July, 1985

INCREMENTAL ALGORITHMS FOR SOFTWARE SYSTEMS

Barbara G. Ryder

DCS-TR-158

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

ABSTRACT

This report is a 2 year NSF research proposal in the area of incremental algorithms for analyzing software systems. The systems of interest can be characterized as large and complex, in the sense that they are written and maintained by many people. The algorithms we are designing will enable people to alter the intermodular structure of such systems and *a prior* see the side effects of such changes.

1. Project Summary

Data flow analysis algorithms gather facts about the use and definition of data in programs. Since software development occurs in a dynamic environment, algorithms are needed to maintain the correspondence between data flow information and the changing program text. Incremental update algorithms which only change affected information, avoiding total recalculation in response to program changes are ideal for this purpose. Update algorithms can be profitably applied to interprocedural data flow analysis, where the information they gather aids software development.

The goal of our research is to develop a taxonomy of program structures and changes amenable to incremental updating. We plan to extend our interval-based incremental update algorithms to become more generally applicable and to develop new update algorithms based on the iterative algorithm. Algorithm performance will be studied in its application to interprocedural analysis. We will gather empirical information on the structure of interprocedural communication in large software systems written in C and study their development by collecting change histories which will be summarized. We will develop a prototype implementation of our extended interval-based incremental algorithms to test their average performance. Based on knowledge gained from our empirical studies, we will analyze our incremental update algorithms on a restricted class of program/change combinations, proving the efficacy of incremental updating over re-analysis.

Our research in programming languages includes theoretical work in algorithm analysis and design, empirical fact gathering and implementation of a prototype.

2. Introduction

Historically, data flow analysis which gathers facts about the definition and use of data in a program or set of programs, was used for code optimization. The presence of procedure calls complicated this process. For example, in Figure 2-1,

```
     declare w,x,y,z : integer
     procedure P(a : integer)
     . . .
     end P
     x := 2
     P(y)
L:   z := 5 * x
     end
```

Figure 2-1: Side Effects of Procedure P

we can substitute **z:=10** for statement **L**, if we know that the call on **P** does not alter the value of **x**, which is a global variable accessible in **P**. Interprocedural data flow analysis tells us which variables may have their values changed as a result of the execution of a call of **P**; this is **summary data flow information** for P. The variables affected may include variables appearing explicitly in the call as arguments, (i.e., **y**), as well as other global variables which are accessible to **P** (i.e., **x**). Aliasing makes it impossible to discern these affected variables from the program text of **P** without further analysis; **aliasing** occurs when two distinct names refer to the same storage location. Parameter passing mechanisms, static storage sharing through programmer specification as in Fortran EQUIVALENCE statements, and dynamic storage sharing through pointer assignments all can result in aliasing. In Figure 2-1, assuming that parameters are passed by reference, if the call **P(w)** occurred in the main program, then during the execution of that call, **w** and **a** would be aliases in **P**; that is, code in **P** containing references to **w** or **a** would be referring to the same location.

Interprocedural data flow analysis algorithms summarize the behavior of each procedure, its effects on its parameters and on global variables. Several interprocedural algorithms were developed in the 1970's, but most of them are not yet used in production compilers [3, 6, 7, 25, 36]. We are interested in using these algorithms to analyze large software systems during their development and maintenance. Most interprocedural analysis algorithms presumed a batch environment; it was assumed that they would be applied once to analyze a set of procedures. There was no mechanism to deal with program changes, other than re-application of the algorithm to the entire set of procedures.

Software development is a dynamic process in which interprocedural data flow information can aid in debugging and documentation. Data flow information made available in interactive programming environments can greatly aid programmer understanding during development. There is a need to keep data flow information current, corresponding to a dynamically changing collection of procedures. Only recently has this task been addressed in programming environments [12, 15, 24, 37]. In dealing with program changes, data flow information should be calculated once and updated thereafter to reflect the effects of these program changes. If the update work is less than the effort expended in completely re-solving the problem, this strategy is efficient. With updated interprocedural data flow information, we can insure the consistent use of global data and confirm the correctness of procedure calling sequences as a software system is being developed, saving testing and debugging effort.

To be of practical utility in dealing with large software systems, interprocedural algorithms must embody the notion of updating. An **incremental update algorithm** for data flow analysis modifies known data flow information derived from a program to reflect changes in that program; that is, an incremental update algorithm obtains the new solution of an altered problem without application of the exhaustive data flow algorithm which was originally applied. Clearly, an incremental update algorithm is quite useful in a software development process which involves experimenting with different system configurations.

Another important application of incremental update algorithms is in software maintenance. Large systems often maintain histories of source code changes. Our proposed studies of these histories would yield information about the kinds of changes large systems are likely to undergo. This information alone would be valuable to software designers, as the transformation of a set of algorithms and data structures into a working software system is not well understood in large practical applications, although various software design techniques exist.

Software maintenance usually involves making small changes to a large system whose internal structure is often unfamiliar to the programmer performing the changes. Data flow information documents the system for the programmer. The PFORT Verifier was a program which checked Fortran programs for adherence to a portable subset of ANS Fortran '66 [27, 28]. It performed simple-minded analyses of interprocedural communication using parameters and COMMON and reported the procedure calling relations. The popularity of this program attests to the benefit of even rudimentary interprocedural

analysis of large systems. An incremental update algorithm for interprocedural analysis would enable the programmer to delineate the scope of a system change, thus providing an even more useful analysis tool.

Incremental update algorithms are necessary for source-to-source transformation systems. Here, data flow information is used to trigger certain transformations which subsequently change source code and may change the associated data flow information. This information insures the semantic correctness of these transformations. Currently, *ad hoc* methods are used to update information in response to local changes; a more systematic approach is necessary.

Incremental update algorithms are applicable between the phases of an optimizing compiler. Optimizing transformations are not independent; one transformation often creates the opportunity for another optimization. To find these opportunities and thus perform all possible optimizations, data flow information must be updated to insure that it correctly describes the current state of the program. In optimizing compilers, usually the data flow problems are re-solved; updating may be a better approach.

The research in this proposal will extend our previous work in the design of incremental update algorithms for data flow analysis to applications in interprocedural analysis. In improving our algorithms and designing new algorithms, we will develop general performance measures for incremental updating as an algorithm design methodology. Existing incremental algorithms for interprocedural analysis are problem dependent; they are not generally applicable to an arbitrary problem solvable by some underlying algorithm. We will explore the design of incremental algorithms of general applicability with the aim of putting current work in a general theoretical framework.

In addition, our research involves empirical studies of common communication structures of large software systems and the kinds of changes made to such systems. Our goal is to have our empirical findings direct our algorithm design efforts towards consideration of structures and changes which actually occur in practice. Also, if we find good incremental algorithms dealing with specific changes on certain structures, our empirical studies will give us evidence of the frequency with which these combinations occur. Our planned prototype implementation of our extended interval-based incremental algorithm will enable us to obtain measurements of actual algorithm performance.

The remainder of this proposal consists of:

- **Basic Concepts** - definitions and background on our previous work in incremental update algorithms for data flow analysis,
- **Research Objectives** - elaboration of the research goals,
- **Relation to Other Work** - background in interprocedural and incremental data flow analysis,
- **Research and Implementation Plan** - proposed chronology of activities.

3. Basic Concepts

In our previous work, we developed incremental update algorithms for data flow analysis by modelling how Allen/Cocke interval analysis solves the set of equations describing a data flow problem [4, 32]. Based on this model we built an update algorithm by only repeating those parts of the algorithm necessary to calculate the effects of a program change. In this section we describe the definition of a data flow problem as the solution of a set of equations. We present Allen/Cocke interval analysis as a solution procedure for these equations. We describe our interval-based incremental update algorithms and outline the complexity results we have obtained for them. Extended discussions of these topics are presented in our paper in the Appendix [31].

Our data flow algorithms are based on a digraph representation of the procedure or set of procedures being analyzed. This is either a **flow graph** which describes the possible execution flow between **basic blocks**, single-entry single-exit code sequences, or a **call graph** which describes the possible calling relations in a set of procedures [13]. Each node in the digraph has associated data flow constants which describe how the code represented by that node affects data. The desired data flow information is describable as a solution to a system of equations whose variables correspond to data flow on entry to or on exit from nodes in the digraph.[1] In this discussion, we will describe **forward data flow problems**, that is, problems in which the information is propagated in the same direction as the execution of the program. The solution of a forward problem is a set of facts true on entry to a node. **Backward data flow problems** are problems in which the information propagation occurs in the reverse direction of execution flow; corresponding

[1] Linear transformations easily facilitate going from the "on entry" form of the solution to the "on exit"; therefore we can use either form without loss of generality.

definitions and descriptions exist for backward problems as well [29]. Consider equations of the form:

$$X_m = \odot\, j \in pred(m)\ \{\ a_{m,j} \cap X_j \cup b_{m,j}\ \} \qquad (1)$$

where

X_m is a set of facts true on entry to m
pred(m) is a subset of the nodes in the digraph
$\odot$ is $\cup$ or $\cap$
$a_{m,j}, b_{m,j}$ are data flow constants associated with node j

These equations are solvable by a Gaussian elimination-like technique [22, 32]. The basic step in this method is repeated substitution. Given a term in X_m which occurs somewhere in the system of equations, we substitute for that term, the right hand side of the equation for X_m. This repeated substitution is the basic step of both the elimination and propagation phases of the algorithm. During elimination, we substitute for every X_m term in the system, thus removing X_m from the system of equations. This process may introduce a self-dependence into an equation; that is, the process may yield an equation of the form $X_k = d \cap X_k \cup e$ where d is a constant and e is independent of X_k. We must define a loop-breaking rule by which we can remove this equation from the system and replace it with another equation such that a solution to the revised system of equations is also a solution to the original system of equations [22]. The appropriate loop-breaking rule is determined by the operators in the equations, here $\cup$ and $\cap$. Loop-breaking rules often select an extremal solution for the system, that is, a largest or smallest solution. In the classical data flow problems we select a minimal solution for union problems and a maximal solution for intersection problems [13]. In our above example, we can replace the self-dependent equation $X_k = d \cap X_k \cup e$ by $X_k = e$. At the end of elimination, the system consists of one equation in one variable, which is solvable. During propagation, we replace a term in X_m by the solution for X_m. By doing this repeated substitution of solutions in equations using the reverse order in which variables were eliminated, we obtain solutions for all the variables.

Given this representation of a data flow problem, we developed a model of how Allen/Cocke interval analysis solves these equations. The key idea of interval analysis is to establish a linear order among the variables, **interval order**, such that when the equations are written in that order they exhibit a highly structured and easily solved coefficient

matrix. The algorithm reduces the problem of solving n equations to one of solving k equations by partitioning the n equations into k subgroups called **intervals** which it solves individually. Repeated substitution steps within an interval derive the reduced equation for each variable in an interval as a linear function of the interval head variable. A new system of equations is formed, consisting only of former interval head variables and their equations; we call this the **derived system of equations**. We re-perform the process of interval finding and reduction on this new system. Repeated application of these steps reduces the set of equations in the system to a final equation which can be solved. Then, by substituting the solution for an interval head variable into the reduced equations of all variables in its interval, we derive solutions for all these variables. Each variable is identified with corresponding interval head variable in the previous system of equations and their solutions are set equal. We repeat these steps of substitution and variable identification through the sequence of systems of equations taken in reverse order until all solutions in the original system of equations are obtained. The process described here assumes that the original system of equations is associated with a **reducible** digraph, a digraph containing only single-entry loops [13]. This assumption seems reasonable in practice in intraprocedural data flow analysis; it is a hypothesis in interprocedural data flow analysis [14, 20].

Our incremental update algorithms **ACINCF** and **ACINCB** for forward and backward data flow problems respectively, are based on our model of Allen/Cocke interval analysis. Essentially we mimic the interval algorithm for equations affected by the changes. During the elimination phase, we propagate the effects of any coefficient changes through the affected intervals, obtaining new reduced equations for some variables. In the propagation phase, we recalculate the solutions for any updated equations and for any interval in which the interval head solution has changed. We have proved that if the original system of equations is reducible, then the effects generated by changes on the derived system of equations are limited. In any derived system, the equations affected consist only of a set of interval head equations and, at most, the equations in one interval in that system [30, 31].

We analyzed the behavior of our algorithms on a robust structured programming language L with multiple exit loops similar to SAIL. We identified loop exit structures that affect the complexity of incremental updating and established the extent of this effect offered by combinations of such structures. We also showed that the depth of loop nesting affects the complexity of the update algorithms and the interval analysis algorithm as well.

For a reducible flow graph with loop nesting level bounded by a constant, Allen/Cocke interval analysis exhibits **O(n)** worst case behavior. We considered **localised program changes**, non-structural changes affecting the data flow constants at a set of nodes within one interval in a nested loop in a program in L. We characterized the set of all variables whose equations might be affected by these changes, in terms of their corresponding program structures and their relation to the original change site. This result allows us to examine a flow graph in L with a set of possible program changes, identifying *a priori* equations which may be affected by these changes. If a change occurs in a heavily nested loop, the propagation phase of the algorithm may have to be re-preformed throughout that loop; this work is linear in the size of the loop, as it consists of the re-substitution of an interval head solution into each of the reduced equations for variables in its interval. Our result allows us *a priori* to bound the extent to which the elimination phase must be re-performed on the systems of equations.

4. Research Objectives

The goal of the research in this proposal is to categorize situations in which incremental update algorithms for data flow analysis are demonstratably superior to exhaustive re-analysis. To accomplish this goal, we will classify programs by the structure of their digraph representation and enumerate the possibilities for changes to them. We expect that for specific program/change pairs, an incremental update algorithm will have provably good performance. Our prototype implementation of our incremental update algorithms will provide empirical evidence of the effectiveness of these algorithms.

This research is an extension of our previous efforts in the design and analysis of incremental update algorithms based on elimination methods. We have new results indicating that the worst case behavior of our previous algorithms is the best we can expect from a general purpose update algorithm [8]. By **general purpose update algorithm** we mean an update algorithm intended to handle any program change for any program structure. We intend to extend our previous algorithms to become general purpose update algorithms, by enabling them to handle structural program changes (i.e., additions or deletions of edges or nodes in the digraph), to accommodate Tarjan intervals which more closely reflect the loop structure of the program than the Allen/Cocke intervals, and to deal with the possible irreducibility of the digraph. Then we can use these algorithms as a basis for comparison with other update algorithms. To better understand the performance of our extended algorithms, we will implement a prototype version of them

in an interprocedural setting (see Section 6).

Given our previous work as a basis, the proposed extensions are feasible. Structural changes which do not affect the interval structure of the digraph are easily accommodated. When the interval structure is affected, it may prove unreasonable to perform an update, because it would involve work tantamount to re-performing the exhaustive analysis algorithm. In an interprocedural application where our algorithm is used during system development, typical changes would involve changing the parameters in a procedure (i.e., non-structural changes) and addition/removal of procedure calls (i.e., structural changes). Adding a procedure call can introduce a new cycle in the call graph; deleting a call may disconnect parts of the call graph which were formerly connected. Both kinds of changes may or may not destroy the interval structure of the original graph as Tarjan intervals are strongly connected components of the digraph [34].

Empirical evidence on typical call graph structures and their evolution during program development gathered during our research, will help direct our attention to changes an incremental update algorithm must handle to be practical. Although we are discussing structural changes in an interprocedural domain, an algorithm dealing with such changes would be equally applicable to the intraprocedural domain since the same representation is used in both. We aim to derive an *a priori* bound on algorithm performance for this extended algorithm, as we did for our previous algorithms (see Section 3).

The question of handling irreducibility in the digraph is interesting theoretically, although it is unclear from empirical evidence whether this is of practical import. Our empirical studies hopefully will clarify the importance of handling irreducibility. Our current plan of accommodating irreducibility in the extended algorithm is to isolate an irreducible subgraph and and treat it as a "super-node", performing data flow analysis on this area separately using the iterative method, and subsequently factoring its data flow effects into the rest of the graph.

We also will investigate the development of general purpose incremental update algorithms based on the iterative algorithm [13, 19]. The wide range of problems amenable to application of this method will insure the utility of incremental update algorithms based on it. Also, the simplicity of the iterative algorithm often results in its being preferred in practice over other methods. It is a compact, easily programmed algorithm, although if naively coded it may be quite inefficient. Our aim is to characterize the performance of an

iterative-based update algorithm in a manner similar to our previous analyses.

To develop a taxonomy of problems amenable to incremental updating, we will consider a set of program structures with possible program changes to them. Our aim will be to demonstrate program/change pairs for which incremental updating is provably better than re-application of an exhaustive algorithm. Suggestions of program/change pairs to be considered will be provided by empirical studies of large software systems written in C. Our empirical studies will focus on categorizing common call graph structures, patterns of parameter and global variable usage, aliasing etc. Having obtained this information, we will focus on "typical" call graph structures in defining the program component of the program/change pairs. Then, we will empirically study the dynamic development of some large software systems by collecting change histories maintained by source control systems for systems under development. By categorization of these changes, we can focus our attention on "reasonable" changes. In a sense, the empirical information will direct our investigations toward changes the update algorithms must handle. Use of the static analysis and history recording tools has another benefit as well. If we identify a particularly interesting class of programs in our theoretical analysis, we can verify whether this class actually occurs in practice. Thus, the empirical studies in this research will both direct and reinforce the analytic work.

Another benefit of this empirical information is that we will be able to make simplifying assumptions with respect to difficult programing language constructs like pointers. Our hypothesis is that the data flow information obtainable from large programs written in C, a language which encourages the use of pointers, will not be limited by pointer usage because that usage will be constrained in order for the system to remain demonstrably correct. We will empirically test this hypothesis.

The overriding theoretical question being addressed by this research is "How well can we accommodate change in a problem by using an incremental update algorithm?" The research in this proposal seeks to answer that question with respect to interprocedural data flow analysis, an area where there are clear application benefits from incremental update algorithms.

To summarize, our research objectives are:

i. to extend our interval-based incremental update algorithms to become general purpose algorithms

ii. to investigate the development of general purpose incremental update algorithms based on the iterative algorithm.

iii. to characterize program/change pairs for which incremental update algorithms are provably good

iv. to empirically gather information about common structures of interprocedural communication and reasonable system changes encountered during software development.

v. to implement a prototype version of our extended incremental update algorithm

5. Relation to Other Work in the Area

The proposed research outlined here is primarily based on our previous investigations of incremental update algorithms for data flow analysis; however, there is relevant previous work which impacts on our efforts. In this section we will consider work related to our research objectives. In addition, there have been related investigations of incremental update algorithms, which we will briefly describe at the end of this section.

Our objective of extending our existing incremental update algorithms to handle structural changes, irreducible digraphs and Tarjan intervals, is primarily related to our previous work. Our incremental algorithms **ACINCF** and **ACINCB**, the model of Allen/Cocke interval analysis on which these algorithms are based, and the bounds on the work of updating we established are described in in Section 3 and elaborated on in the paper in the Appendix [31].

In considering how to modify our algorithms to deal with structural changes in the digraph, we distinguish between structural changes which effect the interval structure of the digraph and those which do not. This distinction was hypothesized by Burke in his presentation of an interval-based algorithm to solve for the **formal bound set** of each formal parameter in a procedure, that is, the set of formal parameters which may be aliased to a parameter during program execution [9]. This information is necessary to calculate the side effects of procedures in a program. For example, in Figure 5-1 the call of **Q** at **L1** binds **a** to **b**. The call of **P** at **L2** binds formal parameter **x** to parameter **b**; subsequently, the call of **Q** at **L2** binds **x** to **b**. Burke's algorithm calculates all the sets

```
        procedure R(x)
        procedure P(a)
        ...
L1:     Q(a)
        ...
        end P
        procedure Q(b)
        ...
        b :=
        ...
        end Q
L2:     P(x)
        end R
```

Figure 5-1: Formal Bound Set

of pairs of formal parameters (e.g., **(a,b)**, **(x,b)**) which bear this relation to one another some time during the execution of the program. If parameters are passed by reference in this program then the store into **b** in procedure **Q** is in effect a store into **a** as well! If procedure calls change, Burke wishes to update the bound set information already calculated; he suggests use of our incremental update algorithm for non-structural changes. He suggests differentiating between those structural changes which affect the interval structure of the call graph and those which do not. For the former, he abandons the updating approach; for the latter, he hypothesizes the applicability our update algorithm. We will elaborate on these suggestions and study the resultant algorithm, considering alternatives as well.

The idea of isolating the irreducible portion of the digraph and using iteration to solve its data flow was suggested by Rosen and also is patterned after the approach taking by Mintz, Fischer and Sharir in building an optimizing compiler based on analytic methods of data flow analysis [21, 26]. In regions of the program where the analytic method could not be used, they used an iterative method to solve for the data flow and then calculated the side effects of this region with respect to the rest of the program.

Our objective of implementing an incremental update algorithm for interprocedural data flow analysis, relies on existing work in interprocedural data flow analysis algorithms. Many algorithms have been developed for this analysis, although few are yet implemented in working compilers [1, 2, 6, 7, 9, 12, 25, 33, 36]. The work of Banning, Burke, and Cooper are most relevant to our implementation.

Banning calculates precise summary data flow information for programs with recursion

and aliasing. He finds aliases using a depth first exploration of calling chains in the program (i.e., paths in the call graph). Each alias pair that Banning calculates is realizable along some execution path in the program; his aliases are more accurate than those obtained from relation-based methods [7, 33]. Figure 5-2 illustrates the inaccuracy of the relation-based algorithms. Statement **N** causes **Z** to be aliased to **B** and statement **M** causes **W** to be aliased to **D**, but the relation-based algorithms calculate that both **Z** and **W** are aliased to both **B** and **D**.

```
        procedure P (X : procedure, Y : integer)
        ...
        X(Y)
        ...
        end P
        procedure A (Z : integer)
        ...
        end A
        procedure C (W : integer)
        ...
        end C
        ...
N:      P(A,B)
M:      P(C,D)
        end
```

Figure 5-2: Aliasing

Banning's main contribution is his structuring of the summary data flow problem for procedures. Here we consider his formulation for the question "What variables may have their values affected by execution of a particular procedure call?". We call these variables the **MOD set** of the procedure. Banning separates the data flow analysis of each procedure from consideration of aliasing within the program. He factors in the side effects caused by aliases in a final step of his algorithm. This formulation is used by Cooper as well.

To calculate alias-free MOD information, Banning defines two sets:

DMOD(s) - the set of variables directly modified by statement s

GMOD(P) - the set of variables possibly modified by a call on procedure P, ignoring aliasing

To calculate these sets, he defines the following equations which are amenable to solution by any of a number of data flow analysis algorithms.

IMOD(P) - the set of variables in {y | y $\in$ DMOD(s) where s$\in$P, s not a procedure call}

GLOBAL(P) - the set of variables global to and accessible to P

REF(P) - the set of formal reference parameters of P

The equations relating these quantities are:

$$GMOD(P) = (GLOBAL(P) \cup REF(P)) \cap (IMOD(P) \cup (\cup_{s \in P} DMOD(s), s \text{ a call}))$$

$$DMOD(s) = (GMOD(Q) \cap GLOBAL(Q)) \cup \{x \mid y \in GMOD(Q) \cap REF(Q), \text{ where actual argument x is bound to formal parameter y at call s}\}$$

Cooper presents a formulation of this problem which splits the calculation of MOD into one involving possible modifications of global variables and another involving modification of reference parameters [11]. This split enables him to solve each subproblem more efficiently. The global modification problem is solved by application of the depth first formulation of the iterative algorithm. The formal parameter modification problem is solved by the algorithm developed by Tarjan for single source path expression problems [35].[23] Since Cooper is implementing in the interactive programming environment R^n, he is concerned about program changes [17, 18]. He considers updating the formal parameter associations due to changes in the sequences of procedure calls, either new parameter bindings, new calls or deleted calls. These parameter relations are represented by a reflexive, transitive closure relation Map*, where the Map relation describes all the pairs of parameter association along an edge in the call graph. He suggests techniques for updating Map* after program changes [12]. Additional bindings can be easily factored into Map*; deletions cannot. To handle deletions, Cooper suggests restarting the iteration which originally formed Map* after deleting the known defunct relations.

Our investigation of an incremental update algorithm based on the iterative method, will generalize these suggestions recently offered by Cooper. He hypothesized the use of the iterative method to form updates which were problem specific; we will investigate the general applicability of these suggestions. We will study the efficacy of this algorithm, only

---

[2]Tarjan himself suggests that this almost linear algorithm should not be used in practice.

[3]Burke offers an alternative solution procedure for the global variable and reference parameter bound set problems (see previous discussion).

limiting the problems we consider if necessary, to obtain sharp performance results or to justify the correctness of updating.

Our characterizations of program/change pairs for which incremental update algorithms are a proven "win", will rely on the re-application of analysis techniques used in previous work [31]. In considering reasonable program structures and changes, we will be guided by the results of our empirical studies as well as other published results [5, 10]. We also will have access to in-house studies of C within Bell Laboratories [16].

There are other researchers addressing the general problem of incremental updating in different problem domains, including the updating of attributes in language-based editors, the updating of data flow information within a basic block, and the updating of specialized data flow information. These efforts are related to our work in that they too are trying to replace a repeated exhaustive calculation by an incremental one. However, these efforts are directed either to algorithms quite different from the interval and iterative algorithms we are considering or to the development of incremental algorithms applicable to a limited set of problems. Our work is directed towards designing a general purpose incremental algorithm.

In the language-based editors, attribute grammars are used to capture factual information about an evolving program. The update problem here involves pruning a subtree from a consistently attributed parse tree, replacing it with another subtree and making the resulting parse tree consistently attributed. Two recent algorithms have attacked this problem [15, 24]. If we define **Affected** to be the set of attribute instances in the altered tree which require new values after a subtree replacement, both algorithms claim a worst case bound of **O(Affected)**.

The updating of local data flow information is being studied in the context of providing symbolic debugging within an optimizing compiler [23]. Using a three address code representation of the program, an algorithm is presented which responds to additions/deletions of code by undoing any optimizations which are invalidated and performing any additional optimizations enabled.

Zadeck has developed data flow analysis algorithms based on the iterative method for **cluster problems**, a limited set of problems which are partitionable into independently solvable subproblems or clusters [37]. For most common data flow problems, a separate

cluster must be solved for every variable in the program. Zadeck designed incremental update algorithms for his method which can respond to structural and non-structural program changes.

6. Research and Implementation Plan

The implementation objective in this proposal can be characterized as having three major activities, empirical fact gathering, change analysis and prototype building. First, we must perform static analysis of the characteristics of procedure call sequences, parameter passing and recursion usage in large programs in C, in order to design a reasonable incremental interprocedural analysis algorithm. In collecting this information, we will rely on the software tools provided by UNIX. C is an attractive programming language to study because of the availability of large programs to serve as test data, (e.g., the system itself) and its wide usage as the language of systems programming with UNIX. With these studies we will broaden the currently available information about static analysis of procedure interaction in modern programming languages. They will be valuable in that they provide empirical evidence of realistic structures for interprocedural communication.

Second, we will instrument some working source control system to monitor the kinds of changes which occur in the development and maintenance of practical software systems. This will provide for us a notion of "reasonable" program changes as well as some idea of their frequency of occurrence.

Third, we will implement a prototype of our extended incremental update algorithm. A full implementation involves having the front end of a compiler as well as extensive intraprocedural analysis routines. We hope to augment an existing compiler to do intraprocedural analysis at least for the MOD problem, perhaps avoiding the calculation of aliases until a later implementation. Then, we will build the interprocedural analysis module with the capability of doing incremental analysis; our objective will be to study algorithm performance. In building this prototype, we will also deal with data representation issues which are not covered in the algorithm design. A great deal of information must be saved to do incremental updating; the best organization of this information must be determined through experimentation. Eventually, we would like to have an implementation of our algorithms available to solve a variety of interprocedural problems in the presence of aliasing, as a part of an existing compiler. However, the information gained from this prototype implementation will be valuable as an experimental

laboratory for testing the average performance of the algorithms and the ease of their implementation. The idea of a prototype coupled with a static analysis tool which insures that the test data is realistic is appealing.

This research has two year timetable. The first year will be spent gathering static procedure usage information, gathering and investigating change histories and extending our current incremental algorithms. We will also begin some initial investigations of the iterative algorithm, by doing case studies of the classical data flow problems. Theoretical work will be focused on the development of an iterative-based incremental update algorithm. The static analysis implementation will be the major responsibility of RA_1, with RA_2 analyzing the change histories and familiarizing himself with our incremental algorithm through the extension work. During the second year, both RA's will implement the prototype. Theoretical work will focus on finishing the design of an iterative based incremental update algorithm and restricting that algorithm in order to prove tighter performance bounds.

References

[1] Allen, F. E.
A Basis for Program Optimization.
In *Proceedings of 1971 IFIP Congress*, pages 385-390. Institute of Electrical and Electronics Engineers, Inc., North Holland Publishing Company, Amsterdam, Holland, 1971.

[2] Allen, F. E. and Schwartz, J. T.
Determining the Data Relationships in a Collection of Procedures.
private communication.
1973

[3] Allen, F. E.
Interprocedural Data Flow Analysis.
In *Proceedings of 1974 IFIP Congress*, pages 398-402. Institute of Electrical and Electronics Engineers, Inc., North Holland Publishing Company, Amsterdam, Holland, 1974.

[4] Allen, F. E. and Cocke, J.
A Program Data Flow Analysis Procedure.
Communications of the ACM 19(3):137-147, 1977.

[5] Banning, J. P.
A Method for Determining the Side Effects of Procedure Calls.
PhD thesis, Department of Electrical Engineering, Stanford University, 1978.

[6] Banning, J.
An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables.
In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29-41. Association for Computing Machinery-SIGPLAN, January, 1979.

[7] Barth, J. M.
A Practical Interprocedural Data Flow Analysis Algorithm.
Communications of the ACM 21(9):724-736, 1978.

[8] Berman, M. A., Paull, M. C., and Ryder, B. G.
Proving Relative Lower Bounds for Incremental Algorithms.
Computer Science Technical Report DCS-TR-154, Department of Computer Science, Rutgers University, April, 1985.

[9] Burke, M.
An Interval Analysis Approach Toward Interprocedural Data Flow Analysis.
Computer Science Technical Report RC 10640, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, July, 1984.

[10] Carter, L. R.
An Analysis of Pascal Programs.
UMI Research Press, 1982.

[11] Cooper, K. D.
Interprocedural Data Flow Analysis in a Programming Environment.
PhD thesis, Department of Mathematical Sciences, Rice University, 1983.

[12] Cooper, K. and Kennedy, K.
Efficient Computation of Flow Insensitive Interprocedural Summary Information.
In *Proceedings of SIGPLAN '84 Symposium on Compiler Construction*, pages 247-258. June, 1984.
SIGPLAN Notices, Vol 19, No 6.

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

[14] Hobbs, S.
private communication.

[15] Johnson, G. and Fischer, C.
A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors.
In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 141-151. Association for Computing Machinery-SIGPLAN, January, 1985.

[16] Johnson, S. C.
private communication.

[17] Hood, R. T. and Kennedy, K.
A Programming Environment for Fortran.
1983

[18] Hood, R. T. and Kennedy, K.
Programming Language Support for Supercomputers.
1983

[19] Kildall, G.
A Unified Approach to Global Program Optimization.
In *Conference Record of the ACM Symposium on the Principles of Programming Languages*, pages 194-206. Association for Computing Machinery-SIGPLAN, January, 1973.

[20] Knuth, D. E.
An Empirical Study of FORTRAN Programs.
Software Practice and Experience 1:105-133, 1971.

[21] Mintz, R. J., Fisher, G. A. and Sharir, M.
The Design of a Global Optimizer.
In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 226-234. Association for Computing Machinery-SIGPLAN, August, 1979.
SIGPLAN Notices, Vol. 14 No. 8.

[22] Paull, M. C.
Introduction to Algorithm Design Principles.
Wiley-Interscience, 1985.
pre-publication manuscript.

[23] Pollack, L. L. and Soffa, M. L.
Incremental Compilation of Optimized Code.
In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 152-164. Association for Computing Machinery-SIGPLAN, January, 1985.

[24] Reps, T.
Optimal-time Incremental Semantic Analysis for Syntax-directed Editors.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 169-176. Association for Computing Machinery-SIGPLAN, January, 1982.

[25] Rosen, B. K.
Data Flow Analysis for Procedural Languages.
Journal of the ACM 26(2):322-344, April, 1979.

[26] Rosen, B. K.
private communication.

[27] Ryder, B. G.
The PFORT Verifier.
Software Practice and Experience 4:359-377, 1974.

[28] Ryder, B. G.
Constructing the Call Graph of a Program.
IEEE Transactions on Software Engineering SE-5(3):216-225, May, 1979.

[29] Ryder, B. G.
Incremental Data Flow Analysis Based on a Unified Model of Elimination Algorithms.
PhD thesis, Department of Computer Science, Rutgers University, 1982.

[30] Ryder, B. G.
Incremental Data Flow Analysis.
In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 167-176. Association for Computing Machinery-SIGPLAN, January, 1982.

[31] Ryder, B. G. and Paull, M. C.
Incremental Data Flow Analysis Algorithms.
1983.
to appear in ACM Transactions on Programming Languages and Systems.

[32] Ryder, B. G. and Paull, M. C.
A Unified Model of Elimination Algorithms.
1984.
being reviewed for publication in ACM Computing Surveys.

[33] Spillman,T .
Exposing Side Effects in a PL-I Optimizing Compiler.
In *Proceedings of IFIPS Conference*, pages TA-3-56:TA-3-62. 1971.

[34] Tarjan, R. E.
Testing Flow Graph Reducibility.
Journal of Computer and System Sciences 9:355-365, 1974.

[35] Tarjan, R. E.
Fast Algorithms for Solving Path Problems.
Journal of the ACM 28(3):594-614, 1981.

[36] Wiehl, W. E.
Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83-94. Association for Computing Machinery-SIGPLAN, January, 1980.

[37] Zadeck, F. K.
Incremental Data Flow Analysis in a Structured Program Editor.
In *Proceedings of SIGPLAN '84 Symposium on Compiler Construction*, pages 132-143. June, 1984.
SIGPLAN Notices, Vol 19, No 6.