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 new algorithm has three 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.

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 [1,9,20,18,19,10]. 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 [3,7]. This problem can be decomposed into two phases: computing side effects disregarding aliases (the so-called DMod problem), and computing alias information [3,8,7]. 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 invalidate previously computed dataflow information at other points in the program. In some 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 [17,25,16,13]. 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 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. It is desirable, however, for a demand dataflow analysis algorithm to minimize the

[†] 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

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

SIGSOFT '95 Washington, D.C., USA
© 1995 ACM 0-89791-716-2/95/0010...$3.50

amount of such auxiliary information computed. Certainly the worst-case cost of a demand dataflow 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 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 has three 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 [21], which is currently the best exhaustive algorithm for the class of dataflow problems that can be handled precisely by our demand algorithm.

The remainder of the paper is organized as follows: Section 2 provides background material. First, the class of dataflow 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. Preliminary experimental results on C programs are reported in Section 4. Section 5 discusses related work.

2. Background

The algorithm given in Section 3 can be used to solve any interprocedural dataflow problem in which the dataflow facts form a finite set D, and the dataflow functions (which are of type $2^D \to 2^D$) 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 [12], copy constant propagation [11, pp. 660], and possibly-uninitialized variables. The IFDS framework was defined in [21], 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 [23], 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 [15]. These frameworks generalize Kildall's concept of the "meet-over-all-paths" solution of an *intra*procedural dataflow-analysis problem [14] 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 semi-lattice (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 $2^D \to 2^D$) 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

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

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

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

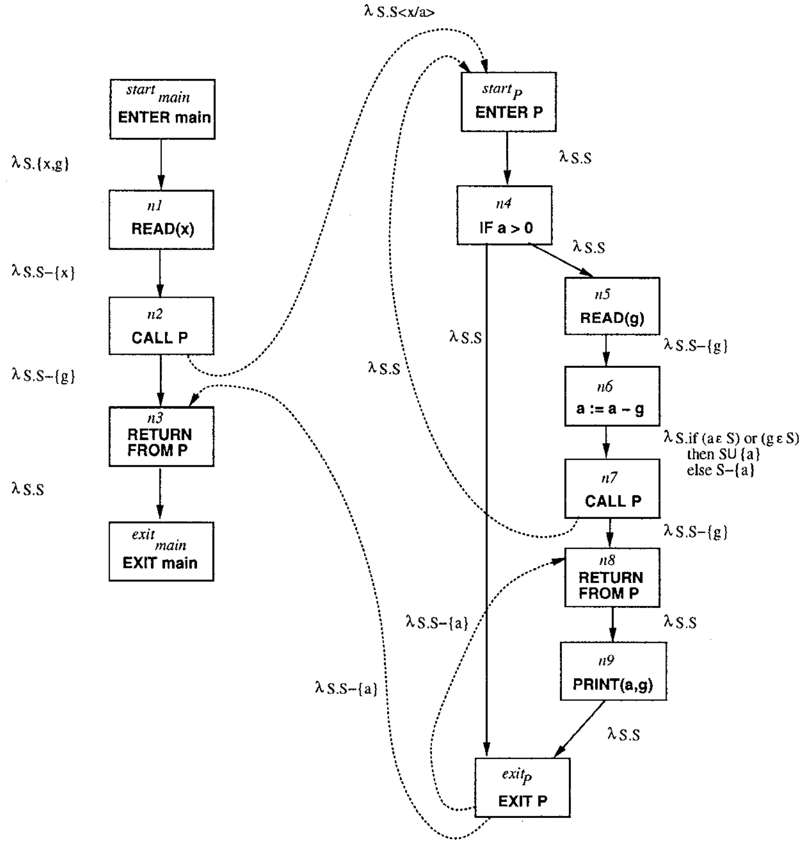
is a non-same-level valid path (because the call edge

```
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.

$n2 \rightarrow start_p$ has no matching return 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 return edge $exit_p \rightarrow n8$ does not correspond to the preceding call 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 $n6$. □

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.

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

The nodes of supergraph G^* are "exploded" into the nodes of $G^\#$ as follows:

- For every node n in G^*, there is a node $\langle n, \mathbf{0} \rangle$ in $G^\#$.
- For every node n in G^*, and every dataflow fact $d \in D$, there is a node $\langle n, d \rangle$ in $G^\#$.

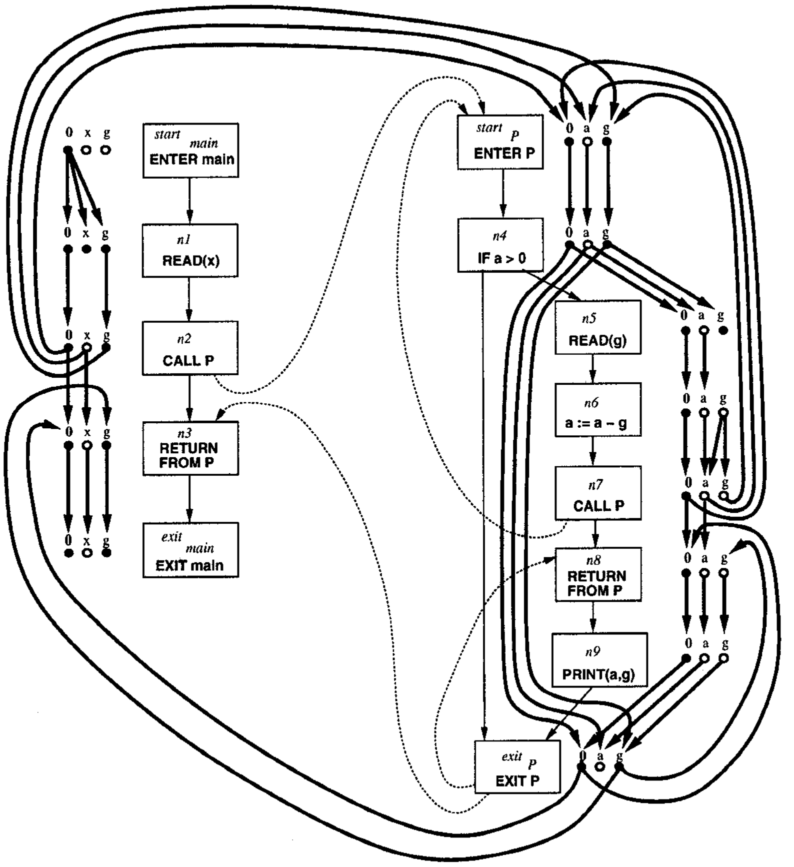


Figure 2. The exploded supergraph that corresponds to the instance of the possibly-uninitialized variables problem shown in Figure 1. Closed circles represent nodes of $G^{\#}$ that are reachable along realizable paths from $\langle start_{main}, \mathbf{0}\rangle$. Open circles represent nodes not reachable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

The edges of $G^{\#}$ represent the dataflow functions on the edges of G^{*} as follows: Given function f associated with edge $m \rightarrow n$ of G^{*}:

- There is an edge in $G^{\#}$ from node $\langle m, \mathbf{0}\rangle$ to node $\langle n, d\rangle$ for every $d \in f(\emptyset)$.
- 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 $d_2 \notin f(\emptyset)$.
- There is an edge in $G^{\#}$ from node $\langle m, \mathbf{0}\rangle$ to node $\langle n, \mathbf{0}\rangle$.

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 [18] 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. 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($\langle \bar{n}, d\rangle$) returns **true** iff there is a realizable path from node $\langle start_{main}, \mathbf{0}\rangle$ to node $\langle \bar{n}, d\rangle$ in $G^{\#}$. Such a path exists iff the meet-over-all-valid-paths solution to the dataflow problem at node $\bar{n}$ of G^{*} includes dataflow fact d.

IsMemberOfSolution maintains a set called ReachableNodes; an exploded-graph node $\langle m, d\rangle$ is in this set

```
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 $\{\langle m, \mathbf{0}\rangle \mid m \in N^*\}$   /* This set is preserved across calls */
  VisitedNodes:                          global node set, initially empty                          /* This set is preserved across calls */
  ReachableNodesRelevantToDemand:        global node set

function IsMemberOfSolution($\langle \bar{n}, \bar{d}\rangle$: exploded supergraph node) returns boolean
declare NodeWorkList, EdgeWorkList: edge set
begin
  ReachableNodesRelevantToDemand := $\emptyset$; NodeWorkList := $\emptyset$
  Visit($\langle \bar{n}, \bar{d}\rangle$, NodeWorkList)
  while NodeWorkList $\neq \emptyset$ do
    Select and remove a node $\langle n, d\rangle$ from NodeWorkList
    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 := $\emptyset$
          for each $d'$ such that $\langle exit_p, d'\rangle \to \langle n, d\rangle \in E^\#$ do Propagate($\langle exit_p, d'\rangle \to \langle exit_p, d'\rangle$, EdgeWorkList) od
          BackwardTabulateSLRPs(EdgeWorkList)
          for each $d'$ such that $\langle c, d'\rangle \to \langle n, d\rangle \in (E^\# \cup$SummaryEdge) do Visit($\langle c, d'\rangle$, NodeWorkList) od
        end let
      end case
      case $n$ is the start node of procedure $p$ :
        for each $c \in$ Callers($p$) do
          for each $d'$ such that $\langle c, d'\rangle \to \langle n, d\rangle \in E^\#$ do Visit($\langle c, d'\rangle$, NodeWorkList) od
        od
      end case
      default :
        for each $\langle m, d'\rangle$ such that $\langle m, d'\rangle \to \langle n, d\rangle \in E^\#$ do Visit($\langle m, d'\rangle$, NodeWorkList) od
      end case
    end switch
  od
  UpdateReachableNodes()
  return($\langle \bar{n}, \bar{d}\rangle \in$ ReachableNodes)
end

procedure Visit ($n^\#$: exploded supergraph node, NodeWorkList: node set)
begin
  if $n^\# \in$ ReachableNodes then Insert $n^\#$ into ReachableNodesRelevantToDemand
  else if $n^\# \notin$ VisitedNodes then Insert $n^\#$ into VisitedNodes; Insert $n^\#$ into NodeWorkList fi
  fi
end

procedure UpdateReachableNodes()
begin
  while ReachableNodesRelevantToDemand $\neq \emptyset$ do
    Select and remove an exploded node $\langle n, d\rangle$ from ReachableNodesRelevantToDemand
    for each $\langle m, d'\rangle$ such that ($\langle n, d\rangle \to \langle m, d'\rangle \in (E^\# \cup$SummaryEdge)) and $\langle n, d\rangle \to \langle m, d'\rangle$ is not an exit-to-return-site edge, and
                    ($\langle m, d'\rangle \in$ VisitedNodes) and ($\langle m, d'\rangle \notin$ ReachableNodes) do
      Insert $\langle m, d'\rangle$ into ReachableNodes
      Insert $\langle m, d'\rangle$ into ReachableNodesRelevantToDemand
    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.

when it has been determined that there is a realizable path from $\langle start_{main}, \mathbf{0}\rangle$ to $\langle m, d\rangle$. Before the first call on IsMemberOfSolution is performed, ReachableNodes is initialized to $\{ \langle m, \mathbf{0}\rangle \}$ for all supergraph nodes m.

IsMemberOfSolution also maintains a set called VisitedNodes (initially empty) that includes all exploded-graph nodes visited during some invocation of IsMemberOfSolution. At the end of an invocation of IsMemberOfSolution, the nodes that were added to VisitedNodes during that invocation and that are reachable from $\langle start_{main}, \mathbf{0}\rangle$ by a realizable path are placed in ReachableNodes by procedure UpdateReachableNodes. Finally, if the "demand node" $\langle \bar{n}, \bar{d}\rangle$ is in ReachableNodes, IsMemberOfSolution returns *true*; otherwise it returns *false*.

IsMemberOfSolution works by starting at demand node $\langle \bar{n}, \bar{d}\rangle$ and traversing $G^\#$ backwards, one edge at a time, encountering nodes from which there are realizable paths to $\langle \bar{n}, \bar{d}\rangle$. IsMemberOfSolution avoids repeating work done on a previous call by making use of the ReachableNodes and VisitedNodes sets. If an encountered node is in

ReachableNodes, it is inserted into the set ReachableNodesRelevantToDemand (line [26]), and because its reachability status is already known, its predecessors are not explored. Otherwise, if the encountered node is not in VisitedNodes, it is inserted into VisitedNodes and is placed onto the NodeWorkList so that its predecessors will be explored.

When the traversal of $G^{\#}$ is finished, procedure UpdateReachableNodes is used to augment the ReachableNodes set by adding all nodes $\langle m, d \rangle$ that were encountered for the first time during the current invocation of IsMemberOfSolution, and that are reachable from $\langle start_{main}, \mathbf{0} \rangle$ by a realizable path. UpdateReachableNodes starts from the nodes in ReachableNodesRelevantToDemand and works forward in $G^{\#}$, adding nodes encountered during the current invocation of IsMemberOfSolution to ReachableNodes.

Note that IsMemberOfSolution will ultimately return *true* iff it encounters a member of ReachableNodes. The reason it does not return immediately when such a node is encountered is to ensure that all newly encountered nodes that are reachable from $\langle start_{main}, \mathbf{0} \rangle$ get inserted into ReachableNodes. This is necessary to guarantee that the Demand-Tabulation Algorithm has the same-worst-case-cost property with respect to the algorithm of [21] (see Section 3.2). Note that the traversal performed by UpdateReachableNodes only involves edges already visited by IsMemberOfSolution during the same invocation, so this does not increase the asymptotic cost of the algorithm when it is used for just a single demand. The practical consequences of this extra work are explored in Section 4.

The interesting aspect of the backwards traversal performed by IsMemberOfSolution is the way it ensures that only realizable paths are followed. This is accomplished by the call to BackwardTabulateSLRPs at line [10], which occurs when the node $\langle n, d \rangle$ removed from NodeWorkList 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 represents a same-level realizable path from a node of $G^{\#}$ that corresponds to a call node, to a node of $G^{\#}$ that corresponds to the matching return-site node. Summary edges are recorded in the (global) set named SummaryEdge. After calling BackwardTabulateSLRPs, IsMemberOfSolution can continue its backward traversal across the newly discovered summary edges (line [11]).

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

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):

1. Node $\langle n9, g \rangle$ is inserted into VisitedNodes and into NodeWorkList by the call to Visit on line [2].
2. Node $\langle n9, g \rangle$ is removed from NodeWorkList (line [4]); the default case (line [19]) is taken, and node $\langle n8, g \rangle$ is inserted into VisitedNodes and NodeWorkList by procedure Visit.
3. Node $\langle n8, g \rangle$ is removed from NodeWorkList; the case on line [6] 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).
4. Node $\langle n7, g \rangle$ is inserted into VisitedNodes and NodeWorkList by procedure Visit (called at line [11]).
5. Node $\langle n7, g \rangle$ is removed from NodeWorkList; the default case is taken, and node $\langle n6, g \rangle$ is inserted into VisitedNodes and NodeWorkList.
6. Node $\langle n6, g \rangle$ is removed from NodeWorkList; the default case is taken, but there are no edges that satisfy the for-loop condition (line [20]).
7. NodeWorkList is now empty, so UpdateReachableNodes is called. There are no nodes in the set ReachableNodesRelevantToDemand, so no new nodes are inserted into ReachableNodes.
8. Finally, the return statement in IsMemberOfSolution is executed. Node $\langle \bar{n}, d \rangle$ is not in ReachableNodes, so the function 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 $D + 1$

```
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 $\neq \varnothing$ do
    Select and remove an edge $\langle n, d_2\rangle \rightarrow \langle exit_p, d_1\rangle$ from EdgeWorkList
    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 $d_3$ such that $\langle exit_q, d_3\rangle \rightarrow \langle n, d_2\rangle \in E^\#$ do Propagate($\langle exit_q, d_3\rangle \rightarrow \langle exit_q, d_3\rangle$, EdgeWorkList) od
          for each $d_3$ such that $\langle c, d_3\rangle \rightarrow \langle n, d_2\rangle \in (E^\# \cup$SummaryEdge$)$ do Propagate($\langle c, d_3\rangle \rightarrow \langle exit_p, d_1\rangle$, EdgeWorkList) od
        end let
      end case
      case $n$ the start node of procedure $p$ :
        for each $c \in$ Callers($p$) do
          let $q$ be $c$'s procedure, and $r$ be the return-site node that corresponds to $c$
            for each $d_4, d_5$ such that $\langle c, d_5\rangle \rightarrow \langle n, d_2\rangle \in E^\#$ and $\langle exit_p, d_1\rangle \rightarrow \langle r, d_4\rangle \in E^\#$ do
              if $\langle c, d_5\rangle \rightarrow \langle r, d_4\rangle \notin$ SummaryEdge then
                Insert $\langle c, d_5\rangle \rightarrow \langle r, d_4\rangle$ into SummaryEdge
                for each $d_3$ such that $\langle r, d_4\rangle \rightarrow \langle exit_q, d_3\rangle \in$ PathEdge do Propagate($\langle c, d_5\rangle \rightarrow \langle exit_q, d_3\rangle$, EdgeWorkList) od
              fi
            od
          end let
        od
      end case
      default :
        for each $m, d_3$ such that $\langle m, d_3\rangle \rightarrow \langle n, d_2\rangle \in E^\#$ do Propagate($\langle m, d_3\rangle \rightarrow \langle exit_p, d_1\rangle$, EdgeWorkList) od
      end case
    end switch
  od
end
procedure Propagate($e$: edge, EdgeWorkList: edge set)
begin
  if $e \notin$ PathEdge then Insert $e$ into PathEdge; Insert $e$ into EdgeWorkList fi
end
```

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

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. □

The table in Figure 5 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 [18].

The most efficient exhaustive algorithm known for the class of IFDS problems is the one given in [21]. 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 7).

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 [21]. 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 possible nodes of $G^\#$, IsMemberOfSolution proper will traverse a given edge in $G^\#$ at most once during the processing of the request sequence. BackwardTabulateSLRPs will traverse a given edge of $G^\#$ in procedure p at most D times (once for each node of the form $\langle exit_p, d\rangle$). (The information accumulated in sets PathEdge and SummaryEdge prevents procedure BackwardTabulateSLRPs from performing additional work, and the information accumulated in ReachableNodes and VisitedNodes prevents IsMemberOfSolution proper from performing additional work.) This is the same amount of work that could be performed in the worst case by the exhaustive algorithm given in [21]. Thus, the Demand-Tabulation

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/function representation	$O(ED^2)$	$O(ED^3)$
h-sparse	At most $O(hD)$ edges/function representation	$O(hED)$	$O(Call\,D^3 + hED^2)$
(Locally) separable	Component-wise dependences	$O(E)$	$O(ED)$

Figure 5. Asymptotic running time of the Demand-Tabulation Algorithm (for answering a single demand) for six different classes of dataflow-analysis problems.

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 8).

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 [21], and two further experiments to study the trade-off between the benefit and overhead of the caching performed by the Demand-Tabulation Algorithm.

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 encounters a node of the form $\langle m, \mathbf{0}\rangle$, does not maintain the set ReachableNodes, and reinitializes the set VisitedNodes to $\emptyset$ after each invocation of IsMemberOfSolution, but *does* preserve the sets PathEdge and SummaryEdge across calls to IsMemberOfSolution), and (3) the exhaustive algorithm reported in [21].

All three algorithms were implemented in C and used with a front end that analyzes a C program and generates the corresponding exploded supergraph for two dataflow problems: the possibly-uninitialized variables problem that we have used as our running example, and the *truly-live variables* problem [12]. A 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 predicate or in a call to a library routine, or on which x is used to assign a value to a truly-live variable. (The more standard, locally-separable version of the live-variables problem relaxes the third condition so that x is considered to be live if it is used to assign a value to *any* variable.) It is useful to identify assignments to non-truly-live variables; programming tools might flag them as indicating possible logical errors, and optimizing compilers could remove them.

Procedure calls via pointers to procedures, and aliasing due to pointers were handled by our C front end as follows: For both dataflow 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 the truly-live variables problem, pointer-induced aliasing was handled conservatively: every memory read and every memory write via a pointer was considered to be a possible read/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 possibly-uninitialized variables problem, this conservative approach was inappropriate; since 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. Therefore, for this problem, memory writes via a pointer were handled as described above for the truly-live variables problem, but memory reads were considered to read only the value of the pointer itself.

Tests were carried out on a Sun SPARCstation 20 Model 71 with 64 MB of RAM. The study used thirteen standard C programs; the table in Figure 6 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), data for the possibly-uninitialized variables problem (the number of uses of scalar variables, and the number of those uses that were found to be possibly uninitialized), and data for the truly-live variables problem (the number of assignments to scalar variables, and the number of those assignments that were found to be truly live).

4.2. Experiments

Experiment 1: Single demand vs. Exhaustive

Our first two experiments compare 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 test program: (1) the time used by the exhaustive algorithm to find all possibly uninitialized variables at all supergraph nodes; (2) the time used by the exhaustive algorithm to find all truly-live variables at all supergraph nodes; (3) the average time used by the

Example	Lines of source code	CFG statistics			possibly-uninit statistics		truly-live statistics	
		P	*Call*	*N*	*# uses*	*# uninit uses*	*# assignments*	*# truly live assignments*
diff.diffh	303	13	48	616	219	4	320	277
compress	657	14	27	1472	428	0	772	679
ratfor	1531	51	265	2654	847	232	1245	941
struct.beauty	1701	32	211	2625	708	18	1273	1060
diff.diff	1761	40	125	3496	1178	75	1839	1538
gnugo	1963	27	87	2967	1147	120	1409	1134
twig	2555	75	221	4161	1278	16	2049	1813
cdecl	2577	31	202	2967	730	2	1519	1345
lex	2645	61	328	6146	2577	201	3192	2747
patch	2746	53	263	4850	1569	216	2592	2433
unzip	3261	39	125	3192	1231	185	1711	1551
tbl	3462	82	313	5793	2147	135	2964	2562
flex-2.4.7	9488	142	785	10893	3879	548	5119	4813

Figure 6. Test program information. (The number of uninitialized uses and the number of truly live assignments both reflect the number of times a demand leads to a "yes" answer. However, in the case of the truly-live variables problem, it is the instances where the demand leads to a "no" answer—a non-live variable is being assigned to—that are of interest to a programmer or compiler.)

Demand-Tabulation Algorithm to answer a single demand "might *x* be uninitialized here?" for 100 randomly selected uses of a scalar variable *x*; (4) the average time used by the Demand-Tabulation Algorithm to answer a single demand "is *x* truly-live here?" for 100 randomly selected assignments to a scalar variable *x*.

This data is summarized in Figure 7. For each test program, for each of the two dataflow problems, the graph bars indicate the (average) running time of the Demand-Tabulation Algorithm for a single demand, normalized to the time for the exhaustive algorithm; the actual times (user time + system time in seconds) are given below each bar.

Clearly, the Demand-Tabulation Algorithm can be expected to be much faster than the exhaustive algorithm when the former is used to respond to 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 dataflow information is desired at every program point *p*. For this study we recorded: (1) the time used by the Demand-Tabulation Algorithm to answer the sequence of demands "might *x* be uninitialized here?" for every use of a scalar variable *x*; and (2) the time used by the Demand-Tabulation Algorithm to answer the sequence of demands "is *x* truly-live here?" for every assignment to a scalar variable *x*. This data is summarized in Figure 8.

For the truly-live variables problem, the Demand-Tabulation Algorithm outperforms the exhaustive algorithm in 9 out of 13 cases (and at worst, in one case, is about 55% slower); however, this is true for only 4 out of 13 cases for the possibly-uninitialized variables problem (and the Demand-Tabulation Algorithm is slightly more than 4 times slower than the exhaustive algorithm in one

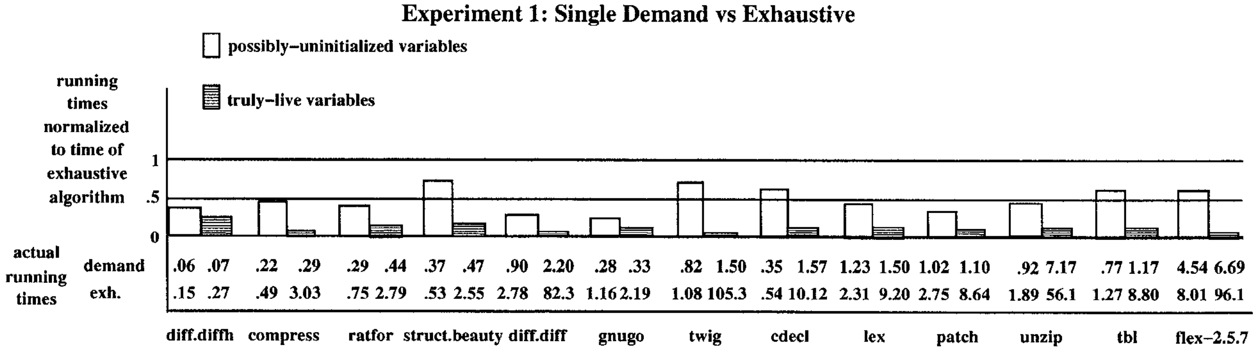
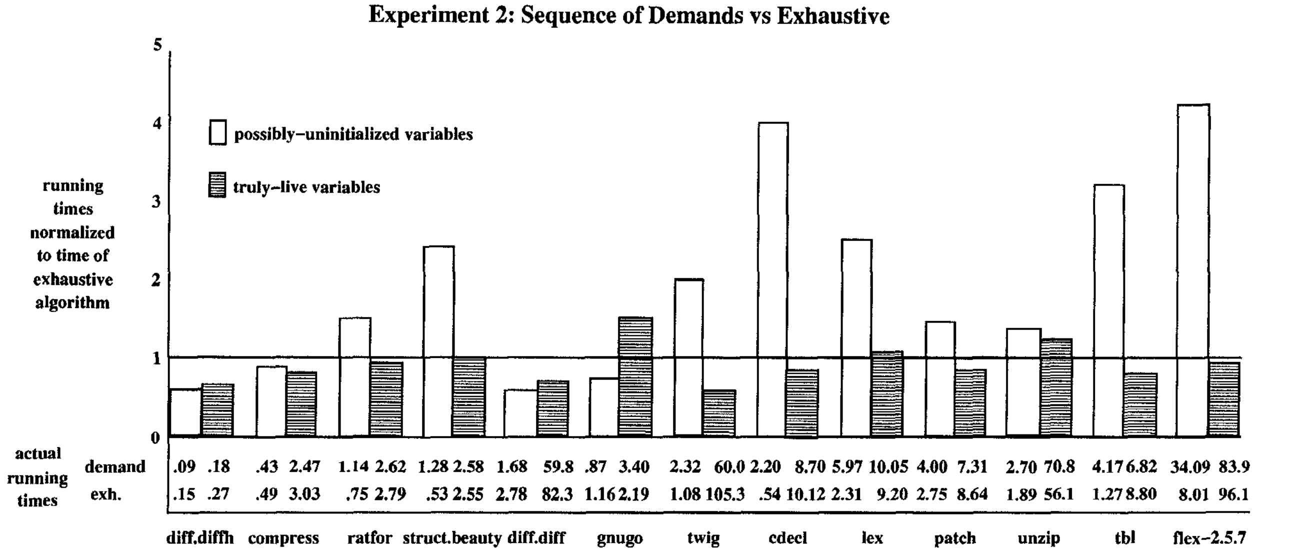


Figure 7. First comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm. Bars show the running times of the Demand-Tabulation Algorithm, normalized to the times of the exhaustive algorithm. The times for the Demand-Tabulation Algorithm are the average times required to answer a single demand (using 100 randomly selected demands).



actual running times		diff.diffh	compress	ratfor	struct.beauty	diff.diff	gnugo	twig	cdecl	lex	patch	unzip	tbl	flex–2.5.7
	demand	.09 .18	.43 2.47	1.14 2.62	1.28 2.58	1.68 59.8	.87 3.40	2.32 60.0	2.20 8.70	5.97 10.05	4.00 7.31	2.70 70.8	4.17 6.82	34.09 83.9
	exh.	.15 .27	.49 3.03	.75 2.79	.53 2.55	2.78 82.3	1.16 2.19	1.08 105.3	.54 10.12	2.31 9.20	2.75 8.64	1.89 56.1	1.27 8.80	8.01 96.1

Figure 8. Second comparison of the Demand-Tabulation Algorithm and the exhaustive algorithm. Bars show the running times of the Demand-Tabulation Algorithm, normalized to the times of the exhaustive algorithm. The times for the Demand-Tabulation algorithm are the total times required for all demands.

case). We believe that this is because in the case of possibly-uninitialized variables, most of the demands ("might *x* be uninitialized at this use?") lead to a *no* answer, while in the case of truly-live variables, most of the demands ("is *x* truly-live at this assignment?") lead to a *yes* 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.

We hypothesize that when the goal is to answer demands at all 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 for a small part of a program before performing a transformation), 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 100 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 9. The graph bars indicate the running time of the non-caching algorithm normalized to the time for the (caching) Demand-Tabulation Algorithm.

It appears that for the uninitialized-variables problem, in the case of a single demand, caching introduces only a modest amount of overhead. However, in all cases the overheads for the truly-live variables problem are more significant.

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

For our final experiment, we applied the non-caching demand algorithm to the same two sequences of demands (for possibly-uninitialized variables and for truly-live variables) to which the Demand-Tabulation Algorithm was applied in Experiment 2. The results of this experiment are shown in Figure 10.

For possibly-uninitialized variables, in the case of a sequence of demands, the benefits of caching far outweigh its overhead. However, for truly-live variables, the non-caching algorithm seems to be preferable to the caching one. The reasons for this phenomenon deserve further study; it would be desirable to understand which characteristics of a dataflow problem and of a problem instance will predict whether a caching or a non-caching algorithm is preferable.

5. Relation to Previous Work

Until very recently, work on demand-driven dataflow analysis only considered the *intra*procedural case (*cf.* [1]) and work on *inter*procedural dataflow analysis only considered the exhaustive case (*cf.* [23,6,5,15]). 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.

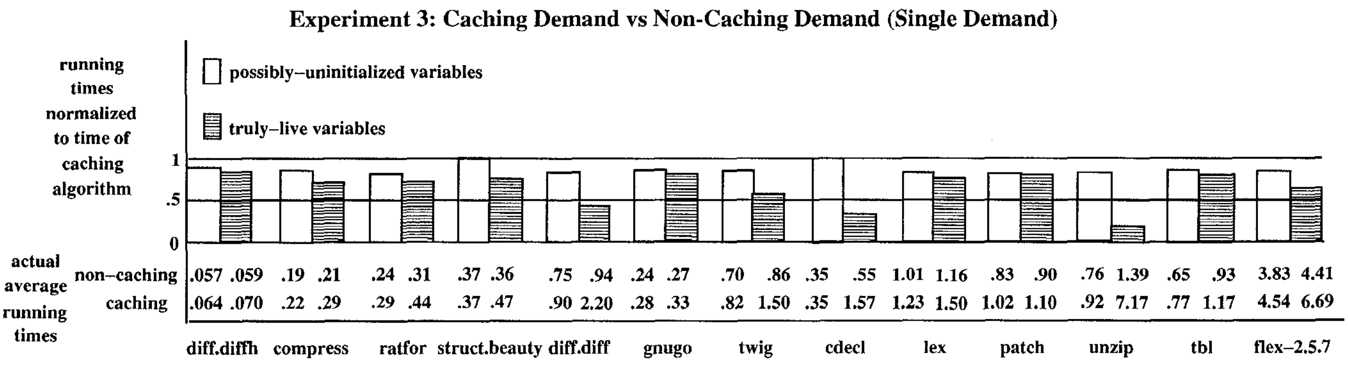


Figure 9. Comparison of the caching and non-caching demand algorithms for a single demand. Bars show the average running times of the non-caching algorithm, normalized to the average times of the (caching) Demand-Tabulation Algorithm.

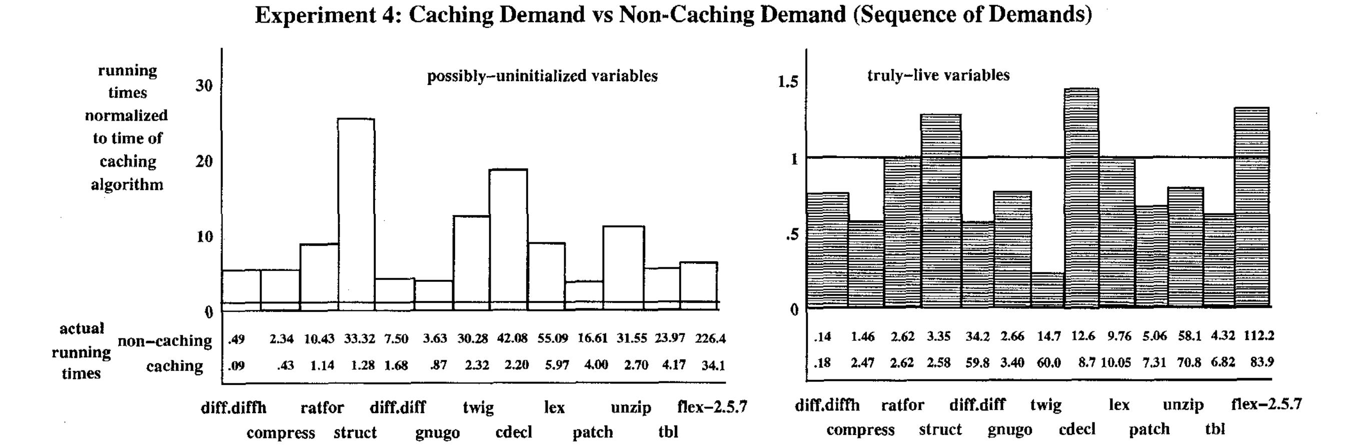


Figure 10. Comparison of the caching and non-caching demand algorithms for a sequence of demands. Bars show the running times of the non-caching algorithm, normalized to the times of the (caching) Demand-Tabulation Algorithm.

One approach to obtaining demand algorithms for interprocedural dataflow-analysis problems was described by Reps [20,19]. 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 logic programs [22,2,4,24]. Reps illustrated this approach by showing how to obtain a demand algorithm for the interprocedural locally separable problems. Subsequent work by Reps, Sagiv, and Horwitz extended the logic-programming approach to the class of IFDS problems [18,21]. (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 demand algorithm given in Section 3 can be viewed as an analog of the magic-sets-transformed exhaustive dataflow analysis program. However, the algorithm of Section 3 has a simple, low-overhead implementation in an imperative programming language (such as C). The implementation is based on array indexing and linked lists, and involves neither term-unification nor term-matching.

A different approach to obtaining demand versions of dataflow-analysis algorithms has been investigated by Duesterwald, Gupta, and Soffa, first for intraprocedural problems [9] and subsequently for interprocedural problems [10]. 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. 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).
2. 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,* (1986).
3. 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).
4. 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).
5. 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).
6. 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).
7. 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).
8. 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).
9. 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).
10. 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).
11. Fischer, C.N. and LeBlanc, R.J., *Crafting a Compiler,* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1988).
12. 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).
13. 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).
14. 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).
15. 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).
16. 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).
17. 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).
18. 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).
19. Reps, T., "Demand interprocedural program analysis using logic databases," in *Applications of Logic Databases,* ed. R. Ramakrishnan,Kluwer Academic Publishers, Boston, MA (1994).
20. 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).
21. 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).
22. 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).
23. 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).
24. Ullman, J.D., *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies,* Computer Science Press, Rockville, MD (1989).
25. Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).