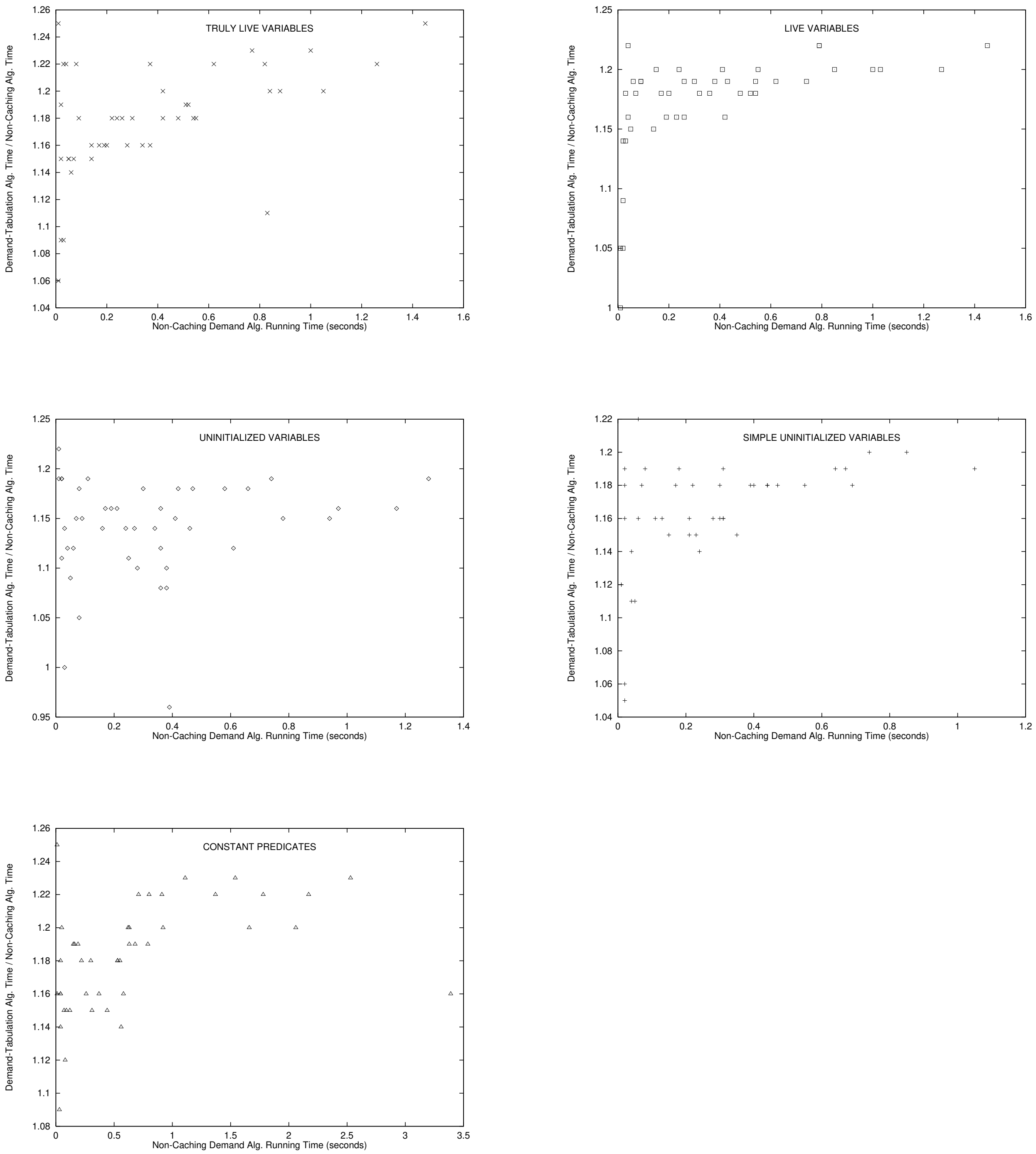


Figure 11. Comparison of the caching and non-caching demand algorithms for a single demand.

Experiment 4: Caching Demand vs Non-Caching Demand (Sequence of Demands)

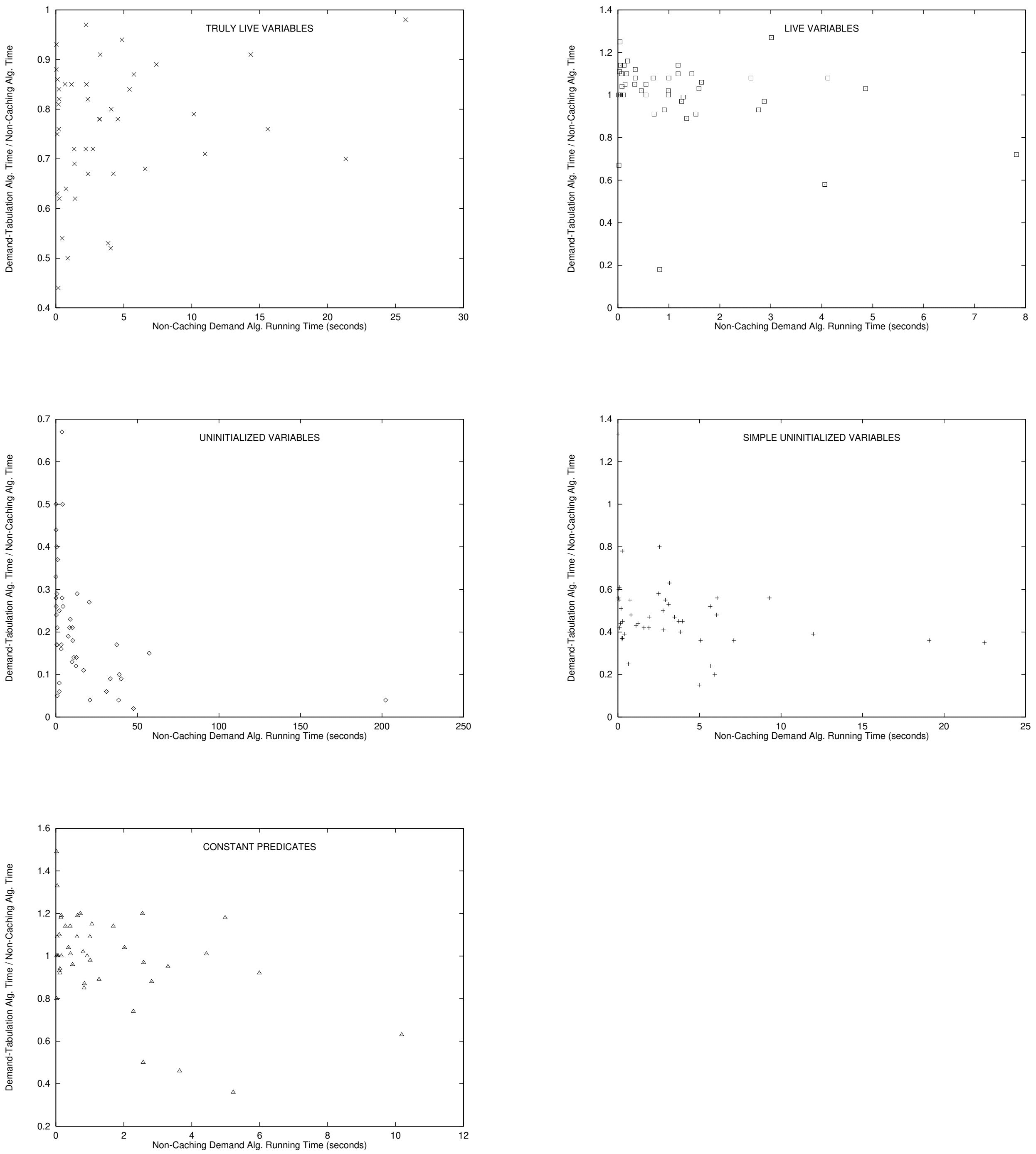


Figure 12. Comparison of the caching and non-caching demand algorithms for a sequence of demands.

Horwitz extended the logic-programming approach to the class of IFDS problems [20,23]. (The latter papers do not make use of logic-programming terminology; however, the exhaustive algorithms described in the papers have straightforward implementations as logic programs. Demand algorithms can then be obtained by applying the magic-sets transformation.)

Several people, leery of the (space, time, and conceptual) overheads involved in using logic databases, questioned whether the logic-programming approach to obtaining demand algorithms for interprocedural dataflow analysis can really produce implementations that are efficient enough to be used in real-world program-analysis tools. Although the jury is still out on this issue (waiting for improved logic-database implementations), it is natural to ask a related question: "Is there a way to adapt the ideas so that they can be used in program-analysis tools written in imperative programming languages?"

In this paper, we answer this question in the affirmative. The two basic ideas used in the magic-sets transformation are *propagation of queries* and *caching of results*, and it is fairly easy to transfer these notions over to demand algorithms written in an imperative programming language (such as C). The Demand-Tabulation Algorithm given in Section 3 can be viewed as an analog of the magic-sets-transformed exhaustive dataflow analysis program: The operations that push nodes onto NodeStack in BackwardDFS and insert edges into EdgeWorkList in BackwardTabulateSLRPs for subsequent processing can be viewed as "query-propagation" operations; the sets ReachableNodes, VisitedNodes, PathEdge, and SummaryEdge, whose values are preserved across calls, can be viewed as caches of previously computed results (and previously computed "intermediate values").

On the other hand, there are a number of benefits obtained when an imperative programming language is used to implement these ideas. The most important benefit is that the algorithm of Section 3 has a simple, low-overhead implementation in an imperative programming language. The implementation is based on array indexing and linked lists, and involves neither term-unification nor term-matching. In addition, an imperative implementation has the opportunity to exploit specific properties of the problem that are not present in all logic programs (and hence would not be exploited by either the magic-sets transformation or the bottom-up "engine" used for evaluating logic-programs). The two forms of "early cut-off" employed by the algorithm given in Section 3 provide two examples of how such properties can be exploited to improve performance:[2]

- The use of a depth-first-search strategy in BackwardDFS allows BackwardDFS to terminate as soon as a node in ReachableNodes is encountered.
- In BackwardTabulateSLRPs, once it is known that there is a path from some node of the form $\langle m, \mathbf{0} \rangle$ to $\langle exit_p, d \rangle$, the path edge $\langle start_p, \mathbf{0} \rangle \rightarrow \langle exit_p, d \rangle$ is inserted into PathEdge. Subsequent tests for

[2] A previous version of the Demand-Tabulation Algorithm that was presented in [15] did not employ either of these strategies for early cut-off.

membership of this edge in PathEdge allow BackwardTabulateSLRPs to avoid processing more path edges with target $\langle exit_p, d \rangle$.

Of course, these early cut-offs can only be taken when the answer to the current demand is *yes*. Thus, these improvements to the algorithm are most significant for problems with a high ratio of *yes* answers.

A different approach to obtaining demand versions of dataflow-analysis algorithms has been investigated by Duesterwald, Gupta, and Soffa, first for intraprocedural problems [10] and subsequently for interprocedural problems [11]. In their approach, for each demand of the form "Is fact d in the solution set at flowgraph node x?", a set of dataflow equations is set up on the flow graph (but as if all edges were reversed). The flow functions on the reverse graph are the (approximate) inverses of the original forward functions. (A special function—derived from the demand—is used for the reversed flow function of vertex x.) These equations are then solved using a demand-driven fixed-point-finding procedure to obtain a value for the entry vertex. The answer to the demand (true or false) is determined from the value so obtained.

Their techniques are somewhat more general than ours because they can handle distributive problems on any finite lattice, while the Demand-Tabulation Algorithm is limited to finite subset lattices. However, our work has the following advantages:

(i) When applied to an arbitrary IFDS problem, the worst-case cost of the Duesterwald-Gupta-Soffa technique is $O(E\,D\,2^D)$. In contrast, the cost of the Demand-Tabulation Algorithm is only $O(E\,D^3)$.

(ii) The Demand-Tabulation Algorithm uses only very simple operations (*e.g.*, set insertions and membership tests that can be implemented with marks). This makes the algorithm very easy to implement, and makes it likely that it will perform better than the Duesterwald-Gupta-Soffa algorithm in practice.

References

1. , "SPEC Component CPU Integer Release 2/1992," (CINT92), Standard Performance Evaluation Corporation (SPEC), Fairfax, VA (1992).
2. Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part II. Demand analysis," *Acta Informatica* **10**(3) pp. 265-272 (October 1978).
3. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic sets and other strange ways to implement logic programs," in *Proceedings of the Fifth ACM Symposium on Principles of Database Systems,* (Cambridge, MA), (March 1986).
4. Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages,* (San Antonio, TX, January 29-31, 1979), (January 1979).
5. Beeri, C. and Ramakrishnan, R., "On the power of magic," pp. 269-293 in *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* (San Diego, CA, March 1987), (March 1987).
6. Callahan, D., Cooper, K.D., Kennedy, K., and Torczon, L., "Interprocedural constant propagation," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction,* (Palo Alto, CA, June 25-27, 1986), *ACM SIGPLAN Notices* **21**(7) pp. 152-161 (July 1986).
7. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).
8. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 57-66 (July 1988).

9. Cooper, K.D. and Kennedy, K., “Fast interprocedural alias analysis,” pp. 49-59 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, January 11-13, 1989), (January 1989).
10. Duesterwald, E., Gupta, R., and Soffa, M.L., “Demand-driven program analysis,” Technical Report TR-93-15, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA (October 1993).
11. Duesterwald, E., Gupta, R., and Soffa, M.L., “Demand-driven computation of interprocedural data flow,” in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, January 23-25, 1995), (January 1995).
12. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
13. Giegerich, R., Moncke, U., and Wilhelm, R., “Invariance of approximative semantics with respect to program transformations.,” pp. 1-10 in *Informatik-Fachberichte 50,* Springer-Verlag, Berlin Heidelberg New York (1981).
14. Horwitz, S. and Teitelbaum, T., “Generating editing environments based on relations and attributes,” *ACM Transactions on Programming Languages and Systems* **8**(4) pp. 577-608 (October 1986).
15. Horwitz, S., Reps, T., and Sagiv, M., “Demand interprocedural dataflow analysis,” in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, (October 1995).
16. Kildall, G., “A unified approach to global program optimization,” pp. 194-206 in *Conference Record of the First ACM Symposium on Principles of Programming Languages,* (Boston, MA, October 1-3, 1973), (October 1973).
17. Knoop, J. and Steffen, B., “The interprocedural coincidence theorem,” pp. 125-140 in *Proceedings of the Fourth International Conference on Compiler Construction*, (Paderborn, FRG, October 5-7, 1992), *Lecture Notes in Computer Science,* Vol. 641, ed. U. Kastens and P. Pfahler,Springer-Verlag, New York, NY (1992).
18. Linton, M.A., “Implementing relational views of programs,” pp. 132-140 in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, April 23-25, 1984), (April 1984).
19. Masinter, L.M., “Global program analysis in an interactive environment,” Tech. Rep. SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, CA (January 1980).
20. Reps, T., Sagiv, M., and Horwitz, S., “Interprocedural dataflow analysis via graph reachability,” Technical Report 94-14, Datalogisk Institut, University of Copenhagen, Copenhagen, Denmark (April 1994).
21. Reps, T., “Demand interprocedural program analysis using logic databases,” in *Applications of Logic Databases*, ed. R. Ramakrishnan,Kluwer Academic Publishers, Boston, MA (1994).
22. Reps, T., “Solving demand versions of interprocedural analysis problems,” pp. 389-403 in *Proceedings of the Fifth International Conference on Compiler Construction*, (Edinburgh, Scotland, April 7-9, 1994), *Lecture Notes in Computer Science,* Vol. 786, ed. P. Fritzson,Springer-Verlag, New York, NY (1994).
23. Reps, T., Sagiv, M., and Horwitz, S., “Precise interprocedural dataflow analysis via graph reachability,” in *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, January 23-25, 1995), (January 1995).
24. Rohmer, R., Lescoeur, R., and Kersit, J.-M., “The Alexander method, a technique for the processing of recursive axioms in deductive databases,” *New Generation Computing* **4**(3) pp. 273-285 (1986).
25. Sharir, M. and Pnueli, A., “Two approaches to interprocedural data flow analysis,” pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).
26. Ullman, J.D., *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies,* Computer Science Press, Rockville, MD (1989).
27. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).