An Incremental Version of Iterative Data Flow Analysis

LORI L. POLLOCK, MEMBER, IEEE, AND MARY LOU SOFFA

***Abstract*—Information about the flow of data in a program is required in a number of applications including code optimization and parallelization, instruction scheduling, and data flow testing. Program edits and transformations applied to program code mandate updating the data flow information to reflect the changes. The update of data flow information can be performed either by a complete reanalysis or by reusing previous data flow information through an incremental approach. We present a technique to incrementally update solutions to both union and intersection data flow problems in response to program edits and transformations. For generality, the technique is based on the iterative approach to computing data flow information. We show that for both union and intersection problems, some changes can be incrementally incorporated immediately into the data flow sets while others are handled by a two phase approach. The first phase updates the data flow sets to overestimate the effect of the program change enabling the second phase to incrementally update the affected data flow sets to reflect the actual program change. An important problem that is addressed in this paper is the computation of the data flow changes that need to be propagated throughout a program, based on different local code changes. Finally, our technique is compared to other approaches to incremental data flow analysis.**

***Index Terms*—Code optimizations, data flow analysis, incremental method, iterative computation, programming environments.**

I. Introduction

GIVEN a control flow structure, *data flow analysis* is the process of collecting information about the flow of data throughout the corresponding code segment [1], [12]. Information about the flow of data in a procedure has traditionally been used in the last two phases of the compiling process, namely intraprocedural optimization and target code generation. Currently, the use of data flow information extends well beyond these boundaries. It is also being used in editing, symbolic debugging, interprocedural optimization, parallelization, instruction scheduling, the generation of assertions for program verification, and data flow testing. In all of these applications, edits and transformations applied to program code require the update of data flow information to reflect the changed code.

Manuscript received July 31, 1987; revised June 29, 1989. Recommended by J. E. Urban. This work was supported in part by the National Science Foundation under Grants CCR-8618699 and CCR-8716914 to Rice University and RII-886298 and CCR-8801104 to the University of Pittsburgh.

L. L. Pollock is in the Department of Computer Science, Rice University, Houston, TX 17251.

M. L. Soffa is in the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

IEEE Log Number 8931152.

When data flow information is used in a compiler to detect safe conditions for optimizing transformations, the information must be made consistent with the transformed code to enable subsequent passes of safe optimization. Similarly, data flow updates are required between transformations that are applied during instruction scheduling for parallel machines [5], [8]. If data flow information is used by a language-based editor or symbolic debugger to report current static semantic errors, the information must be updated in response to each source program edit. When interprocedural optimization is performed across separately compiled modules, recompilation analysis is based on edits and resulting changes to interprocedural data flow [4]. In finer grain recompilation with intraprocedural optimization, the effects of a local edit, interprocedural change or optimization change on the safety of optimizations are detected using changes in data flow information [13].

Data flow information can be updated either by repeating the original, exhaustive analysis after a change or by using incremental techniques that attempt to reuse some of the analysis and recompute only the information affected by the change. Presuming that a change typically has a localized effect, or at least does not affect all of the program's data flow, incremental algorithms that efficiently update only the information affected by the change are desired. Recognizing the need for incremental techniques that are applicable to a wide range of applications over both structured and unstructured programs, we present algorithms in this paper to incrementally compute data flow information based on the iterative approach [1]. The techniques use a flow graph representation of a program and handle both structural (flow graph edge or basic block) and nonstructural (variable use or definition) changes. The algorithms include the computation of the set of data flow changes to be propagated and the propagation of the data flow changes throughout a program. The incremental techniques are independent of the precision of the information; that is, issues involving procedure calls, pointer variables, and array variable assignments are not addressed in this work because conservative approaches can be used when more precise information is unavailable.

Section II gives background on data flow analysis, while Section III describes the different scenarios of incremental update and presents the incremental data flow analysis algorithms. The approach in this paper is compared in Section IV to other incremental data flow techniques.

II. Data Flow Problems

Global or *intraprocedural* data flow problems demand a solution within a single procedure, while *interprocedural* data flow problems are solved over a set of procedures composing a whole program. Although much of this work is applicable to both global and interprocedural problems, this paper focuses on global data flow problems including reaching definitions, reachable uses, and available expressions. The reaching definitions problem consists of determining the set of variable definitions that reach various points in a procedure. This information is useful for detecting opportunities for constant folding and variables that are used without previously being defined. Similarly, the reachable uses problem, which arises in copy and constant propagation and also discerns information useful for register allocation and dead code elimination, consists of finding the variable uses that are reachable from various points in a procedure. The removal of redundant expression evaluations requires information about the previous computation, or availability, of the expressions. The applicable data flow problem is available expression computation.

Intraprocedurally, data flow analysis typically consists of solving a system of data flow equations that relate information at the beginning and end of basic blocks of sequential code within a flow graph representation of a procedure [1]. The data flow equations consist of set or bit vector operations on a basic block's GEN, KILL, IN, and OUT sets. Each block's GEN set contains information about data generated within the block, and the KILL set contains information about data that is stopped from flowing upon executing the block. The IN set summarizes the flow of data at the start of the block, while the OUT set summarizes the flow of data on exit from the block. Global data flow analysis is the process of computing the IN and OUT sets for each basic block after a simple textual scan to compute the GEN and KILL sets.

The operation applied at each block to compute the data flow information entering the block from joining paths is called the *confluence* operator. Most data flow problems fall into two classes based on this operator. Problems that are solved by *set intersection* require that certain conditions be true along *all* possible paths of execution that reach a program point p in order to include elements in the data flow set at p. Problems involving *set union* require that the information be true along *at least one* path that reaches p. Both reaching definitions and reachable uses are set union problems, in that a variable definition is said to reach program point p if it reaches along *any* path to p. Available expressions is an example of a set intersection problem. An evaluation of an expression must occur along *all* possible paths of execution to point p in order to be considered available at p.

Data flow problems are further classified as either *forward* or *backward* according to the direction that information is propagated. In forward data flow problems, including reaching definitions and available expressions, data flow sets are computed in terms of data flowing into a basic block or what can occur *before* control reaches the block. In contrast, backward data flow problems, such as reachable uses, require that sets be computed according to what can occur *after* control leaves a basic block or how the data at a block flows to other blocks. To illustrate the basic notions of data flow problems, data flow equations for three data flow problems are given below.

Reaching definitions (forward and union) where DefIN is the set of definitions that reach the start of basic block B, and DefOUT is the set of definitions that reach the end of B:

$$\text{DefIN}(B) = \cup\ \text{DefOUT}(P),\ P \text{ is predecessor of } B$$
$$\text{DefOUT}(B) = \text{DefGEN}(B) \cup (\text{DefIN}(B) = -\ \text{DefKILL}(B))$$

Reachable uses (backward and union) where UseIN is the set of uses reachable from the start of basic block B, and UseOUT is the set of uses that are reachable from the end of B:

$$\text{UseOUT}(B) = \cup\ \text{UseIN}(S),\ S \text{ is successor of } B$$
$$\text{UseIN}(B) = \text{UseGEN}(B) \cup (\text{UseOUT}(B) - \text{UseKILL}(B))$$

Available expressions (forward and intersection) where AvailIN is the set of expressions available at the start of basic block B, and AvailOUT is the set of expressions available at the end of B:

$$\text{AvailIN}(B_0) = 0 \text{ where } B_0 = \text{initial block}$$
$$\text{AvailIN}(B) = \cap\ \text{AvailOUT}(P),\ P \text{ is predecessor of } B,\ B \langle\rangle \text{ initial block}$$
$$\text{AvailOUT}(B) = \text{ExprGEN}(B) \cup (\text{AvailIN}(B) - \text{ExprKILL}(B))$$

One approach to computing global data flow information is to exploit the hierarchical structure of the program in an attempt to reduce the number of set operations. Structured techniques include Allen/Cocke interval analysis [2], Hecht/Ullmann T1-T2 analysis [18], Graham/Wegman path compression [7], and Tarjan interval analysis [17]. Although these techniques may prove to be relatively fast on reducible flow graphs, they typically require expensive transformations to handle nonreducible flow graphs. The simplest approach to data flow analysis, and the one that handles general flow graphs without special consideration, is the iterative technique [1]. The algorithm iterates through the blocks of the control flow graph applying the equations that compute the confluence of flow entering the blocks and local data effects until the data flow sets at each block have stabilized.

III. Incremental Data Flow Analysis

User edits, changes in interprocedural data flow information, and compiler optimizing transformations all initiate data flow changes. A program's data flow can be affected by changes that involve insertion or deletion of variable uses, variable definitions, and flow graph edges. The region to which the effects of a code change can extend is typically called the *affected region*. Advocates of incremental techniques believe that, in practice, this region rarely encompasses the whole procedure, and is frequently limited to a small part of the program.

The incremental update of data flow information, commonly called *incremental data flow analysis*, can be formulated as the problem:

> Given a program and a correct solution to a data flow problem over that program, update the affected parts of the current solution to reflect a change in the program without unnecessary reinitialization and recalculation of the entire data flow solution.

When a program change occurs within a basic block B_c, our incremental data flow analysis begins by updating the data flow information local to B_c, namely GEN(B_c) and KILL(B_c). The set of data flow changes that potentially affect the data flow at additional blocks, *LocalSet*, is computed from the changes in GEN(B_c) and KILL(B_c). For example, a potential increase in data flowing into other blocks can be caused by an increase in GEN(B_c) or a decrease in KILL(B_c). The computation of *LocalSet* under different editing scenarios is described in a later section.

The global effects of these local changes are determined by propagating data flow changes to all of the blocks in the affected region starting with *LocalSet* at block B_c. The method for propagating global data flow changes from B_c depends on whether the data flow problem is a forward or backward, union or intersection problem, and whether the change causes the flow of data items to increase to additional blocks, decrease (no longer propagate to some blocks), or increase for some items while decreasing for others.

Analogous to the exhaustive methods, the effects of a program change on the solution to a forward data flow problem propagate to the *successors* of the changed block, while effects on backward flow information propagate to the *predecessors*. Since the incremental algorithms for forward and backward problems differ only in the direction of change propagation, we present our algorithms for forward problems without loss of generality.

Based on whether the problem is union or intersection and whether the flow of data items is increasing or decreasing, we found that one of two distinct incremental updating scenarios can apply. Given a code change at block B_c, the data flow set at a block B_n can be either

1) immediately updated when B_n is visited along *any* path from B_c to B_n during change propagation (*any path scenario*), or

2) updated while traversing a path from B_c to B_n during change propagation only after ensuring that certain conditions are satisfied on *all* incoming paths to B_n (*all paths scenario*).

For example, when a definition of v is inserted into block B_c, the new definition can be added immediately to the reaching definition sets at all successors of B_c that are visited along each path from B_c before a redefinition of v because the new definition is ensured to reach these blocks (*any path* scenario). Since the inserted definition also blocks the flow of other definitions of v through B_c, the incremental analysis also must delete the blocked definitions from the sets at all successors of B_c where they no longer reach. Definitions of v that occur in B_c prior to the point of insertion can be deleted immediately from the successors' sets because they are unquestionably blocked by the new definition (*any path* scenario). However, definitions of v that occur in other blocks and previously reached through B_c cannot be deleted automatically from the set at a block B_n without first ensuring that they do not reach B_n through an alternate path that does not include B_c (*all paths* scenario). In general, the *any path* scenario applies to a union problem when flow is increasing or flow is unquestionably decreasing, and the *all paths* scenario applies when flow is potentially decreasing but alternate paths may sustain the flow.

The incremental update scenario for intersection problems is a reflection of the scenario for union problems. For example, an expression can be deleted immediately from the available expression set at a block B_n when an inserted definition or deleted expression evaluation prevents an evaluation of the expression from reaching B_n along at least one path (*any path* scenario). When a change such as inserting an expression or deleting a definition extends the reachability of a particular expression evaluation to a block B_n, the expression cannot be added automatically to the available expressions set at B_n without first ensuring that the expression is now available at all immediate predecessors of B_n (*all paths* scenario). In general, the *any path* scenario applies to an intersection problem when flow is decreasing, while the *all paths* scenario applies when flow is potentially increasing.

The *any path* and *all paths* categorization of data flow changes in union and intersection problems is presented below along with the edits that could cause these changes in reaching definitions and available expressions. A single edit may involve both scenarios to handle the total effect of the edit. When this occurs, we indicate the aspect of the edit being handled by the *any path* or *all paths* update.

The *any path* update scenario applies when program change results in the following situations.

- *Union Problem and Flow Increasing:*
  (Solution is getting larger.)
  Causes: insert a flow graph edge, insert a definition (propagate the new definition), delete a definition (propagate previously blocked definitions of the same variable).
- *Union Problem and Flow Decreasing:*
  (Solution is getting smaller.)
  Causes: delete a definition (propagate removal of the definition), insert a definition (propagate removal of local definitions of the same variable blocked by the new definition).
- *Intersection Problem and Flow Decreasing:*
  (Solution is getting smaller.)

TABLE I
INCREMENTAL UPDATES TO REACHING DEFINITIONS

Program Change	Action
Insert a use ***u*** of variable *v* in block B at statement *s*	No global effect on reaching definitions.
Delete a use ***u*** of variable *v* in block B at statement *s*	No global effect on reaching definitions.
Insert a definition *d* of variable *v* in block B at statement *s* (increase DefGEN) (increase DefKILL)	*if there is a definition of v in B after s*: No global effect. *else if there is no definition of v in B after s*: (1) Add *d* to reaching definitions of all reachable successors of B. (any path) (2) *if there is a definition of v in B prior to s*: Let *dp* = most recent definition of *v* in B prior to *s*. Delete *dp* from reaching definitions of all reachable successors of B. (any path) *else if no definition of v in B prior to s*: Delete all definitions of *v* in DefIN(B) from reaching definitions of all reachable successors of B where they no longer reach by any path. (all paths)
Delete a definition *d* of variable *v* in block B at statement *s* (decrease DefGEN) (decrease DefKILL)	*if there is a definition of v in B after s*: No global effect. *else if there is no definition of v in B after s*: (1) Delete *d* from reaching definitions of all reachable successors of B. (any path) (2) *if there is a definition of v in B prior to s*: Let *dp* = most recent definition of *v* in B prior to *s*. Add *dp* to reaching definitions of all reachable successors of B. (any path) *else if no definition of v in B prior to s*: Add all definitions of *v* in DefIN(B) to reaching definitions of all reachable successors of B. (any path)
Insert a flow graph edge from block B_s to block B_t	Add all definitions (of any variable) in DefOUT(B_s) to reaching definitions of B_t and all reachable successors of B_t. (any path)
Delete a flow graph edge from block B_s to block B_t	Delete all definitions (of any variable) in DefOUT(B_s) from reaching definitions of B_t and all reachable successors of B_t where they no longer reach by any path. (all paths)

Causes: delete an expression, insert a definition, insert a flow graph edge (create the only path to a block without the otherwise intersecting items), delete a flow graph edge (delete the only path connecting two segments such that all flow to the target segment is decreased).

The *all paths* update scenario applies when program change results in the following situations.

- *Union Problem and Flow Potentially Decreasing:* (Solution may be getting smaller.)
  Causes: delete a flow graph edge, insert a definition (propagate removal of remote definitions of the same variable blocked by the new definition).
- *Intersection Problem and Flow Potentially Increasing:* (Solution may be getting larger.)
  Causes: insert an expression, delete a definition, delete a flow graph edge (destroy the only path to a block without the otherwise intersecting items), insert a flow graph edge (create the only path connecting two segments such that new data flows into the target segment).

Tables I and II give a more detailed picture of the required updates to reaching definitions and available expressions in response to program changes.

While nonstructural changes in a block B_c occur as changes in GEN(B_c) and KILL(B_c), edge changes sever and create new control flow paths along which data can flow. The increased or decreased flow from the source to the target block of an edge change can be handled analogously to the increase or decrease in flow from local block changes. However, an edge deletion can actually cause part of the control flow graph to become unreachable from other segments of the program if the deleted edge was the

TABLE II
INCREMENTAL UPDATES TO AVAILABLE EXPRESSIONS

Program Change	Action
Insert an expression *e* in block B at statement *s* (increase ExprGEN)	*if e is not already available at the end of B* *and if there is no redefinition of the operands of e after s in B:* Add *e* to available expressions of all successors of B where *e* at *s* reaches and *e* is now available from all paths. (all paths)
Delete an expression *e* in block B at statement *s* (decrease ExprGEN)	*if e was previously available at the end of B and if there are no other evaluations of e in B or in AvailIN(B) reaching the end of B:* Delete *e* from available expressions of all successors of B where *e* at *s* previously reached and *e* is no longer available. (any path)
Insert a definition *d* of variable *v* in block B at statement *s* (increase ExprKILL)	*For each expression e using v and previously available at the end of B*: Let *sp* = the location of *e*. *if there are no evaluations of e after s reaching end of B:* Delete *e* from available expressions of all successors of B where *e* at *sp* previously reached and *e* is no longer available. (any path)
Delete a definition *d* of variable *v* in block B at statement *s* (decrease ExprKILL)	*For each expression e using v and now available at s*: Let *sp* = the location of *e*. *if e is not already available at the end of B* *and if there is no redefinition of the operands of e after s in B:* Add *e* to available expressions of all successors of B where *e* at *sp* now reaches and *e* is now available from all paths. (all paths)
Insert a flow graph edge from block B_s to block B_t	*If not adding a code segment*: Delete expressions from available expressions of B_t and any successors of B_t where there now exists at least one path from which the expressions are not available. (any path) *else if adding a new code segment*: Perform exhaustive analysis on new segment in the context of the new edge.
Delete a flow graph edge from block B_s to block B_t	*If not deleting a code segment*: Add expressions to available expressions of B_t and any successors of B_t where they are now available from all paths by eliminating the only path where they were not available. (all paths) *else if deleting a code segment*: No data flow update.

only edge connecting the regions. Similarly, edge insertions can cause previously unreachable segments to become reachable from other segments.

We choose not to update the data flow sets of a code segment when the segment becomes unreachable. We assume that upon inserting an edge that adds an unreachable segment to the current version of a procedure, the data flow sets of the unreachable segment are calculated exhaustively, taking into account the information flowing through the new edge. Thus, incremental analysis is not unnecessarily performed on code that is discarded during editing or transformation. Also, as we believe that exhaustive analysis should be used for initial program entry when code is drastically changing, our treatment of unreachable segments allows code segments that are incorporated later, but temporarily unreachable, when they are created, to exploit exhaustive analysis. Tables I and II account for unreachable segments based on our assumptions.

The next sections describe and analyze the **IMMEDIATE** and **TWO_PHASE** algorithms which we have developed to correctly update data flow information in response to program changes under the *any path* and *all paths* update scenarios, respectively. In order to reduce redundant traversals, code changes that cause both deletions and additions to data flow sets in the same direction of propagation could be handled by integrating the update process. For example, when deleting a variable definition, the same reaching definition sets are being altered by addition and deletion such that both operations could be performed on the same update traversal.

A. *The* IMMEDIATE *Update Algorithm*

The simplest changes to handle are those that can be incorporated under the *any path* scheme. Starting at the block B_c containing the program change, update consists of immediately updating the data flow sets without recomputing the confluence of flow at each block along paths extending from B_c. The set of data flow changes being propagated, *ChangePropSet*, is initialized to *LocalSet*. Throughout the change propagation process, *ChangePropSet* is updated to reflect the local GEN and KILL effects of blocks along the current path of updates from B_c.

When the flow of only a single item (e.g., variable or expression) is affected by a program change, *ChangePropSet* contains data flow changes for that item only, and either contains the changes as initialized or is empty. For example, when a variable definition is deleted and reaching definitions are being considered, the flow of other definitions of the same variable will be increased, and these definitions will comprise *ChangePropSet*. A redefinition of the same variable will block the increased flow of all of these definitions. *ChangePropSet* then becomes empty such that no further traversal along a path with the redefinition is necessary. However, when a program change, such as inserting a flow graph edge or changing a variable definition when computing available expressions, affects the flow of multiple items, *ChangePropSet* may contain various items at different points during change propagation. If a redefinition of one item in the set does not affect the flow of the other items through that block, all instances of the redefined item will be deleted from *ChangePropSet*, but instances of other items will remain in the set.

Traversal along a path halts when the data flow set at a block is no longer affected (i.e., has stabilized). When adding elements to reaching definition sets, traversal along a path halts when either the definition has already been added, the definition was in the set prior to the change, or reachability of the definition is stopped by a redefinition in the block. Traversal of a path to delete an expression is halted when either the expression has already been deleted from the available expression set, the expression was never in the set, or another evaluation of the expression is encountered. The **IMMEDIATE** algorithm terminates when sets along all paths originating at the program change have stabilized.

Fig. 1 contains the forward data flow formulation of algorithm **IMMEDIATE** that calls procedures **ADDFLOW** and **DELETEFLOW.** The recursive procedure **DELETEFLOW** is included in Fig. 1. Given the data flow problem being updated (*DataSet*), the initial set of data flow changes that need to be propagated (*LocalSet*), an increase or decrease of flow (*change*), a structural or nonstructural change (*edgechange*), the target of the edge change or node containing the nonstructural change (*startnode*), and single or multiple item update (*multitem*), **IMMEDIATE** initiates the change propagation. When a decrease in flow is indicated, procedure **DELETEFLOW** is recursively called at each node requiring data flow changes to update the data flow sets at that node, update *ChangePropSet*, and guide the traversal to subsequent nodes along the current change propagation path. The procedure **ADDFLOW** for immediate update to reflect an increase in the flow of data follows analogously to the **DELETEFLOW** procedure.

Since nonstructural changes can only affect the reaching definitions of a single variable, an affected node should be visited only once to correctly update the reaching definitions information at that node. The set *visited* ensures that affected nodes are visited only once in response to nonstructural changes. In contrast, edge changes can affect the flow of several different variables. At a node along the update traversal, the data flow set is updated for all of the variable definitions in *LocalSet* reaching the node from the current path of updates to avoid a separate update traversal for each variable. Some variable definitions may

```
ALGORITHM IMMEDIATE. Incrementally update global data flow under any path scenario.

INPUT:  DataSet:    type of data flow sets being updated (DefIN/DefOUT, AvailIN/AvailOUT,...).
        LocalSet:   set of local data flow changes to be propagated globally.
        change:     type of operation on sets (addflow, deleteflow).
        edgechange: boolean value - true if change is edge, else false (i.e., nonstructural change).
        startnode:  target of edge change or node containing nonstructural change.
        multitem:   boolean value - true if multiple items affected.

OUTPUT: Updated global data flow sets of DataSet type.
DECLARE visited:    nodes visited thus far.

BEGIN {IMMEDIATE}
visited := {};
IF edgechange THEN
 IF change = addflow THEN ADDFLOW(LocalSet,startnode,DataSet,visited,multitem)
   ELSE DELETEFLOW(LocalSet,startnode,DataSet,visited,multitem);
ELSE BEGIN   {variable change}
  Update DataSet_OUT(startnode) to reflect LocalSet;
  FOR EACH immediate successor s of startnode DO
     IF change = addflow THEN ADDFLOW(LocalSet,s,DataSet,visited,multitem)
        ELSE DELETEFLOW(LocalSet,s,DataSet,visited,multitem);
  END; {variable change}
END.  {IMMEDIATE}
```

```
RECURSIVE PROCEDURE DELETEFLOW(ChangePropSet,node,DataSet,visited,multitem);
  ChangePropSet:     current change propagation set.
  node:              node  currently being traversed.
  DataSet:           type of data flow sets being updated (DefIN/DefOUT, AvailIN/AvailOUT,...).
  visited:           nodes visited thus far.
  multitem:          boolean value - true if multiple items affected.

DECLARE NewChPropSet: Newly computed current change propagation set.

BEGIN  {delete elements from data flow set at this node and continue}
NewChPropSet := ChangePropSet ∩ DataSet_IN(node);
IF (NewChPropSet is not empty) THEN BEGIN  {not fully updated}
  visited := visited ∪ {node};
  DataSet_IN(node) := DataSet_IN(node) - NewChPropSet;  {update set entering node}

  NewChPropSet := (NewChPropSet - DataSet_GEN(node)) ∩ DataSet_OUT(node);
  DataSet_OUT(node) := DataSet_OUT(node) - NewChPropSet; {update set exiting node}

  IF (NewChPropSet is not empty) THEN {continue traversal}
     FOR EACH immediate successor s of node DO

        IF (s not in visited) OR (multitem) THEN
           DELETEFLOW(NewChPropSet,s,DataSet,visited,multitem);
  ENDIF;  {not fully updated}
END. {DELETEFLOW}
```

Fig. 1. Algorithm **IMMEDIATE** and procedure **DELETEFLOW**.

reach an affected node by different paths such that the node is visited and updated from different paths. In particular, the **DELETEFLOW** procedure is called for each affected node at most v times where v is the number of different variables with flow possibly affected by the change (i.e., within *LocalSet*). On each call, **DELETEFLOW** performs a constant amount of work to update the data flow sets at one node. Therefore, algorithm **IMMEDIATE** applied to reaching definitions has a worst case time bound of $O(a*v)$ where a is the number of nodes affected by the change. Using the policy that as many updates as possible are performed at a node on each update traversal, it is expected that the time would be $O(a*x)$ for some $x << v$.

The update of available expressions can involve different expressions in response to any change except inserting or deleting an expression evaluation, because there may be several expressions involving the same variable. The affected nodes could be updated at most e times where e is the number of different expressions with availability possibly altered by the change (i.e, within *LocalSet*), yielding a worst case time bound of $O(a*e)$ where a is the number of nodes in the affected region. The expected time is $O(a*x)$ for $x << e$.

B. The TWO-PHASE *Update Algorithm*

The *all paths* scenario of incremental data flow update is more complex to handle because the incremental analysis must ensure that certain conditions are satisfied along *all* paths to a block in order to correctly update its data flow sets. A naive approach to the *all paths* update might traverse the paths extending from the immediate successors of the changed block and update the data flow sets by recomputing the data flow equations. Unlike the **IMMEDIATE** algorithm that automatically adds or deletes elements from data flow sets as it traverses blocks, the *all paths* update would compute the confluence of flow entering a block B_n based on the data flow sets at the predecessors of B_n.

Although it may appear that computing the confluence would account for all paths entering a block, it fails when the affected area contains a strongly connected region, which could correspond to a natural loop, looplike structure, or nest of these structures. In this situation, at least one predecessor of a block B_n in the strongly connected region has not been fully updated when its data flow set is being used in the computation of the new data flow set at B_n. Recomputing the original data flow equations in any traversal order through the strongly connected region will not yield the desired new solution, because the strongly connected region acts as a preserving mechanism for old information from the original solution [16].

Consider a definition d that reaches all of the blocks in a strongly connected region R, but does not reach these blocks after a program change. The definition d is originally in the data flow sets of all of the blocks in R, and incremental analysis should delete d from all of these sets. Confluence of flow is recomputed when entering the first block B_n selected from R. At least one of B_n's predecessors will also be in R and not yet have d removed from its set. This predecessor will cause the result of the union at B_n to include d in B_n's data flow set and in the sets at all of the successors of B_n before a redefinition of the same variable. This encompasses the entire strongly connected region R. In general, a strongly connected region causes at least one predecessor set used in a union computation to be larger than the desired new solution, and applying the union operator to sets that represent a solution larger than the desired new solution will never yield the smaller, desired solution.

Similarly, an expression e that was unavailable throughout a strongly connected region before a program change and becomes available after the change will be inappropriately represented as unavailable throughout the strongly connected region after the incremental update. At least one of the predecessors of each block in the strongly connected region will also be in the region and not contain e (i.e., reflecting the old smaller solution). The smaller sets from the old solution will prevent the intersection of the predecessors from creating the larger desired set.

We present a two phase solution to this problem. The first phase, the *exaggerate* phase, overestimates the effects of the program change on data flow sets, obtaining the solution that represents the largest possible effects of *LocalSet*. The second phase, the *adjust* phase, then adjusts the exaggerated data flow solution to reflect the actual effects of the program change by recomputing data flow equations similar to the exhaustive, iterative algorithm at blocks where data flow sets were altered by the *exaggerate* phase. Algorithm **TWO_PHASE** is presented in Fig. 2. Fig. 3 contains the **ADJUST** procedure called recursively during the second phase.

Since the *all paths* scenario applies to a union problem when flow is potentially decreasing, and the new solution should be smaller than the original solution, the *exaggerate* phase obtains the smallest possible new data flow solution with respect to the changed data. This phase corresponds to initializing the data flow sets of union problems to empty (i.e., the smallest possible solution) in the exhaustive version of iterative analysis. In our incremental analysis, the smallest possible new solution is obtained by deleting only elements with potentially decreased flow from the data flow sets at all blocks to which they had previously propagated from the program change. This consists of executing the **IMMEDIATE** algorithm with recursive calls to **DELETEFLOW.** This solution may not be the final solution because flow to some of these blocks may not actually be decreased due to alternate paths around the program change. For example, the *exaggerate* phase for updating a reaching definitions solution in response to deleting an edge from B_s to B_t assumes that there is no other path along which the definitions reaching the end of B_s can now flow to B_t and its successors; these definitions are deleted from all blocks previously reachable through that edge, obtaining the smallest possible new solution with respect to these variables. However, the second phase may find that some of these definitions reach these blocks through other paths.

The *exaggerate* phase overestimates the effect of a program change on an intersection problem by finding the largest possible new solution with respect to the potentially affected data items. This solution is obtained by adding elements with potentially increased flow to the data flow sets at all blocks to which they could possibly propagate, starting at the program change. This update is performed by executing the **IMMEDIATE** algorithm with recursive calls to **ADDFLOW.** The resulting sets will be greater than or equal to the final solution and correspond to the initialization of sets in exhaustive analysis of intersection problems to the universal set (i.e., the largest possible solution). For example, when deleting the only definition of variable v from block B_c, all expressions involving v and able to reach the end of B_c are assumed to become available leaving B_c. The *exaggerate* phase adds these edges to the data flow sets of all blocks to which they can reach through B_c after the program change. The final updated solution may not include all of these expressions as available at these blocks.

The second phase of incremental update, *adjust*, visits the blocks with data flow sets altered during the *exaggerate* phase and recomputes data flow equations using the updated data flow sets, similar to the exhaustive iterative algorithm, but focusing on the region of the program

```
ALGORITHM TWO_PHASE. Incrementally update global data flow under all paths scenario.

INPUT:   DataSet:    type of data flow sets being updated (DefIN/DefOUT, AvailIN/AvailOUT,...).
         LocalSet:   set of local data flow changes to be propagated globally.
         edgechange: boolean value - true if change is edge, else false (i.e., nonstructural change).
         startnode:  target of edge change or node containing nonstructural change.
         multitem:   boolean value - true if multiple items affected.

OUTPUT: Updated global data flow sets of DataSet type.

DECLARE change:     type of operation on sets (addflow, deleteflow).
        visited:    nodes with sets updated during exaggerate phase.

BEGIN {TWO_PHASE}

{exaggerate phase - assume largest possible effect of LocalSet change}
IF (DataSet is of type intersection) THEN change := addflow {∩ - largest possible new solution}
  ELSE change := deleteflow;   {∪ - smallest possible new solution}
Run IMMEDIATE algorithm, returning visited := {nodes with sets updated during exaggerate phase};

{adjust phase - adjust current data flow sets of visited to reflect actual change}

  FOR EACH node n in visited DO
   ADJUST(n,DataSet);

END. {TWO_PHASE}
```

Fig. 2. Algorithm **TWO_PHASE**.

```
RECURSIVE PROCEDURE ADJUST(node,DataSet);
   node:          node currently being traversed.
   DataSet:       type of data flow sets being updated (DefIN/DefOUT, AvailIN/AvailOUT,...).

DECLARE NewSet: Newly computed data flow set.

BEGIN   {recompute data flow sets at this node and continue}

   {recompute confluence}
   IF (DataSet is of type union) THEN
     NewSet := ∪ DataSet_OUT(p) for all predecessors p of node;
    ELSE IF (DataSet is of type intersection) THEN
     NewSet := ∩ DataSet_OUT(p) for all predecessors p of node;

   IF NewSet <> DataSet_IN(node) THEN BEGIN  {DataSet_IN changed}
     DataSet_IN(node) := NewSet;
     NewSet := (DataSet_IN(node) - DataSet_KILL(node)) ∪ DataSet_GEN(node); {local effects}

     IF NewSet <> DataSet_OUT(node) THEN BEGIN {DataSet_OUT changed}
      DataSet_OUT(node) := NewSet;
      FOR EACH immediate successor s of node DO    {continue traversal}
       ADJUST(s,DataSet);
       END; {DataSet_OUT changed}
    END; {DataSet_IN changed}
END. {ADJUST}
```

Fig. 3. Procedure **ADJUST**.

where data flow sets were altered in the first phase. Applying the confluence operator over all predecessors of a block implicitly determines whether the program change has in fact altered the data flow facts on all paths to the block.

In the reaching definitions problem, the *adjust* phase inserts definitions back into sets at blocks where they are still able to reach from at least one path after the change. The insertion is not explicit, but rather implicit in the union of the exaggerated sets. Elements are deleted from the IN set at a block B_n during the *adjust* phase for an intersection problem when the set on exit from any of the predecessors of B_n does not contain that element, indicating that the program change has not actually caused this flow information to be true along all paths to B_n. Again, the deletion is not explicit, but rather implicit in performing the intersection of the exaggerated sets. The *adjust* phase terminates when the data flow sets have stabilized, similar to the exhaustive algorithm.

Since the *exaggerate* phase of the **TWO_PHASE** algorithm runs the **IMMEDIATE** algorithm with *LocalSet*, it has a worst case running time of $O(n*v)$ for problems involving variables and $O(n*e)$ for problems with expression set elements, where n is the number of blocks to which these changes would flow in the *any path* scenario, v is the number of distinct variables referenced in *LocalSet*, and e is the number of distinct expressions appearing in *LocalSet*.

When a variable definition is inserted, the *exaggerate* phase traverses exactly those blocks that may no longer be reached by definitions that previously reached the insertion point. Since definitions that appear to be blocked by the new definition may actually reach some of these blocks through other paths, the traversed blocks may include some blocks where definitions are not ultimately eliminated. However, since the new definition is effectively replacing the reachability of the blocked definitions, the blocks being traversed are exactly those blocks at which the new definition is being added to the data flow set. Thus, in this case, n is exactly a where a is the number of blocks affected by the program change. Similarly, n is also equal to a when a definition is deleted.

When an edge is deleted and reaching definitions are being updated, the blocks traversed by the *exaggerate* phase may include blocks not affected in the final solution since the edge change does not involve replacement of decreased flow of some definitions by an increased flow of other definitions. Thus, n may be greater than a in this case. In an intersection problem, the *exaggerate* phase traverses all blocks to which the *any path* solution would add elements. For available expressions, this may include blocks where the expressions would not be included in the final solution (i.e., n may be greater than a), but will include only blocks where the availability of expressions in *LocalSet* could potentially increase.

The *adjust* phase traverses the same blocks (called *visited* in the algorithm of Fig. 2 and shown above to be of size n) traversed during the *exaggerate* phase, possibly a different number of times. The number of blocks visited during the *adjust* phase is no more than $O(|visited|^2)$ or $O(n^2)$ because each block in *visited* with unique data flow changes could, in the worst case, force a repeated traversal of all of the blocks in *visited* to propagate its inconsistencies. However, this situation is unlikely as it implies that 1) the elements of *LocalSet* that need to be propagated all originate at different blocks in *visited* or at least one unique element from *LocalSet* originates from each block in *visited*, 2) the path structure within *visited* is such that changes originating in different blocks are not processed during the propagation of changes from any other block in *visited* and thus require a separate traversal, and 3) each change propagation path visits every block in *visited*. Thus, the expected running time of *adjust* is much less. In cases where the *exaggerate* phase may visit some blocks not ultimately affected by the program change and include these blocks in the set *visited*, some blocks that are not ultimately affected by the program change may be visited during both phases. On each visit to a block during both phases, data flow sets are updated in a constant amount of time. Thus, the overall, worst case running time of **TWO_PHASE** is $O(n^2)$ where n is the number of blocks to which a change in *LocalSet* would flow in an *any path* scenario.

C. LocalSet and Change Propagation

In both *any path* and *all paths* update scenarios, *ChangePropSet* is initialized to contain the elements in *LocalSet*, the data flow changes local to the code change that may potentially have a larger effect on the program's data flow. Assuming that data flow sets are not incrementally updated within unreachable code segments, our discussion of *LocalSet* does not consider the possibility of the segment containing the change being unreachable. A nonstructural program change at a block B_c is handled as an increase or decrease in GEN(B_c) or KILL(B_c), while edge changes are handled in terms of increasing or decreasing the information flow from the source to the target block. Also, when a new reference to a data item (i.e., definition or use) is inserted into B_c, this data item may need to be included in the KILL sets at other blocks since a variable definition kills all other references to the same variable throughout the entire procedure. Similarly, when a data reference is deleted from the code, KILL sets of other blocks may also need to be updated to eliminate this reference. These changes to KILL sets of other blocks will have no effect on the global data flow and thus require no global data flow update.

Table III defines *LocalSet* and indicates the change propagation algorithm used to incrementally update union and intersection problems in response to each local change. We assume that the elements of a data flow set in a union problem are distinct occurrences of a data item such as specific definitions of a variable. Since the most common intersection problem in optimization is available expressions, we assume that the elements of a data flow set in an intersection problem are generic occurrences of a data item such as any evaluation of the expression $a + b$ rather than a specific evaluation of $a + b$ at statement s. The assumption of generic or nongeneric data items contributes to the difference in the *LocalSet* computation for intersection and union problems. This can be seen easily by considering the generation of a new item upon statement insertion. For the nongeneric case, this item would be placed in *LocalSet* regardless of whether an intersection or union problem is being solved because the uniqueness of the item ensures that it is not already being propagated. Under the generic assumption, the item would only be placed in *LocalSet* in either problem if it is not already propagated from another statement. The IncreaseGen entries in Table III illustrate the different *LocalSets* caused by the generic or nongeneric assumption.

The *LocalSet* computation for a union problem is straightforward, possibly involving a small constant number of bit vector operations given the changes in GEN(B_c) or KILL(B_c). When GEN(B_c) is changed corresponding to a local data item being inserted or deleted, *LocalSet* is exactly the set of elements added to or deleted from GEN(B_c) [i.e., addGEN(B_c) or deleteGEN(B_c)]. When a change in B_c causes KILL(B_c) to increase, possibly two different kinds of global update are needed, depending on where the items being added to KILL are generated. A *LocalSet* containing all data items previously generated in B_c and now killed within B_c, described by the set $\{x \mid x \in (\text{addKILL}(B) \cap \text{deleteGEN}(B))\}$, will be propagated by the **IMMEDIATE** algorithm. A *LocalSet* containing all other data items reaching B_c and killed by the change in B_c, represented by the set $\{x \mid x \in (\text{addKILL}(B) \cap \text{IN}(B))\}$ will be propagated by the **TWO_PHASE** algorithm. A decrease in KILL(B_c) involves propagating a *LocalSet* that contains the data items deleted from KILL(B_c) and either generated by B_c due to the decreased KILL or flowing into B_c prior to the change. This *LocalSet* involves computing the set expression $\{x \mid x \in (\text{deleteKILL}(B) \cap (\text{addGEN}(B) \cup \text{IN}(B)))\}$. The *LocalSet* for an edge change from B_s to B_t is the set of elements in OUT(B_s), since the edge change is either creating or removing all flow of these items from B_s to B_t.

When GEN(B_c) is increased for intersection problems, *LocalSet* is the set of newly generated data items, (addGEN(B_c)), which are not already in OUT(B_c) due to another existing occurrence of the same data item. When KILL(B_c) is increased, *LocalSet* consists of all data items that become killed by B_c and are either flowing into B_c prior to the change or no longer generated in B_c after the change. It should be noted that *LocalSet* for intersection problems in response to a change in GEN(B_c) or increase in KILL(B_c) is the generic equivalent to the nongeneric *LocalSet* computation for union problems. If the generic assumption for an intersection problem was changed to nongeneric, the *LocalSet* computations for both would be identical.

Upon inserting an edge from B_s to B_t that does not add a new code segment to the program, *LocalSet* for an intersection problem contains all items that appear in IN(B_t), but not in OUT(B_s), because these items no longer flow into B_t from all paths. Unlike other code changes, *LocalSet* may not be readily apparent from the local data flow sets in response to decreasing a KILL set or deleting a flow graph edge. The *LocalSet* computation is straightforward when the change does not occur within a strongly connected region. When KILL(B_c) is decreased and B_c is not in a strongly connected region, the set of elements in deleteKILL(B_c) and either generated in B_c after the change or flowing into B_c is used as *LocalSet* in running the **TWO_PHASE** algorithm, similar to union problems. When an edge $B_s \rightarrow B_t$ is deleted, and B_t is not in a strongly connected region, **TWO_PHASE** is called with the set of elements that appear in the OUT set of all of the remaining predecessors of B_t since there may be items that flow to B_t from all predecessors but B_s.

When KILL(B_c) is decreased or an edge into B_c is deleted and B_c is in a strongly connected region, both data flowing into the region and data generated within the region could begin to flow through B_c and along all paths to successors of B_c located both in and after the region. However, the original KILL or edge could have previ-

TABLE III
LocalSet Computation and Change Propagation

Local Change	LocalSet	Change Propagation
	FOR UNION PROBLEMS	
Increase GEN(B) with addGEN(B)	$\{x \mid x \in \text{addGEN}(B)\}$	IMMEDIATE add
Decrease GEN(B) with deleteGEN(B)	$\{x \mid x \in \text{deleteGEN}(B)\}$	IMMEDIATE delete
Increase KILL(B) with addKILL(B)	$\{x \mid x \in (\text{addKILL}(B) \cap \text{deleteGEN}(B))\}$ $\{x \mid x \in (\text{addKILL}(B) \cap \text{IN}(B))\}$	IMMEDIATE delete TWO_PHASE delete
Decrease KILL(B) with deleteKILL(B)	$\{x \mid x \in (\text{deleteKILL}(B) \cap (\text{addGEN}(B) \cup \text{IN}(B)))\}$	IMMEDIATE add
Insert Edge B_s->B_t	$\{x \mid x \in \text{OUT}(B_s)\}$	IMMEDIATE add
Delete Edge B_s->B_t	$\{x \mid x \in \text{OUT}(B_s)\}$	TWO_PHASE delete
	FOR INTERSECTION PROBLEMS	
Increase GEN(B) with addGEN(B)	$\{x \mid x \in (\text{addGEN}(B) - \text{OUT}(B))\}$	TWO_PHASE add
Decrease GEN(B) with addGEN(B)	$\{x \mid x \in (\text{deleteGEN}(B) - (\text{IN}(B) - \text{KILL}(B)))\}$	IMMEDIATE delete
Increase KILL(B) with addKILL(B)	$\{x \mid x \in (\text{addKILL}(B) \cap (\text{IN}(B) \cup \text{deleteGEN}(B)))\}$	IMMEDIATE delete

Decrease KILL(B) with deleteKILL(B)	*if* B *not in scr*:*
	$\{x \mid x \in (\text{deleteKILL}(B) \cap (\text{addGEN}(B) \cup \text{IN}(B)))\}$
	else if B *in scr:*
	$\{x \mid x \in (\text{deleteKILL}(B) \cap \text{OUT}(E))$, for one entry E into scr$\}$
		TWO_PHASE add
		
		TWO_PHASE alternate add

Insert Edge B_s->B_t	$\{x \mid x \in (\text{IN}(B_t) - (\text{IN}(B_t) \cap \text{OUT}(B_s)))\}$	IMMEDIATE delete

Delete Edge B_s->B_t	*if* B_t *not in scr:*
	$\{x \mid x \in (\cap\, \text{OUT}(P)$, for all predecessors P of B_t, $P <> B_s)\}$
	else if B_t *in scr:*
	$\{x \mid x \in \text{OUT}(E)$, for one entry E into scr$\}$
		TWO_PHASE add
		
		TWO_PHASE alternate add

*scr = strongly connected region

ously prevented these items from being included throughout the entire strongly connected region, making it difficult to determine *LocalSet* from the sets at the predecessors of B_c or IN(B_c). The *LocalSet* computation would consist of examining the strongly connected region to determine those items that are either generated in the strongly connected region or entering the region from outside and reaching the changed block.

Rather than performing this analysis to compute the *LocalSet*, and then running **TWO_PHASE** to perform the incremental change propagation, we execute **TWO_PHASE** with a slightly different version of the **ADDFLOW** procedure called during the first phase and an easily computed *LocalSet*. The procedure **ALT_ADDFLOW** presented in Fig. 4 is based on two observations: 1) the largest set of potentially affected data items is the set of items that are either generated somewhere in the strongly connected region or flow into the region from all entries; 2) if an item does not flow into the strongly connected region from all entries, but the change causes it to be added to at least one set within the region, then the item must be generated within the region.

Thus, it is sufficient to start the first phase of the **TWO_PHASE** algorithm at any entry into the strongly connected region with a *LocalSet* containing the elements of OUT of the entry's source node (i.e., ensuring that items that flow from all entries are added). **ADDFLOW** is changed to ensure that every node in the strongly connected region is visited at least once and the items in the GEN sets potentially having an effect within the region are added to the current change propagation set as each

```
RECURSIVE PROCEDURE ALT_ADDFLOW(ChangePropSet,node,DataSet,visited,tovisit);
   ChangePropSet:   current change propagation set.
   node:            node currently being traversed.
   DataSet:         type of data flow sets being updated (DefIN/DefOUT, AvailIN/AvailOUT,...).
   visited:         set of nodes visited during addflow update.
   tovisit:         set of nodes that must be visited at least once during update.

DECLARE NewChPropSet: Newly computed current change propagation set.

BEGIN  {add elements to data flow set at this node and continue}
NewChPropSet := ChangePropSet - (ChangePropSet $\cap$ DataSet_IN(node));
IF (NewChPropSet is not empty) or (node is in tovisit)) THEN BEGIN  {sets not fully updated}
   visited := visited $\cup$ {node};
   DataSet_IN(node) := DataSet_IN(node) $\cup$ NewChPropSet;  {update set entering node}
   IF (node in tovisit)  {update change set}
     THEN NewChPropSet := (NewChPropSet $\cup$ DataSet_GEN(node)) - DataSet_KILL(node);
     ELSE NewChPropSet := NewChPropSet - DataSet_GEN(node) - DataSet_KILL(node);
   DataSet_OUT(node) := DataSet_OUT(node) $\cup$ NewChPropSet; {update set exiting node}

   tovisit := tovisit - {node};
   FOR EACH immediate successor s of node DO  {continue traversal}
     ALT_ADDFLOW(NewChPropSet,s,DataSet,visited,tovisit);
   ENDIF;  {not fully updated}
END. {ALT_ADDFLOW}
```

Fig. 4. Procedure **ALT_ADDFLOW**.

node is visited. The *tovisit* parameter is initialized to the set of nodes composing the strongly connected region and used to ensure that every node in the region is visited. Unlike **ADDFLOW**, the *visited* variable in **ALT_ADDFLOW** is used only to compute the set of all visited nodes in the first phase in order to ensure that all of these nodes are visited at least once in the second phase. In response to a decrease in a KILL set, only those items in the GEN sets that are also in deleteKILL need to be added during the first phase.

This approach limits the reanalysis to the strongly connected region which may be affected. Another approach may be to reinitialize the sets at each node in the potentially affected region, and then run the exhaustive iterative algorithm on these nodes. However, we believe that our *adjust* phase will terminate faster than this method because the sets after our first phase are closer to the final solution. Analogous to **TWO_PHASE**, the worst case running time of **TWO_PHASE** using **ALT_ADDFLOW** is $O(|visited|^2)$ where *visited* includes all nodes in the strongly connected region plus any nodes outside the strongly connected region to which the elements generated within or flowing into the strongly connected region are added.

In summary, change propagation in both union and intersection problems consists of computing the appropriate *LocalSet* and then executing the **IMMEDIATE**, **TWO_PHASE**, or **TWO_PHASE alternate add** incremental update algorithm, depending on the program change. The *LocalSet* computation for both union and intersection problems takes a constant number of bit vector operations, contributing little to the overall running time of the incremental analysis. When an intersection problem is being updated and a KILL is decreased or an edge is deleted, the *LocalSet* computation and change propagation algorithm depends on whether the change occurs within a strongly connected region. Otherwise, the program structure surrounding the change has no effect on *LocalSet* or the method of change propagation. Only maximal strongly connected regions must be identified. In response to structural changes, update of this structural information can be done quickly using incremental or exhaustive methods.

IV. Comparison to Other Incremental Methods

The focus in this work has been global data flow problems used in the detection of various common global optimizations. Different approaches to incremental data flow analysis have been developed by Ryder [15], Ghodssi [6], Zadeck [19], and Keables, Roberson, and von Mayrhauser [11] primarily intended to provide helpful information to the programmer during editing and by Jain and Thompson [10] for use in optimization. Since our interests include the handling of edits, we do not consider further the latter work.

Modeling the data flow problem by a system of linear equations, Ryder [15] developed incremental versions of elimination methods for data flow analysis including Allen/Cocke interval analysis, Tarjan interval analysis, and the Hecht/Ullman T1-T2 technique. The incremental update algorithms consist of two phases, namely recalculating all coefficients and constants in all data flow equations that are affected by the program change, and then recalculating all affected solutions. The original work was limited to handling only nonstructural program changes; however, more recently, the work was extended to handle flow graph structure changes [3]. The worst case complexity of the incremental Allen/Cocke algorithms is $O(n^2)$ as in the exhaustive Allen/Cocke method. However, Ryder rightly claims that worst case analysis is not appropriate for incremental update algorithms as pathological cases can always be found, but they are rare.

A considerable amount of information must be maintained to enable incremental update using Ryder's method: the derived sequence of flow graphs with the interval structure and interval order of the blocks, the corresponding systems of linear equations with coefficient matrices and reduced equations for all intervals in each system, and the current data flow solutions for all variables in all of the linear systems. Also, based on elimination methods, the work is not easily applicable in an environment where programs may result in nonreducible flow graphs.

Zadeck [19] developed a different paradigm for solving a subset of data flow problems in both the exhaustive and incremental contexts. In order to avoid the added complexity of cycles in the flow graph, the data flow problem is partitioned into a series of independent problems called *clusters* which are each solved separately. Since each cluster typically corresponds to a unique variable in the program, the technique is called the Partitioned Variable Technique (PVT). Focusing on the update of a single cluster, the flow graph is manipulated according to information contained within each block. The graph is split at every block that stops propagation of the variable's information chain. This ensures that any cycle in the resulting graph will have the property that information reachable from *any* block in the cycle is reachable from *every* block in the cycle since the cycle can be executed any number of times. Thus, the strongly connected regions of the flow graph can be collapsed to form a directed acyclic graph representation of the flow graph. Information is propagated globally on the return from a depth first traversal starting at the blocks that end information chains.

By tailoring the flow graph to a single cluster, the incremental algorithms in response to most program changes, but not all, attain a worst case time complexity of $O(a)$ where a is the number of affected blocks in the original flow graph. However, when a program change such as an edge change can affect multiple variables, the algorithm must be repeated for each variable with potentially affected flow. Thus, the total incremental analysis takes $O(a*v)$ where v is the number of distinct variables with altered flow, and the incremental PVT has a running time identical to the worst case complexity of **IMMEDIATE**. The expected running time of **IMMEDIATE** is better because changes to the flow of different variables are processed as a set in **IMMEDIATE** requiring fewer visits to each affected node. Incremental PVT also requires time for each affected variable to collapse the graph and map the information back to the original flow graph.

The incremental PVT technique for updating union problems in response to adding a definition (increasing a KILL set) or deleting an edge follows a similar strategy to the **TWO_PHASE** algorithm. Old information previously propagating through the changed block is deleted, and the information sets are then rebuilt by propagating the information from the border of the affected region. The region of visited nodes corresponds to the nodes visited by the **TWO_PHASE** algorithm. However, the **TWO_PHASE** algorithm may not perform as well as the incremental PVT in these situations under certain scenarios of *LocalSet* and program structure within the *visited* region because *adjust* actually recomputes data flow equations and may visit some nodes more often than PVT.

Incremental update with PVT will require more information than our method because the transformed flow graph with data flow sets and a mapping back to the original flow graph must be maintained in addition to the original flow graph and corresponding data flow sets. PVT is also limited to the subset of data flow problems that are *partitionable* into clusters such that each strongly connected region of a cluster has the same information available for each block of that region. An example is the reaching definitions of a variable at each block. Many of the most common union data flow problems fall into this class, but Zadeck indicates that there are few, if any, intersection problems that are clustered.

Ghodssi [6] developed straightforward incremental update algorithms for union data flow problems (e.g., reaching definitions and live variable analysis) based on the Hecht/Ullman iterative global data flow analysis algorithms [9]. The local data flow sets, namely GEN and KILL, are first updated to reflect a program change. In the reaching definitions problem, a work list is initialized to the successors of the blocks containing the definitions with flow potentially affected by the program change in addition to the successors of the changed block or the target of the edge change. Unlike both **TWO_PHASE** and incremental PVT, definitions previously propagated and then blocked by an edge deletion are removed from all sets in which they occur. A similar pass is also executed when a definition is inserted into a block that contains no other definition of the same variable. After initializing the work list in response to any program change, the algorithm removes the lowest node under a depth first ordering, say B_n, from the work list, recomputes IN(B_n) by taking the union of the sets at all of B_n's predecessors, and compares the new IN set with the old IN set. If they differ, the new IN set replaces the old IN set, and all of the successors of B_n are added to the work list.

By consistently choosing the lowest node from the work list using a depth first ordering, the worst case of the incremental algorithm uses $O(d*n)$ steps where n is the number of nodes in the flow graph and d is the largest number of back edges in any acyclic path in the flow graph. In situations where **TWO_PHASE** must be applied, our approach computes a better approximation of the worst case scenario in the first phase and drives the change propagation in the second phase by data flow changes rather than a depth first ordering. It should be noted that *adjust* could be structured to utilize a depth first ordering and work list approach without affecting the overall **TWO_PHASE** algorithm. However, in addition to requiring a new computation of the depth first ordering in response to each structural change, this method also requires time to locate the least index node in the current work list each time the next visited node is needed during change propagation.

Ghodssi does not distinguish between the *any path* and *all paths* scenarios, so a less expensive update in the *any path* scenario is not exploited. The complete set of data flow equations is recomputed at each node put into the work list. Also, intersection problems are not addressed by Ghodssi. Based on the same exhaustive method, Ghodssi's technique requires space comparable to our technique, namely the flow graph representation of the program and the current data flow sets at each block.

A common thread in the work presented by Zadeck, Ghodssi, and in this paper is a two phase strategy to update data flow under the *all paths* scenario. The techniques differ in the work that is done in each phase, in particular, how the update is performed in the second phase. This two phase strategy is used to eliminate old information from the potentially affected region so the desired information can be computed on the second phase. Like Ghodssi, we use this two phase strategy to update in the presence of cycles in the flow graph, while Zadeck, who collapses all strongly connected regions, uses this approach to avoid recomputing data flow confluence on entry to each affected block.

Keables, Roberson, and von Mayrhauser [11] have developed another approach to updating union problems, particularly reaching definitions, in a maintenance environment. They assume that the variables affected by a code change (*LocalSet*) can be identified. Their algorithm first computes the affected region of the program for all variables involved in a change. As the affected region is being computed, the desired data flow sets are initialized to empty for the blocks in this region. The exhaustive iterative algorithm is then applied to these blocks. Thus, they essentially run our IMMEDIATE algorithm without updating any sets and then completely recompute the data flow sets in the affected area. Since they do not handle structural changes and focus on union problems, they are guaranteed to recompute sets only in the affected region. Techniques to find the affected region in response to edge changes would require a different analysis, similar to **TWO_PHASE**. Similar to Ghodssi, they do not exploit the faster update possible in an *any path* scenario.

All of these incremental update methods operate on a low level representation of the program, namely the control flow graph. An optimal time algorithm developed by Reps [14] for incrementally updating attribute values in an attributed parse tree after a program change could also be used to solve the incremental data flow analysis problem. However, this high level data flow analysis is not easily expressible in an attribute grammar framework when the control flow of programs does not always model their syntactic structure.

V. Conclusions

Incorporating program edits, performing intraprocedural optimizations, transforming code to enable parallel scheduling and changing interprocedural information all affect the flow of data throughout a program by changing variable definitions, uses, and control flow. The data flow information must reflect the current state of a program and thus, upon such a change, the data flow sets need to be updated. Because the updates are needed so frequently, it is vital that an efficient method be used. Our approach to this problem is the development of an incremental technique for data flow analysis. Efficiency issues including traversal of only affected areas of the flow graph and minimizing the frequency of visits to a node were considered.

Our technique was developed to handle a wide range of applications, and thus extends previous work on incremental data flow analysis. In particular, it is designed for general flow graphs and handles a wide class of programs. Due to the important role of a data flow analyzer in an optimizing compiler, it is targeted for both union and intersection data flow problems. Although only a subset of data flow problems typically applied in an optimizing compiler were specifically considered in this paper, the techniques are applicable to other union and intersection problems.

In dealing with both union and intersection problems, issues relating to generic and nongeneric data flow items were considered. Consistent with other situations involving both types of problems such as interprocedural analysis is the recognition of the added complexity of performing incremental analysis on intersection problems. We found that the incremental analysis for intersection problems must often consider whether the program change occurs within a strongly connected region. However, the time to run our incremental algorithms is not affected by the nesting depth of loops and looplike program structures.

Efforts are underway to implement a programming environment in which incremental compilation of optimized code plays an important role. The incremental data flow analyzer has been implemented and will be a central tool in this environment. Experiments are currently being conducted to evaluate the performance of the incremental technique.

References

[1] A. Aho, R. Sethi, and J. Ullman, *Compiler Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[2] F. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, pp. 137-147, Mar. 1977.

[3] M. Carroll and B. Ryder, "An incremental algorithm for software analysis," in *Proc. ACM SIGSOFT/SIGPLAN Symp. Practical Software Development Environments*, Dec. 1986, pp. 171-179.

[4] K. Cooper, K. Kennedy, and L. Torczon, "The impact of interprocedural analysis and optimization on the design of a software development environment," in *Proc. ACM Symp. Language Issues in Programming Environments*, June 1985, pp. 107-116.

[5] J. Ellis, *Bulldog: A Compiler for VLIW Architectures*. Cambridge, MA: MIT Press, 1985.

[6] V. Ghodssi, "Incremental analysis of programs," Ph.D. dissertation, Univ. Central Florida, 1983.

[7] S. Graham and M. Wegman, "Fast and usually linear algorithms for global flow analysis," *J. ACM*, vol. 23, no. 1, pp. 172-202, Jan. 1976.

[8] R. Gupta and M. Soffa, "Region scheduling," in *Proc. Second Int. Conf. Supercomputing*, vol. III, May 1987, pp. 141-148.

[9] M. Hecht and J. Ullman, "A simple algorithm for global data flow analysis problems," *SIAM J. Comput.*, vol. 4, pp. 519-532, Dec. 1975.

[10] S. Jain and C. Thompson, "An efficient approach to data flow analysis in a multiple-pass global optimizer," in *Proc. ACM SIGPLAN 1988 Conf. Programming Language Design and Implementation*, June 1988, pp. 154-163.

[11] J. Keables, K. Roberson, and A. von Mayrhauser, "Data flow analysis and its application to software maintenance," in *Proc. IEEE Conf. Software Maintenance*, Oct. 1988, pp. 335-347.

[12] K. Kennedy, "A survey of data flow analysis techniques," in *Program Flow Analysis Theory and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[13] L. Pollock, "An approach to incremental compilation of optimized code," Ph.D. dissertation, Univ. Pittsburgh, Pittsburgh, PA, Apr. 1986.

[14] T. Reps, T. Teitelbaum, and A. Demers, "Incremental context-dependent analysis for language-based editors," *ACM TOPLAS*, vol. 5, no. 3, pp. 449-477, July 1983.

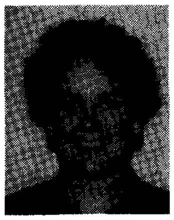
[15] B. Ryder, "Incremental data flow analysis," in *Conf. Rec. Tenth ACM POPL Conf.* Jan. 1983, pp. 167-176.

[16] B. Ryder, T. Marlowe, and M. Paull, "Incremental iteration: When will it work?," Dep. Comput. Sci., Rutgers Univ., New Brunswick, NJ, Tech. Rep. LCSR-TR-89, 1987.

[17] R. Tarjan, "Testing flow graph reducibility," *J. Comput. Syst. Sci.*, vol. 9, pp. 355-365, 1974.

[18] J. Ullman, "Fast algorithms for the elimination of common subexpressions," *Acta Inform.*, vol. 2, no. 3, pp. 191-213, 1973.

[19] F. Zadeck, "Incremental data flow analysis in a structured program editor," in *Proc. ACM SIGPLAN 1984 Symp. Compiler Construction*, June 1984.



Lori L. Pollock (S'84–M'86) received the B.S. degree in computer science and economics from Allegheny College, Meadville, PA, in 1981, and the M.S. and Ph.D. degrees in computer science from the University of Pittsburgh, Pittsburgh, PA, in 1983 and 1986, respectively.

Since January 1986, she has been an Assistant Professor in the Department of Computer Science at Rice University, Houston, TX. Her primary research interests include incremental compilation, compiler optimization, programming environments for highly optimized and parallel programs, and parallel compilation.

Since September 1988, Dr. Pollock has been a visiting lecturer for the ACM Lectureship Program. She is a member of the Association for Computing Machinery, ACM SIGPLAN, and the IEEE Computer Society.

Mary Lou Soffa received the Ph.D. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, in 1977.

She has been on the faculty at the University of Pittsburgh since 1977 and is currently an Associate Professor in the Department of Computer Science. She was awarded an NSF Visiting Professorship for Women in 1987 to spend a year at Berkeley. Her main areas of research interest are compiling techniques for parallel computers, incremental compilation, programming languages, and software tools.

Dr. Soffa serves on the Editorial Advisory Board for *Computer Languages* and is a member of the Association for Computing Machinery, Sigplan, Sigsoft, and the IEEE Computer Society.