An Efficient Algorithm for Incremental Data-Flow Analysis[1]

F. Kenneth Zadeck

Technical Report No. CS-89-11

February 1988

Abstract

This paper presents a new algorithm for data-flow analysis that works for data-flow problems in a new class, *partitionable* data-flow problems. While this class is a subset of the well-established *fast* class introduced by Graham and Wegman, most of the global data-flow problems that are commonly solved are shown to be in this class. This algorithm has two advantages over extant methods:

1. The algorithm is asymptotically faster than any extant algorithm when applied to the partitionable problems.
2. The algorithm can be easily transformed into an efficient incremental algorithm.

[1]A preliminary version of this paper, "Incremental data flow analysis in a structured program editor" appeared in Proc. SIGPLAN'84 Symp. on Compiler Construction. This research was supported under NSF grant MCS81-04006.

An Efficient Algorithm for Incremental Data-Flow Analysis*

F. Kenneth Zadeck
Brown University

February 25, 1989

1 Introduction

For many years, generally accepted methods for solving global data-flow analysis problems (GDFAPs) have relied on the use of *bit vectors*. In these methods, the program is represented as a graph, the *control flow graph* in which the nodes represent the locations in the program and the edges represent the transfer of control between those locations. The information at each node is represented as a vector of facts. The semantics of the program are represented as functions associated with each edge that model how the state of the progam changes when control passes along that edge. The goal of GDFAP algorithms is to determine the maximal fixed point solution (the solution with the largest number of provably true facts) to this set of functions.

Extant algorithms determine this solution either through iteration [Kil73, HU75, KU76] or through analysis and transformation of the control flow graph [All70, GW76, Tar81]. This paper investigates a different paradigm for solving GDFAPs. In this strategy, the data-flow solution for each position in the vector is computed independently. While not all GDFAPs can be formulated so as to be solved by this technique, this algorithm has two advantages over extant techniques for any problems that can be so formulated:

1. The asymptotic complexity of the algorithm is better than any other technique.
2. There is a complete and efficient incremental algorithm based on this technique.

Section 2 contains basic definitions and Section 3 presents the offline algorithm. Section 4 contains a precise definition of the set of problems that can be solved by this technique. The formulations for several common GDFAPs are given as well as a list of representative problems that cannot be solved by this technique. Section 5 explains some of the uses of incremental analysis in compilers and programming environments and contains the incremental algorithm. Section 6 contains comparisons with other incremental data-flow algorithms and Section 7 gives some conclusions and acknowledgments.

2 Preliminaries

This section contains basic definitions for data-flow analysis. Those readers familiar with this area should feel free to skip to Section 3.

Basic Block A sequence of statements in which: if any statement in the sequence is executed, all statements in the sequence are executed. To simplify the exposition, basic blocks are restricted here to contain a single statement.

Def-Use Chains A set of tuples that connect the location of every *definition site* to every reachable *use site* for each variable in the program.

Exhaustive Algorithm An algorithm that must recalculate from scratch each time the input changes.

*A preliminary version of this paper, "Incremental data flow analysis in a structured program editor" appeared in Proc. SIGPLAN'84 Symp. on Compiler Construction. This research was supported under NSF grant MCS81-04006.

Incremental Algorithm An algorithm that can update the affected part of the solution when only a small amount of the input changes.

The graph representing the control flow of a computer program is called a *program flow graph.* A directed graph $\mathtt{G} = (\mathtt{N}, \mathtt{E}, \mathtt{n_0})$ represents the program, where

$\mathtt{N}$ is the list of nodes. Each node in this graph typically corresponds to a single basic block in the program.

$\mathtt{E}$ is the list of edges. The edges represent the block to block transfers.

$\mathtt{n_0}$ is the initial node in the graph. This node represents the entry point for the routine.

`Succ(n)` is the set of arcs, leaving each node n, that travel in the direction of control flow.

`Pred(n)` is the set of edges obtained if each edge in the *Succ* set is reversed. Each edge in *Pred* runs in the direction contrary to control flow.

2.1 Introduction to Lattice Theory

The data-flow problem can be formalized by casting it in the structure of lattice theory.

A *semilattice* L is a set L with a binary *meet* operation $\cap$ such that for all $a, b, c \in L$:

$a \cap a = a$	idempotent
$a \cap b = b \cap a$	commutative
$a \cap (b \cap c) = (a \cap b) \cap c$	associative
$a \geq b \rightarrow a \cap b = b$	
$a > b \rightarrow a \geq b$ and $a \neq b$	
$a \leq b \rightarrow a \cap b = a$	
$a < b \rightarrow a \leq b$ and $a \neq b$	

Lattices may have:

$\bot \cap a = \bot$	bottom element
$\top \cap a = a$	top element

It is assumed here that all semilattices have a *bottom element.*

Given a semilattice, L, a sequence $a_1 \ldots a_j$ of elements of L is said to be a *chain* if for $1 \leq i \leq j$ we have $a_i > a_{i+1}$.

L is said to be *bounded* if for each $a \in L$ there is a constant b_a such that any chain beginning with a has length at most b_a. If L is bounded, then we can take the meet over countably infinite sets if we define:

$$\bigcap_{a \in S} a \text{ where } S = \{a_1, a_2 \ldots\} \leftrightarrow \lim_{n \to \infty} \bigcap_{1 < i < n} a_i$$

The fact that L is bounded assures us that the limit does exist.

2.2 Data-Flow Problems

Data-flow solutions have been formulated to solve a variety of problems in compiler optimization and programming environments. Data-flow techniques can be used to gather information about a computer program. Data-flow analysis is a useful technique to find an approximate solution to these problems, which are undecidable in their most general form. The possible solutions of any data-flow problem can be judged using several measures:

1. The precision of the information obtained.
2. The time and space required for the solution.
3. Applicable algorithms.
4. Uniqueness of the solution.

This paper presents an algorithm to solve directly *partitionable* data-flow problems. For any problem in this class, the algorithm runs in time proportional to the size of the flow graph. These algorithms do not require flow graph reducibility. A similar technique, applicable to a smaller class of data-flow problems, has been presented by Kou [Kou77] and Babich and Jazayeri [BJ78].

3 The Offline PVT Algorithm

The offline or batch version of the *partitioned variable technique* (PVT) is presented here. In Section 3.1 an informal version of the algorithm is used to solve the *reaching definitions problem*, a common data-flow problem, with the hope of conveying the intuition underlying the algorithm . This is followed in Section 3.2 by some generalizations that allow this technique to solve a large class of data-flow problems. In Section 3.3 pseudo-code is given for the complete offline algorithm.

3.1 The Offline Algorithm for Reaching Definitions

The data-flow problem solved here is the *reaching definitions* problem. The goal of this problem is to determine, for each point `P` in the program, the set of definitions that can *reach* `P`. Definitions are typically assignment statements. A definition `D` for variable `V` can reach `P` if there exists a path through the program from `D` to `P` that does not contain another definition of `V`.

Figure 1 contains an outline of the algorithm. Each of the steps is further explained in the section number in noted on the right.

```
for each partition p do                                    3.1.1
   split the nodes containing assignments to V             3.1.2
   collapse the strongly connected regions                 3.1.3
   propagate the information                               3.1.4
   map the information back                                3.1.5
end
```

Figure 1. The Big Picture

3.1.1 Determining the Partitions

A bit vector at each node in the program is used to represent information in extisting data-flow algorithms. Each position of the bit vector corresponds to one definition. A set bit indicates that the corresponding definition for that variable reaches that point in the program. Current algorithms find their solution for all definitions at one time. The algorithm presented here partitions the problem by variable and finds the solution for all of the definitions to a single variable at one time. For this reason, the algorithm is called the partitioned variable technique (or PVT).

To discover the solution for all variables, the algorithm must iterate over all variables.[1] For the reaching definitions problem, bit vectors are used but they are much shorter that those of other algorithms. Like those other algorithms, one bit is used to represent the solution for each assignment. But only the bits corresponding to the assignments for a single variable must be maintained at one time. The bits corresponding to a single variable are called a *partition*. As Figure 1 shows, this algorithm is executed once for each partition p.

3.1.2 Splitting the Nodes in the Control Flow Graph

A second way in which the PVT algorithm differs from extant techniques is that the flow graph is manipulated on the basis of the location of variable definitions contained within each control flow graph node.

As above, for a definition of `V` to reach a point in the program `P` it must reach it along a path that does not contain another definition to `V`. Effectively, this means that one definition of `V` blocks (or *kills*) any other definitions of `V` that reach it. We can model this behavior by manipulating the control flow graph as follows.[2]

[1] Because the solution for each variable is independent, it may be easy to perform this analysis on a parallel processor.

[2] We assume here that basic blocks only contain one node. Having only one definition site per node in the program greatly simplifies the exposition, especially of the pseudo-code in the later sections. In practice, nodes that contain more than one definition of `V` would be split once for each definition.

Any node in the control flow graph that contains a definition of V is split in the middle of the assignment. The right-hand side of the assignment is placed in the first node and the left-hand side is placed in the second node. This allows the first definition of V to reach the uses on the right side of the second definition of V. No control flow graph edge is added between the two resulting nodes. Effectively, there are now no control flow paths leading to definition sites. The control flow graph with split nodes is call the *derived graph*. Figure 2 shows this process.

Figure 2. Splitting of a Node That Contains a Definition of V

It is important to note that this splitting process is correct only because we are determining the solution for one variable at a time. In the reaching definitions problem, definitions of one variable do not generally effect other variables unless aliasing is considered, an area beyond the scope of this paper.

3.1.3 Collapsing the Strongly Connected Regions

Now that there are no control flow edges leading to nodes that contain definitions, it is easy to define the solution to the reaching definitions problem:

> The set of definitions that reach node N is the union of the sets of definitions that reach the predecessors of N in the derived graph.

Such a definition is recursive and in the presence of cycles in the derived graph would normally require iteration. However, there is some structure in this problem that can be utilized to attain an efficient solution.

- No node that contains a definition for V is involved in a cycle in the derived graph. (This is true since there are no edges into a node that contain a definition.
- Each node within a cycle in the derived graph has exactly the same set of reaching definitions. (This is true since there is a path from any node in the cycle to any other node in the cycle and there are no definitions in the cycle.)

Thus, the problem has been reduced to one of determining reachability in a directed graph. If each strongly connected component is collapsed to a single node, the resulting graph is a directed acyclic graph (or DAG). This is called the *reduced graph*. An algorithm developed by Tarjan [Tar72] finds the strongly connected components of a directed graph in $O(N + E)$ time.

3.1.4 Propagating the Information

The definitions that reach each node in the reduced graph can now be determined in one pass through this graph. Each definition is represented as one position in a bit vector, and there is one bit vector for each node in the reduced graph. A set bit in position i of the vector at node P means that the ith definition of the variable reaches node P.

Initially, nodes with no in-edges are assigned bit vectors. A node that contains the ith definition has a bit vector of all 0s except for the ith position; other nodes with no in-edges are assigned vectors with all zeros.

The recursive definition of reaching can be applied directly to the reduced graph, since it is a DAG, by visiting the nodes in topological order. (One of the benefits of using Tarjan's algorithm for finding the strongly connected components is that the resulting graph is already topsorted.)

The value of the bit vector B_N at node N with one or more in-edges is defined as follows: let S be the set of immediate successors of N then

$$S_N = \bigvee_{i \in S} B_i$$

3.1.5 Mapping the Information Back

Because the information at each node within a cycle of the derived graph is the same, mapping the information back to the original control flow graph is straightforward: each node in the derived graph simply gets a pointer to the node that represents its strongly connected region in the reduced graph. Any node in the original control flow graph that contains a definition for `V` has been split into two nodes, each with a different reaching definition set. These two sets represent the reaching definitions at different points within the original control flow graph node. (This information is sometimes referred to as the *local data-flow information*).

In many cases, it is necessary to keep only a small amount of the information gathered. For the reaching definitions problems, one usually needs to know only which definitions of variable `V` actually reach uses of `V`. If space is a problem, the bit-vectors at all other nodes can be discarded before the problem is solved for the next partition.

3.2 Some Generalizations

In this section, several aspects of the algorithm presented in the previous section are generalized to a wider class of data-flow problems.

3.2.1 The Choice of Functions

Following Kildall [Kil73], data-flow analysis problems can be modeled as follows: we choose a semilattice L and attach to its elements a *meaning*, normally data which could reach a point in a flow graph. We associate with each node of the flow graph a function f from L to L that intuitively represents how data is transformed when control passes through the block of code represented by that node.

Having selected a semilattice L and some intended meaning for lattice elements, we say that the admissible functions are those that reflect the actions of straight-line blocks of code on elements of L. We abstract the notation of such a set of functions in the following definition.

Given a bounded semilattice L, a set of functions F on L is said to be an *admissible set of functions for* L if and only if the following conditions are satisfied:

1. Each $f \in F$ distributes over meet, $\cap$:

$$(\forall x, y \in L)(\forall f \in F)[f(x \cap y) = f(x) \cap f(y)]$$

2. There exists an identity function i in F such that

$$(\forall x \in L)[i(x) = x]$$

3. F is closed under composition, i.e. $f, g \in F \rightarrow f \bullet g \in F$, where

$$(\forall f, g \in F)[f \bullet g(x) = f(g(x))]$$

4. For each $x \in L$, there exists a finite subset $H \subseteq F$ such that

$$x = \bigcap_{f \in H} f(\perp)$$

Further restrictions are applied by Graham and Wegman [GW76] and Kam and Ullman [KU76]. Graham and Wegman characterize a data-flow problem as being *fast* if it satisfies the following criterion in addition to the admissible functions presented above:

$$(\forall f \in F)(\forall x \in L)[f(f(x)) \geq f(x) \cap x]$$

Kam and Ullman characterize a data-flow problem as *rapid* if it satisfies the following criterion in addition to the admissible functions:

$$(\forall f, g \in F)(\forall x \in L)[f(g(\perp)) \geq g(\perp) \cap f(x) \cap x]$$

Thus the rapid problems are a subset of fast problems. The PVT algorithm places further restrictions on the set of admissible functions, and the set of problems solvable using PVT is thus a subset of the rapid problems.

In the reaching definitions problem, the *meet* operation *set union* can be implemented using bit-vectors and the binary "or" operation. Let L be a semilattice for a data-flow problem and let M be a function space over L. We say that *(L,M)* is *partitionable* if:

1. There exist semilattices $L_1 \ldots L_n n \geq 1 | l \in L$ can be represented as a tuple $< l_1 \ldots l_n >$ where :
   (a) $l_i \in L_i \, \forall i \, 1 \leq i \leq n$
   (b) $u =< u_1 \ldots u_n >, v =< v_1 \ldots v_n > \rightarrow u \cap v =< u_1 \cap v_1 \ldots u_n \cap v_n >$
2. For each $f \in M \, \exists (f_1 \ldots f_n) | \, \forall l = (l_1 \ldots l_n) \in L$:
   (a) $f(l) =< f_1(l_1) \ldots f_n(l_n) >$ where $f_i(l_i) \in L_i \, \forall i \, 1 \leq i \leq n$
   (b) each f_i is of the form $f_i(v) = c$ or $f_i(v) = v \cap c$ where c is a constant.

   Such functions are called *simple meet linear.*

Rule (1) allows the data-flow problem to be *partitioned* into a series of independent problems. The algorithm given here solves the data-flow problem for a single partition, and thus a separate pass of the algorithm is required for each partition of the problem. For most common data-flow problems a separate partition must be solved for each variable in the program. This method is therefore called the *Partitioned Variable Technique* (PVT).

The reaching definitions problem was described as being *partitionable* since the solutions for all of the definitions for a single variable are solved for at one time. Each partition corresponds to several pieces of information that can be propagated simultaneously. There are several data-flow problems in which this generality is not necessary. These *single-bit* problems have the property that only one piece of information is propagated for each variable. An example of this problem is the *live variable problem*, in which the goal is to determine whether *any* definition of the variable reaches a node in the program.

The PVT algorithm can solve both the partitionable and the single-bit problems. The single-bit problems can also be solved by the related but simpler, algorithms of Kou [Kou77] and Babich and Jazayeri [BJ78].

3.2.2 Choice of Splitting Sites

In the reaching definitions problem, the sites at which definitions occur are used both to split the nodes and to act as the source of the information to be propagated. The functions need not be combined. We can define three unique types of sites within nodes: Sites within nodes where information propagation *begins* are called *B sites.* Sites within nodes where information propagation *ends* are called *E sites.* Sites within nodes where the node must be *split* are called *S sites.* B sites, E sites, and S sites can be in the same node.

3.2.3 Direction of Propagation

Data-flow problems can be divided into categories depending on the direction in which information must be propagated to produce the desired result. *Forward flow problems* are those which ask what can happen before control reaches given a point in the program (i.e., what definitions can affect computations at that point). *Backward flow problems* are those which ask what can happen after control leaves given a point in the program (i.e. what uses can be affected by computations at that point).

The PVT algorithm can solve either type of problem. The reaching definitions problem is an example of a forward problem. Backward problems can be solved by modifying the step in Section 3.1.4 to perform the propagation using the inverse of the topological ordering.

```
procedure OfflineAnalyze(V, NodeList, EdgeList, B_site, E_site, S_site, RegularInfo)
   Stack ← ∅

   RegularVisited[*] ← new                                                    (1)
   Count ← 1
   ReducedEdges[*] ← ∅
   for each node in E_site[V] ∩ NodeList do                                   (2,3)
      ReducedEdges[node] ← EdgeList[node]
      for each n in EdgeList[node] do
         if ( RegularVisited[n] = new )
            then BuildReducedGraph(V, NodeList, EdgeList, B_site, S_site, n)
      end
   end

   ReducedVisited[*] ← new                                                    (1)
   ReducedInfo[*] ← ∅
   for each node in E_site[V] ∩ NodeList do                                   (3)
      DepthFirstSearch(V, RegularToReduced[node])                             (4)
   end

   for each node in RegularNodes do                                           (5)
      RegularInfo[V, node] ← ReducedInfo[RegularToReduced[node]]              (6)
   end
end
```

Figure 3. OfflineAnalyze

3.3 The Complete Offline Algorithm

This section contains the complete pseudo-code for the offline version of the PVT algorithm in Figures 3, 4, and 5. The numbers to the right of statements refer to the following notes:

1. **RegularVisited** is an array of flags, one for each node in the control flow graph. Each flag is initialized to **new**, indicating that the node has not been visited. After the node has been visited, the flag is set to **old**. **ReducedVisited** is handled similarly for the **ReducedGraph**.

2. The iteration over the **E_sites** and checking entry to **S_sites** effectively performs the splitting described in Section 3.1.2.

3. In several places, iteration is done over some set of nodes intersected with the **NodeList**. In the offline case, the **NodeList** is the entire set of nodes, so the intersection operation becomes a no-op. This is utilized in the online version presented later.

4. **DepthFirstSearch** propagates the information in the reduced graph as explained in Section 3.1.4.

5. In this loop, the data-flow is mapped back to the control flow graph from the reduced graph as explained in Section 3.1.5.

6. **RegularInfo** is a two dimensional array, the first dimension is indexed by the variable and the second dimension is indexed by the control flow graph node. **ReducedInfo** is reused on each iteration (by the variable). It is implemented as a one dimensional array indexed by the control flow graph node.

7. **BuildReducedGraph** is based on Algorithm 5.4 on pages 189-195 in [AHU74].

```
procedure BuildReducedGraph(V, NodeList, EdgeList, B_site, S_site, node)          (7)
   DFNumber[node] ← Count
   Count ← Count + 1
   LowLink[node] ← DFNumber[node]
   push node on Stack
   RegularVisited[node] ← onstack

   if (node ∉ S_site[V])                                                          (2)
      then do
         for each s in EdgeList[node] ∩ NodeList do                               (3)
            if (RegularVisited[s] = new)
               then do
                  BuildReducedGraph(V, NodeList, EdgeList, B_site, S_site, s)
                  LowLink[node] ← min ( LowLink[node], LowLink[s] )
               end

               else if ( ( DFNumber[s] < DFNumber[node] )
                     and ( RegularVisited[s] = onstack ) )
                  then LowLink[node] ← min(DFNumber[s], LowLink[node])
         end
      end
   ReducedInfo[node] ← RegularInfo[V, Node]
   if ( LowLink[node] = DFNumber[node])
      then repeat
         pop x from Stack
         RegularVisited[x] ← old
         RegularToReduced[DFNumber[x]] ← DFNumber[node]
         ReducedEdges[node] ← ReducedEdges[node] ∪ EdgeList[node]
         if ( x ∈ B_site[V])
            then ReducedInfo[node] ← ReducedInfo[node] ∪ x
      until ( x = node )
end
```

Figure 4. BuildReducedGraph

```
procedure DepthFirstSearch(V, node)                                             (4)
   ReducedVisited[node] ← old

   for each n in ReducedEdges[node] do
      rn ← RegularToReduced[n]
      if (ReducedVisited[rn] = new )
         then DepthFirstSearch(V, rn)

      ReducedInfo[node] ← ReducedInfo[node] ∪ ReducedInfo[rn]
   end
end
```

Figure 5. DepthFirstSearch

4 Partitioned Formulations of Common Problems

Several GDFAPs are formulated as a collection of equations and implemented as bit vectors. All of these problems meet Kam and Ullman's [KU76] criteria for admissible functions. Each of the problems given here

is also a *partitionable* problem as defined in Section 3.2 and therefore must be solved by a technique that can propagate a bit vector for each variable. The information produced by a solution to each of these problems is used to build some type of information chain. Five problems presented in this section; each has an analogue formulated as a *single-bit* problem.

4.1 Input Structures

The following information is required as input to the data-flow algorithms discussed here. All the information is local to a single basic block and is constructed by examination of that block. It is assumed that some front-end program either builds these structures in offline environments or maintains them in an online environment.

Each data structure is an array of sets. Each array has one element for each variable in the program.

L_Def is a set of locations of definition sites for a single variable **v**. Execution of the node containing this statement guarantees that the value of this variable will change. *L_Def* includes targets of assignment statements and parameters to input statements.

L_Mod is a set of locations of modification sites for a single variable **v**. Execution of the node containing this statement may cause the value of this variable to change. Assignment into single values of arrays and passing variables to subroutines are two statements that can generate *L_Mods*.[3]

L_UnDef is a set of locations at which values become undefined for a single variable **v**. Loop index variables in languages such as BCPL and Fortran become undefined in the exit node from the loop. Also, all local variables are initially undefined on entry to the subroutine or block and become undefined on exit from the subroutine or block.[4]

L_Use is a set of locations of use sites for a single variable **v**. This set is typically made from all variables that appear in any expression, in parameters to subroutines, and in output statements.

4.2 Output Structures

The following information is produced by the algorithms discussed here. Each of these structures is a two-dimensional array of sets. There is one element in the first dimension for each variable in the program, and one element in the second dimension for each location in the program.

G_Def The set for location **n** contains the set of *L_Defs* for **v** that reach **n**. This set is calculated top-down.

G_Mod The set for location **n** contains the set of *L_Defs* for **v** that reach **n**. This set is calculated top-down.

G_UnDef The set for location **n** contains the set of *L_UnDefs* for **v** that reach **n**. This set is calculated top-down.

G_Use The set for location **n** contains the set of *L_Uses* for **v** that can be reached from **n**. This set is calculated bottom-up.

4.3 The Data-Flow Problems

Reaching Definitions Determine the set, *G_Def*, for each node, **x**, of variable definitions that can enter node **x**. *Variable definition* is a statement that can potentially modify the value of a variable. This is a *forward* problem and the output is typically used to build the use-definition chains used in compiler optimization [Hec77].

[3] Parameters to subroutines must be put into both the *Use* and *Mod* set unless other information is known about the use of that parameter within the subroutine.

[4] This is not true for static variables.

Live Variables Determine the set, *G_Use*, for each node, x, of variables that are *live* or may be used after control passes out of node x. This problem is the *backward* analogue of the reaching definitions problem and its output can also be used to build use-definition chains [Hec77].

In addition to being used in compiler optimization, the information produced by a solution of this problem can be used directly to discover if the output of a definition site is ever used. This has applicability in the area of software reliability, since unused definitions may be the result of programming errors.

Unsafe Uses Determine the set, *G_UnDef*, for each node x of variable undefinitions that can reach node x. A use of a variable is *unsafe*, for each node x, if there exists a path that the program may take to node x, at which the variable has not been defined. By characterizing the problem in this manner, we not only find out whether the *use* is safe, but also determine the location of the kill site if the *use* is unsafe. This problem is arrises in studies of software reliability [FO76].

Value Printing Problem This problem is broken into two subproblems, the *Forward Printing Problem* and the *Backward Printing Problem*. The output of these two problems is then processed to find the set of nodes within which a particular value of a single variable is live. These two problems were defined by Fosdick and Osterweil in the *DAVE* system, and have so far been used only in the area of software reliability [FO76].

These two problems have never been formulated for solution by a classical technique. The technique used by Fosdick and Osterweil, similar to the technique proposed by Kou, was used because the number of Use-Def pairs that were needed was small compared to the number of possible Use-Def pairs in a given program.

In the *Forward Printing Problem*, a set of nodes is created for each *L_Def*, *L_Mod* or *L_UnDef* in the program. The set contains each node that is reachable from each beginning node without passing through another *L_Def*, *L_Mod* or *L_UnDef*. In the *Backward Printing Problem*, a set of nodes is created for each *L_Use* in the program. The set contains each node that is reachable from each *Use* node without passing through another *L_Def*, *L_Mod* or *L_UnDef*.

Problem	*Class*	*B sites*	*E sites*	*S sites*
Reaching Definitions	*Pred*	*L_Def* ∪ *L_Mod*	*L_Use* ∪ *L_Mod*	*L_Def* ∪ *L_UnDef*
Live Variables	*Succ*	*L_Use* ∪ *L_Mod*	*L_Def* ∪ *L_Mod*	*L_Def* ∪ *L_UnDef*
UnSafe Uses	*Pred*	*L_UnDef*	*L_Use* ∪ *L_Mod*	*L_Def* ∪ *L_UnDef*
Forward Printing	*Pred*	*L_Def* ∪ *L_Mod*	*Node*	*L_Def* ∪ *L_UnDef*
Backward Printing	*Succ*	*L_Use* ∪ *L_Mod*	*Node*	*L_Def* ∪ *L_UnDef*

Figure 6. Formulations for Partitionable Data-Flow Problems

Figure 6 summarizes the problem descriptions. The "∪" means set addition, i.e and location in either of the sets.

4.4 Problems Not Solvable by PVT

As discussed in Section 3, the PVT algorithm works by dividing the problem into small *partitions* that are solved separately. The key property of a partition is that each *strongly connected component* of a partition has the same information available for each node of that component. This allows the strongly connected regions to be reduced to single nodes, transforming the program flow graph into a directed acyclic graph. A data-flow problem that is amenable to this technique is called a *partitionable problem.*

While it is true that the PVT algorithm can solve most of the *partitionable* problems that are *rapid*, it should not be inferred from this that the PVT algorithm is powerful enough to solve all problems in these classes. The following problem is *rapid* but cannot be solved by PVT:[5]

[5]Kildall [Kil73] describes a similar problem that uses the minimum path distance. Whenever all registers are busy and contain live expressions, the register containing the live expression with the largest distance to its occurrence is displaced.

> Suppose you wish to find the length of the shortest path from each node in a control flow graph to every other node in the graph. The formulation of this problem is $\cap$ = `min` for each element in the partition. There are no *P sites* or *S sites*. The operation at each *B site* is a *meet* with the constant vector:
>
> $$[\infty, \infty, \ldots 0, \ldots \infty]$$
>
> where the number in the ith position for node i is 0.

The formulation of this problem does not satisfy the definition of PVT since an additional operation must be done at the *P nodes*: the value of each path must be incremented by 1 to accumulate the path length. Intuitively, this problem cannot be solved by the PVT since information is lost when the strongly connected regions are collapsed and this information (the length of the shortest path) is utilized in this problem. This problem can, however, be solved by the techniques of Kam and Ullman or Graham and Wegman.

As a second example, consider a common data-flow problem that is not *distributive*, such as constant propagation. This problem is described by Hecht [Hec77] as:

> Try to discover, at the top of each node in the flow graph, those variables that are assigned the same constant value on all paths from the initial node to the top of a given node.

This problem is neither *fast* nor *rapid* since new information may be acquired each time propagation proceeds around any loop. Additionally, this problem has another characteristic that the PVT cannot handle: it fails to be partitionable. In the constant propagation problem, the propagation of a constant to a new node may make other constant propagations possible. There is no way to make the partitions independent of each other.

5 Incremental Data-Flow Analysis

There are two primary motivations for this work, one from compiler optimization and the other from software engineering. In the area of compiler optimization, incremental data-flow analysis can make the program transformations more effective and efficient. Making one transformation frequently exposes opportunities to perform other transformations. Incremental analysis can update the data-flow information as these transformations are made.

In the area of software engineering, incremental data-flow analysis can maintain data-flow information while the program is being edited by the programmer. The algorithms can be used in a structure program editor to give the programmer correctly updated data-flow information as the program is being modified. This analysis produces information that relates to the correctness of the program.

Transformations made in both of these areas have two properties: the granularity of changes is small, and the changes are well structured. It is expected that the affected area of these changes is also fairly small in most cases. It is therefore justified to use algorithms that can exploit this locality. The algorithms presented here behave in a manner that is uniquely suited to this environment.

The idea of performing data-flow analysis in a programming environment is not new. The *Dave* system [FO76] by Fosdick and Osterweil produces such information though in a batch environment. The *Scope* system [Mas80] produces a less precise form of information and is driven only by specific programmer queries. Ryder and Carroll [RC87] propose other incremental algorithms to perform interprocedural analysis in a programming environment. The algorithms presented here are currently being implemented in the *Para-Scope* tool which is part of Rice University's $\mathcal{R}^N$ project [CCH+88].

5.1 The Incremental Model

Here we provide algorithms to maintain data-flow information in the presence of incremental changes to the program. It is convenient to classify the types of changes that can be made, since different algorithms are required to respond to each type. The changes can be classified into three categories:

Type I operations are changes to individual references in the program.

Type II operations are changes that add or delete edges (control flow paths) to the program.

Type III operations are changes that add or delete basic blocks to the program.

Two GDFAPs discussed in Section 4, the Live Variable Problem and the Unsafe Uses Problem are used as examples here. It is assumed that the *L_Use, L_Def, L_UnDef*, and *L_Mod* sets defined in Section 4.1 are maintained by system managing the changes. The algorithms presented here compute the new values for the *G_Use, G_Def, G_UnDef*, and *G_Mod* sets defined in Section 4.2.

The strategy used is to determine the area of the control flow graph that is affected by the changes. This is done with the **FindAffectedArea** routine given in Section 5.2. An incremental version of the offline algorithm called **OnlineAnalyze** is then used to recompute the data-flow information in the affected area; this is also given in Section 5.2. A description of how these algorithms are used to update the data flow information after Type I, Type II and Type III changes is given in Sections 5.3 through 5.5 respectively.

5.2 Special Local Algorithms

```
procedure FindAffectedArea(V, EdgeList, S_site, ChangeLoc, NodeList)
   NodeList ← NodeList + ChangeLoc
   WorkList ← EdgeList[ChangeLoc]
   while ( WorkList ≠ ∅ ) do
      choose any t from WorkList
      WorkList ← WorkList - t
      NodeList ← NodeList + t
      if (t ∉ S_site[V])
         then WorkList ← WorkList ∪ (EdgeList[t] - NodeList)
   endwhile
end
```

Figure 7. FindAffectedArea

The procedure **FindAffectedArea**, shown in Figure 7, is used to find the area affected by a single change in either the forward or backward direction. **FindAffectedArea** calculates the list of nodes that are affected by a change at a particular section of the graph. This list is then used as input by the algorithms that actually compute the new sets. **FindAffectedArea** has five parameters:

V The variable affected by the change.

EdgeList The set of edges that connect the nodes. This list may be either the *Successor* set for *backward problems* or the *Predecessor* set for *forward problems.*

S_site The set of nodes that contain *S sites* for the partition.

ChangeLoc The site of the change.

NodeList The list of nodes that have been visited.

The worst case complexity of **FindAffectedArea** is $O(Nodes + EdgeList)$. Using **NodeList** assures that no node is visited more than once. The **NodeList** field of each node must be checked once for each in-edge. These algorithms modify the graph only for a single value of a single variable. The space required is $O(Nodes * SetSize)$ where *SetSize* is that size of the local and global sets computed. Unlike the offline algorithms, the solution for each variable must be retained since this will be used as the starting point for further solutions.

The expected case complexity should be much better than the number of nodes in the graph. **FindAffectedArea** visits only those nodes through which a single value can flow. While sometimes the entire graph must be visited, many values are short-lived.

The algorithm **OfflineAnalyze**, shown in Figure 3, is used to propagate information from several *B_sites* to other nodes. The **OnlineAnalyze** algorithm in Figure 8 is a modification of the offline version that visits only those nodes discovered by the call to **FindAffectedArea**.

```
procedure OnlineAnalyze(V, NodeList, EdgeList, B_site, E_site, S_site, RegularInfo)
   Stack ← ∅

   Count ← 1
   ReducedEdges[*] ← ∅
   for each node in E_site[V] ∩ NodeList do                                    (1)
      ReducedEdges[node] ← EdgeList[node]
      for each n in EdgeList[node] do
         if ( RegularVisited[n] = new )
            then BuildReducedGraph(V, NodeList, EdgeList, B_site, S_site, n)
      end
   end

   for each node in NodeList do
      ReducedVisited[RegularToReduced[node]] ← new                             (2,3)
      ReducedInfo[RegularToReduced[node]] ← ∅                                   (4)
   end
   for each node in E_site[V] ∩ NodeList do                                    (1)
      DepthFirstSearch(V, RegularToReduced[node])
   end

   for each node in NodeList do
      RegularInfo[V, node] ← ReducedInfo[RegularToReduced[node]]
      RegularVisited[node] ← new                                               (3)
      ReducedVisited[RegularToReduced[node]] ← outside
   end
end
```

Figure 8. OnlineAnalyze

Several changes are required to make the offline algorithms suitable for the online environment. As before, the numbers to the right of the statements refer to the notes below:

1. Here the intersection with **NodeList** is significant. **NodeList** is the set of nodes found by **FindAffectedArea**. This statement defines the iteration to be performed over all **E_sites** in the affected area.

2. The visited array **ReducedVisited** now has three possible states rather than two. The new state is **outside** and signifies that the node in question is outside the affected area by the changes and need not be traversed.

3. The arrays **RegularVisited** and **ReducedVisited** are initialized once to **new** and **outside** before this algorithm is run. These statements reset the elements of these arrays modified by this invocation of **OnlineAnalyze** so that they are ready for the next invocation of **OnlineAnalyze**.

4. In the array **ReducedInfo**, only the elements that will be used in this invocation of **OnlineAnalyze** are initialized.

5. The **ReducedVisited** flag is examined when propagating the data-flow information. If the node from which the information comes is outside the affected area, the information must come from the unreduced version of the graph in which the information has not been affected by this incremental change.

```
procedure DepthFirstSearch(V, node)
   ReducedVisited[node] ← old

   for each n in ReducedEdges[node] do
      rn ← RegularToReduced[n]
      if (ReducedVisited[rn] = new )
         then DepthFirstSearch(V, rn)

      if (ReducedVisited[rn] = outside )                                    (2)
         then ReducedInfo[node] ← ReducedInfo[node] ∪ RegularInfo[V, n]     (5)
         else ReducedInfo[node] ← ReducedInfo[node] ∪ ReducedInfo[rn]
   end
end
```

Figure 9. Online Version of DepthFirstSearch

5.3 Algorithm For Type I Changes

Type I operations are any changes to a definition or use of a variable. Type I operations do not affect the control flow or block structure of the program. While each Type I change is made to a single location, the effect of a single change extends over a wider area, possibly the entire program. Changes may be in assignment or I/O statements or in the expressions of conditional branches or loops.

Here we consider how to keep the four global sets *G_Use, G_Def, G_Undef,* and *G_Mod* up to date as changes are made to the four local sets *L_Use, L_Def, L_Undef,* and *L_Mod.* Figure 10 indicates which global sets can be affected by a change to a local set.

- Type I algorithms examine only a single variable, permitting them to use the actual *L_Use, L_Def, L_Mod* and *L_UnDef* information for that variable to pick an optimal path through the flow graph at run time.
- Type I algorithms do not look at all the values of the variable. The algorithms deal only with those values that either affect or are affected by the part of the program pointed to by the editing cursor.
- Type I algorithms will track this value only through part of the program, the extent over which a single value of the variable is live.

Local Changed Set	Affected Set
L_Def	*G_Use, G_Def, G_Undef, G_Mod*
L_Undef	*G_Use, G_Def, G_Undef, G_Mod*
L_Mod	*G_Mod*
L_Use	*G_Use*

Figure 10. Table Indicating Which Global Sets Are Affected by Local Set Modifications

The general plan is to batch together as many changes for a single variable as possible[6] and then to process all of those changes at one time. It is advantageous to batch together as many changes for one variable as possible because the affected areas for many of the changes can overlap. **FindAffectedArea**'s search strategy never looks at nodes that have already been added to **NodeList** by a different change. Therefore the total cost of batching the changes together is the sum of the number of edges out of nodes in the affected area plus the number of edges out of all change sites. Since nodes within the affected areas of two or more changes are examined only once, batching can yield substantial benefit.

[6] Remember that since the problems that we are solving are partitionable, no change to one in one variable affects the solution of another variable.

```
for each variable V that is associated with a change do                    (1)
   PredNodeList, SuccNodeList ← ∅                                          (2)
   for each change C that affects variable V do
      update local sets affected by C
      if C affects G_Use                                                   (3)
         then FindAffectedArea(V, Pred, S[Use], C.Loc, PredNodeList)
      if C affects G_Def, G_Mod, G_UnDef                                   (3,4)
         then FindAffectedArea(V, Succ, S[Def], C.Loc, SuccNodeList)
   end
   OnlineAnalyze(V, PredNodeList, Pred, B[Use], E[Use], S[Use], G_Use)     (3)
   OnlineAnalyze(V, SuccNodeList, Succ, B[Def], E[Def], S[Def], G_Def)
   OnlineAnalyze(V, SuccNodeList, Succ, B[Mod], E[Mod], S[Mod], G_Mod)
   OnlineAnalyze(V, SuccNodeList, Succ, B[UnDef], E[UnDef], S[UnDef], G_UnDef)
end
```

Figure 11. General Algorithm for Type 1 Changes.

Affected Set	*EdgeList*	B[]	E[]	S[]
G_Use	*Pred*	*L_Use*	*L_Def* ∪ *L_UnDef* ∪ *L_Mod*	*L_Def* ∪ *L_UnDef*
G_Def	*Succ*	*L_Def*	*L_Def* ∪ *L_UnDef*	*L_Def* ∪ *L_UnDef*
G_Undef	*Succ*	*L_Undef*	*L_Def* ∪ *L_UnDef*	*L_Def* ∪ *L_UnDef*
G_Mod	*Succ*	*L_Mod*	*L_Def* ∪ *L_UnDef*	*L_Def* ∪ *L_UnDef*

Figure 12. Table Indicating the Values of Various Parameters to `OnlineAnalyze`.

The general method for updating the data-flow information after a set of Type I changes is outlined in Figure 11. The numbers to the right of the statements refer to the notes below:

1. The changes are processed in an order such that all changes for a single variable are done together. This can be accomplished by sorting the changes by variable name.

2. Two lists of nodes are accumlated: `PredNodeList` is used for all changes to `L_Use` and `SuccNodeList` is used for all changes to `L_Def`, `L_Mod` and `L_UnDef`.

3. For each change, `FindAffectedArea` is called to determine the nodes at which information must be updated. See Table 10 to determine which modifications affect which global sets.

4. Note that since the `S_sites` are the same for the sets *G_Def*, *G_Undef*, and *G_Mod*, it is possible to use the same node list for all of these sets.

5. Once the total affected areas for all of the changes has been determined, calls are made to `OnlineAnalyze` to update the information. Figure 12 gives the values for the parameters `EdgeList`, `B[]`, `E[]`, and `S[]` used in `OnlineAnalyze` in order to compute the various output sets. The parameters are constructed by unioning several of the local sets together. Since the parameters are made up of only local sets and the parameters are organized in the same way as the local sets, it is easy to keep them up to date as the local sets change.

One property of the Type I algorithm is that each node discovered by `FindAffectedArea` has different different data-flow information after the invocation of `OnlineAnalyze` than before. Since the amount of work required in `FindAffectedArea` and `OnlineAnalyze` is linear in the number of nodes in the affected area, it is possible to claim that the Type I algorithm is optimal for these changes. This is similar to the criterion used by Reps [Rep84][7]. Unfortunately, this behavior is not maintained for the totally unstructured Type II changes described in the next section.

[7] In fact, this is more general than the criteria used by Reps since there is no dependence on a cursor movement to move from one change point to another nor are there any dependences on the type of graphs being used.

5.4 Algorithms For Type II Changes

Type II changes are arbitrary changes to the control flow graph and are the result of the addition or deletion of unrestricted `goto` statements. (A different algorithm is presented in Section 5.5 for a restricted set of changes to the control flow graph.)

- Type II algorithms must look at all variables that are active in the region changed.
- Type II algorithms do not look at all values of the variable, but rather deal only with those values that either affect or are affected by the part of the program pointed to by the editing cursor.
- Type II algorithms track these values only through the part of the program in which a single computed value is live.

A Type II change is specified by two pointers to nodes in the control flow graph pointing to the source `SourceP` and destination `DestP` of the edge being manipulated.

Algorithms are given for the two general problems of deletion and addition of an arbitrary edge.

5.4.1 Addition of an Edge

There is one property unique to the edge addition operation that is utilized in the edge addition algorithm: edge addition only adds elements to the global sets, none are ever deleted. This property allows the step where the information is initialized to be skipped. Thus, `FindAffectedArea` is not used when adding an edge to a graph. Instead, a worklist technique similar to that in `FindAffectedArea` is merged into the edge addition algorithm.

The edge addition algorithm has two criterion for stoping the propagation of information. The first are the `S_sites` and the second are the global sets themselves. Propagation only continues as long as the sets to be propagated are not subsets of the prior existing global sets. This keep those subgraphs from being visited in which the information is already correct. The already correct subgraphs are the result of other paths through the control flow graph that carry the same information as the newly added edge.

```
AddEdge(Pred, S[Use], SourceP, G_Use)
AddEdge(Succ, S[Def], DestP, G_Def)
AddEdge(Succ, S[Mod], DestP, G_Mod)
AddEdge(Succ, S[UnDef], DestP, G_UnDef)

AddEdge(EdgeList, S_Site, ChangeLoc, Info)                                  (1)
   for each variable V                                                      (2)
      NodeList ← ChangeLoc
      WorkList ← EdgeList[ChangeLoc]
      while ( WorkList ≠ ∅ ) do
         choose any node from WorkList
         WorkList ← WorkList - node
         NodeList ← NodeList + node
         if ((node ∉ S_site[V]) or (Info[V, node] ⊃ Info[V, ChangeLoc])      (3)
            then WorkList ← WorkList ∪ (EdgeList[node] - NodeList)
         Info[V, node] ← Info[V, node] ∪ Info[V, SourceP]                    (1)
      end
   end
end
```

Figure 13. General Algorithm for Edge Additions

The algorithm to add edges, a modification of the `FindAffectedArea` algorithm, is shown in Figure 13. The numbers to the right of the statements refer to the notes below:

1. The computed result of **AddEdge** is a new value of **Info** rather than the **NodeList** of **FindAffectedArea**.

2. Every variable is affected by a new edge; every variable must therefore be checked.

3. Unlike **FindAffectedArea**, the search is terminated at **S_sites** and when the propagated information is not a superset of the information at the current node.

5.4.2 Deletion of an Edge

It is not possible to use the strategy of starting with the existing information as in the edge addition algorithm. In the case of an edge addition, each global set can change only by adding elements. Elements are valid in the set if they came from an old edge entering a node or the new edge that was added; it does not matter which. When an edge is deleted, it does matter where the elements came from. The elements can come from the remaining edge into the node, from the edge to be deleted, or both. When the information comes from both edges, a difficult case can occur in the presence of loops in the control flow graph.

In data-flow problems, the goal is to determine the largest fixed point solution. In the problems considered here, it is guaranteed that the largest fixed point solution is unique. In incremental analysis, an algorithm is considered correct only if it computes exactly the same solution as the offline algorithm.

It is easy to show that when we add an edge to the graph, the resulting fixed point is never any smaller than the fixed point that existed before the edge was added. Therefore, this problem is easy to solve: start from the existing solution and just add new elements. When we delete an edge, the question is to determine which elements of the solution are to be deleted. The solution that is chosen here is to start from a point sufficiently low so as to be guaranteed to be below the final fixed point and work our way up to the correct fixed point. This problem has been studied elsewhere and is known to be quite difficult. Ryder, Marlowe, and Paull [RMP87] consider how to determine the conditions to restart iteration.

```
for each variable V do                                                    (1)
   PredNodeList, SuccNodeList ← ∅
   for each edge E to be deleted do                                       (2)
      FindAffectedArea(V, Pred, S[Use], C.SourceP, PredNodeList)
      FindAffectedArea(V, Succ, S[Def], C.DestP, SuccNodeList)
   end
   OnlineAnalyze(V, PredNodeList, Pred, B[Use], E[Use], S[Use], G_Use)    (3)
   OnlineAnalyze(V, SuccNodeList, Succ, B[Def], E[Def], S[Def], G_Def)
   OnlineAnalyze(V, SuccNodeList, Succ, B[Mod], E[Mod], S[Mod], G_Mod)
   OnlineAnalyze(V, SuccNodeList, Succ, B[UnDef], E[UnDef], S[UnDef], G_UnDef)
end
```

Figure 14. General Algorithm for Edge Deletions

The algorithm to delete edges, a modification of the Type I algorithm, is shown in Figure 14. The numbers to the right of the statements refer to the notes below:

1. Every variable in the program must be processed.

2. Several edge deletions can be handled at once. Like the Type I changes, the algorithm may perform better if several edge deletions are batched, since overlapping affected areas need be processed only once.

3. Once the affected areas are determined for each variable, **OnlineAnalyze** is called to update the global sets. The values of **B[]**, **E[]**, and **S[]** are defined in the same way as for Type I changes.

The strategy used here is to remove all information that is propagated from the deleted edge, and then rebuild the information in the affected area by propagating the information from the in-edges of that affected region. The algorithm is the same as the algorithm for Type I changes except that it contains a loop around

it so it is executed for each variable in the program. All four sets *G_Def, G_UnDef, G_Mod* and *G_Use* must be updated.

The ramifications of using this algorithm is that the optimality result claimed for Type I algorithms does not hold here. In the Type I algorithm, every node that was marked by `FindAffectedArea` is guaranteed to have some change in the value of some of the global sets for that node. This is not true here. In the Type I algorithms, at least one bit changed in some local set around the change, and that bit must be propagated to any node reachable without going past a `S_site`. For that reason, we could wipe out the global information for that variable at all nodes in the affected area with the understanding that the cost of putting it back could be charged to the bit that must be changed. For Type II changes we have no bit to which to charge this information; therefore, we may (and usually do) delete and restore information from nodes whose values do not change.

5.5 Algorithms for Type III Changes

Type III changes are a special subset of the Type II changes, and a fast algorithm exists to update the data-flow information after some of these changes. While the set of Type III changes is small, it is quite important, since many of the operations performed in a programming environment can be formulated as compositions of Type III operations. Type III are illustrated in Figure 15 and operations occur in the following situations:

1. Serial splitting or combining of a basic block.
2. Parallel splitting or combining of a basic block.
3. Addition or deletion of a self loop.

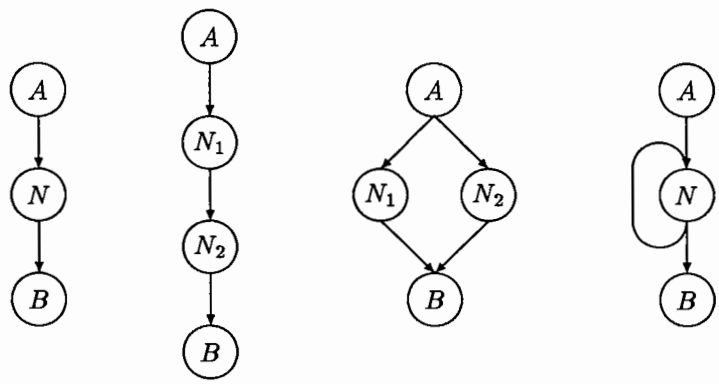

Figure 15. Original Graph, After Serial Splitting, After Parallel Splitting, and After Addition of a Self Loop

Many of the editing operations in structure based editors can be formulated as combinations of these three operations. For instance, the operation of adding a loop to a program can be modeled as first serially splitting an existing node and then adding a self loop to the new node.

The key idea here is that each of these operations involves only the variables located within those actual blocks that are changed. We can therefore model the changes as a series of Type I changes for the variables in the local sets for those blocks, rather than having to perform an operation over every variable in the program, as is done with the general Type II changes. The general algorithm is shown in Figure 16.

```
for each V used or defined in node N do
   reinsert that use or definition using the Type I algorithm
end
```

Figure 16. The General Type III algorithm

6 Comparison with Other Work

Data-flow problems can be classified by their complexity. In general, as data-flow problems become harder, more time and space must be expended to produce a solution. Figure 17 shows several of the standard classes and gives references for both online and offline solution techniques. The table is sorted by class; the top entry is the largest class.

Class	Offline Algorithm	Online Algorithm
Interprocedural	[Bar77, Ban79]	[Bur87]
Distributive	[Kil73]	[CR88]
Fast	[GW76]	[CR88]
Rapid	[KU76]	[CR88]
Partitionable	this paper	this paper
Single Bit	[BJ78, Kou77]	[BJ78]

Figure 17. Data-Flow Analysis Classes

6.1 Comparison with Algorithms for More Restricted Classes

With the exception of the constant propagation problem, most GDFAPs used in compilers fall into the partitionable class. This is not true for the single-bit class. Problems such as reaching definitions or building definition-use chains are difficult to solve with the single-bit algorithms and they have not been heavily used for this reason.

6.2 Comparison with Algorithms for Less Restricted Classes

As mentioned in Section 4.4, few problems are fast or rapid that are not partitionable. Obviously, if it is necessary to solve a non-partitionable problem a more general algorithm is required; but if the problem is partitionable, both the PVT algorithm and the algorithm of Carroll and Ryder [CR88] (referred to as CR below) may be applicable.

- PVT works for all programs, CR works only for programs that are reducible. In practice, only a small percentage of Fortran procedures are not reducible,[8]
- If the program changes are mostly Type I, PVT is the method of choice; if a significant percentage of the changes are Type II, CR may be preferred.

  The amount of work required for a Type I change using PVT is never any larger than CR and in many cases will be smaller. The difference is a result of what each technique takes into account to determine the effective area. In CR, only the control flow graph is used; in PVT, both the control flow graph and the locations of `E_sites` are used. In many cases an `E_site` stops the propagation of a change that otherwise could spread throughout the program.
- Both CR and PVT can handle structural changes. In both techniques the structural changes fall into two groups: those that can be done quickly and those that cannot. The PVT algorithm divides the changes into Type II (the general class of changes) and Type III (the smaller class that can be handled efficiently). The CR algorithm divides changes into those that upset the dominator tree (the general class of changes) and those that do not (the smaller class that can be handled efficiently).

  The PVT's fast set of changes is smaller than that of CR; on the other hand less work is required to process those PVT changes. PVT's general set of changes (Type II) is in theory faster to process than the general changes in CR. In practice, CR will probably be better.

[8] The percentage of non-reducible Fortran programs has been experimentally verified by Knuth [Knu71]; the percentage is probably neglible in Algol-like languages that are more conducive to block structuring. CR's claim that their algorithm is useful for interprocedural problems is more suspect. While it is true that most procedures are reducible, this is not the case for call graphs; common programming practices like mutually recursive procedures give rise to non-reducible call graphs.

For those structural changes that do not fit into the easy groups for either algorithm, the outlook is more grim. It is not clear that a well-implemented, efficient offline algorithm will not better the incremental algorithms most of the time. Luckily, in many environments, the changes are well structured and so lend themselves to efficient online techniques.

6.3 Comparison With Prior Versions of This Work

Preliminary versions of this work have appeared in a PhD dissertation [Zad83] and a conference paper [Zad84]. This paper contains many changes from those earlier versions.

Most significantly, in both prior works, it was expected that the main use of these algorithms would be in programming environments, and little or no consideration was given to the uses in compiler optimization. In the time that has passed, evidence has come to light that the applications to programming environments may not be as useful as had been hoped, but that the algorithms may be very useful in optimizing compilers.

The problem in programming environments is not that the algorithms are unsuitable for the problems, but that the problems themselves are not the correct ones to solve. Data-flow analysis is an approximation technique. General formulations of all of the problems can be developed that are either undecidable or require intractable amounts of time and space. The problems solved by data-flow techniques are thus weak approximations of the general problems. When we talk about a "conservative solution", we are saying that any predicate of which we prove the truth of is true under all possible executions of the program but that we may therefore fail to prove that some true predicates are true.

In the area of compiler optimization, this weakness is easily tolerated. Optimizations are performed only if the predicates involved are proven true; if not, the optimization is not performed.

In the area of software engineering, however, this weakness causes problems. The hope is that data-flow information can be used to detect anomalies in software. But what is an anomaly? Take the case of *UnDef*. It is a legitimate error if there are *executable paths* from the root of the program to a use of a variable `V` containing an *UnDef* for `V` without an intervening *Def* or *Mod* for `V`. But determining the executable paths for a computer program is undecidable.

```
int FT ← 1
int A

while (...)  do
   if FT
      then do
         FT ← 1
         A ← 6
      end
   ...  ← A
end
```

Figure 18. Difficult Case for Error Detection Techniques Based on Data-Flow Analysis

The best that we can hope for is a reasonably approximation to the problem of determining the executable paths. Data-flow analysis techniques use a "meet over all paths" approximation. There are two easily computable versions of this problem. We can decide that it is an error if there are *any paths* from the root of the program to a use `U` that contain an *UnDef*, or we can decide that it is an error if *all paths* from the root to `U` contain an *Undef*. Both problems can be solved with simple reachability. Errors of the first type can be determined by looking at each use `U` of variable `V` and asking if there are any places where *G_UnDef*`[V, U]` is not empty. Errors of the second type are true if *G_UnDef*`[V, U]` is not empty and both *G_Def*`[V, U]` and *G_Mod*`[V, U]` are empty. The problem is that the first definition will produce many spurious errors and the second will miss many errors.

The first problem is the one generally used in programming environments. Consider the example shown in Figure 18. If one understands that `FT` is used as a first-time flag, then it is possible to prove that no *UnDef*

reach the use of A. Since, in general, it is not possible to understand the paths in a computer program so precisely, we are left to produce a spurious error message. In practice a very large number of such messages are produced, making the analysis fairly useless.

Such analyses do prove useful when the programmer has a specific question such as "Where are the definitions that may reach this location?" This was the technique used by Masinter [Mas80]. Here the weakness of the data-flow solutions does not seem to cause any problems. On the other hand, since the questions are asked infrequently, it is not clear that it is justified to attempt to keep the data-flow information up to date as the program is being changed.

In optimizing compilers, the algorithms may be quite useful. In many techniques that are based on data-flow analysis, the compiler performs the analysis and then makes the transformations. In many cases those transformations expose opportunities to perform other transformations. But new analysis based on the transformed program must be performed before it is safe to make the transformation. In the past, new data-flow information was computed from scratch (as in the PL.8 compiler [AH82]). Here incremental analysis can help. In most cases, the transformed program is very similar to the untransformed program. Most optimization transformations can be expressed in a small number of Type I or Type III changes that can be performed quickly. It is, in fact, difficult to find a problem in compiler optimization that requires a Type II change.

7 Conclusions

In this paper, a new algorithm has been presented to perform data-flow analysis over a limited set of problems. The algorithm is capable of building use-definition chains and solving many similar problems. A precise characterization of problems solved has also been given. The asymptotic complexity of the global algorithm is $O(E \times V)$. An incremental version of this algorithm is also given, and the suitability of this algorithm in programming environments and in optimizing compilers is also explored.

Acknowledgments

I would like to thank Alan Demers, and Ken Kennedy who helped me formulate and express the ideas in the original version of this work and Keith Cooper who helped with both version.

References

[AH82] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, 17(6):22–31, June 1982.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[All70] F. E. Allen. Control flow analysis. *Sigplan Notices*, July 1970.

[Ban79] J. B. Banning. An efficent way to find the side effects of procedure calls and the aliases of variables. *Conf. Rec. Sixth ACM Symp. on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar77] J. M. Barth. An interprocedural data flow analysis algorithm. *Conf. Rec. Fourth ACM Symp. on Principles of Programming Languages*, pages 119–131, January 1977.

[BJ78] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis (two parts). *Acta Informatica*, 10:245–272, 1978.

[Bur87] M. Burke. An interval-based approach to exhaustive and incremental data flow analysis. Technical Report 12702, IBM, Yorktown Hts., New York, September 1987.

[CCH+88] D. Callahan, K. D. Cooper, R. T. Hoood, K. W. Kennedy, and L. M. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputing Applications*, 2(4), 1988.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 274–284, January 1988.

[FO76] L. D. Fosdick and L. D. Osterweil. Data flow analysis in software reliability. *Computing Surveys*, 8(3):305–330, September 1976.

[GW76] S. L. Graham and M. N. Wegman. A fast and usually linear algorithm for global flow analysis. *J. ACM*, 23(1):172–202, January 1976.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HU75] M. S. Hecht and J. D. Ullman. A simple algorithm for global data flow analysis problems. *SIAM J. Computing*, 4(4):519–532, December 1975.

[Kil73] G. A. Kildall. A unified approach to global program optimization. *Conf. Rec. First ACM Symp. on Principles of Programming Languages*, pages 194–206, October 1973.

[Knu71] D. E. Knuth. An empirical study of Fortran programs. *Software Practice and Experience*, 1(12):105–134, 1971.

[Kou77] L. T. Kou. On live-dead analysis for global data flow problems. *J. ACM*, 24(3):473–483, July 1977.

[KU76] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, January 1976.

[Mas80] L. M. Masinter. Global program analysis in an interactive environment. Technical Report SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, Ca. 94304, January 1980.

[RC87] B. G. Ryder and M. C. Carroll. An incremental algorithm for sortware analysis. *Proc. Second ACM SIGSOFT/SIGPLAN Engineering Symp. on Practical Software Development Environments*, 22(1):171,179, January 1987.

[Rep84] T. W. Reps. *Generating Language-Based Environments.* The MIT Press, Cambridge, Mass., 1984.

[RMP87] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Incremental iteration: When will it work? Technical Report LCSR-TR-89, Rutgers U., March 1987.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[Tar81] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.

[Zad83] F. K. Zadeck. *Incremental Data Flow Analysis in a Structure Program Editor.* PhD thesis, Dept. of Mathematical Sciences, Rice U., October 1983.

[Zad84] F. K. Zadeck. Incremental data flow analysis in a structure program editor. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, 19(6):132–143, June 1984.