Incremental Analysis of Side Effects for C Software Systems*

Jyh-shiarn Yur† **Barbara G. Ryder**† **William A. Landi**‡ **Phil Stocks**†

†Department of Computer Science
Rutgers University
Piscataway NJ 08855, USA
+1 732 445 2001
{yur,ryder,pstocks}@cs.rutgers.edu

‡Siemens Corporate Research, Inc.
755 College Road East
Princeton NJ 08540, USA
+1 609 734 6500
landi@scr.siemens.com

ABSTRACT
Incremental static analysis seeks to efficiently update semantic information about an evolving software system, without recomputing "from scratch." Interprocedural modification side effect analysis (MOD) calculates the set of variables possibly modified by execution of a procedure or a statement. We introduce a partial incrementalization of MOD for C systems using the hybrid method and present results of a study of 27 C programs, that predicts that our incremental MOD analysis will be substantially cheaper than exhaustive analysis for many program changes.

Keywords
Dataflow analysis, incremental analysis

INTRODUCTION
Information about the flow of data through a program is crucial to software testing, debugging, program understanding, and maintenance, especially of large software systems. Data flow testing of C systems depends on having accurate definition-use information in the presence of assignments through dereferenced pointers. Debuggers can profit from being able to display all the potential uses of a particular variable definition at a breakpoint. Software understanding tools used by maintainers to assess the impact of a source code bug fix or enhancement use extracted side effect descriptions for a program. Interprocedural modification side effect analysis (MOD [LRZ93]) finds the set of variables in a program whose values may be affected by program execution. Side effects are reported for an individual statement and/or an entire procedure. Such information is available at the cost of a system build; however, as a system evolves over time, this information needs to be updated to reflect current system state. For large systems, keeping descriptive semantic information up-to-date is costly. Instead of recomputing side effect information "from scratch," we use *incremental* static analysis techniques, which harness knowledge of the original side effects of the system and subsequent source changes.

*This work was supported, in part, by NSF Grant CCR-9501761.

ICSE 97 Boston MA USA
Copyright 1997 ACM 0-89791-914-9/97/05 ..$3.50

This paper describes how we have taken *PMOD*, a significant subproblem of MOD for C, and made it incremental using the hybrid algorithm for data flow analysis [MR90]. This data flow method partitions the flow graph into components, uses fixed point iteration within components to solve local versions of the data flow problem, and knits together these local solutions into a global solution.

The *call-RA graph*, our program representation used to solve *PMOD*, can experience two kinds of changes, *non-structural* and *structural*. We show how to update *PMOD* with regard to these changes. One source code change can result in several representation changes; therefore, we consider various source changes and analyze their corresponding representation changes and updates.

In our empirical investigation we have seeded assignment statement and function call deletions in 27 C programs and have instrumented our representation to calculate metrics directly proportional to the work necessary to perform the updates. These calculations show that incremental analysis will be substantially cheaper than total reanalysis.

Related Work
Many incremental algorithms have been developed for the data flow analysis, which are useful, especially in a programming environment [Zad84]. Some incremental analyses use incremental elimination methods [Bur90, CR88, RP88]; some are based on the technique of restarting iteration [CK84, PS89]. When a change is made, restarting iteration from the previously computed solution does not always yield a precise solution. [RMP88] shows some sufficient conditions under which restarting iteration results in the same solution as an exhaustive iterative method. Pollock and Soffa [PS89]

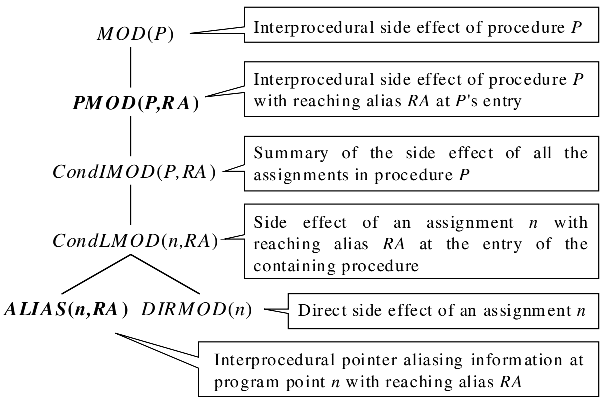


Figure 1: Decomposition of the *MOD* problem

present precise incremental iterative algorithms, using change classification and reinitialization, for bitvector problems. A comparison of these incremental iterative algorithms is found in [BR90]. The incremental hybrid algorithm [MR90] handles changes by combining the elimination and iteration methods. A hybrid incremental algorithm for the reference parameter aliasing problem of Fortran is proposed in [MR91].

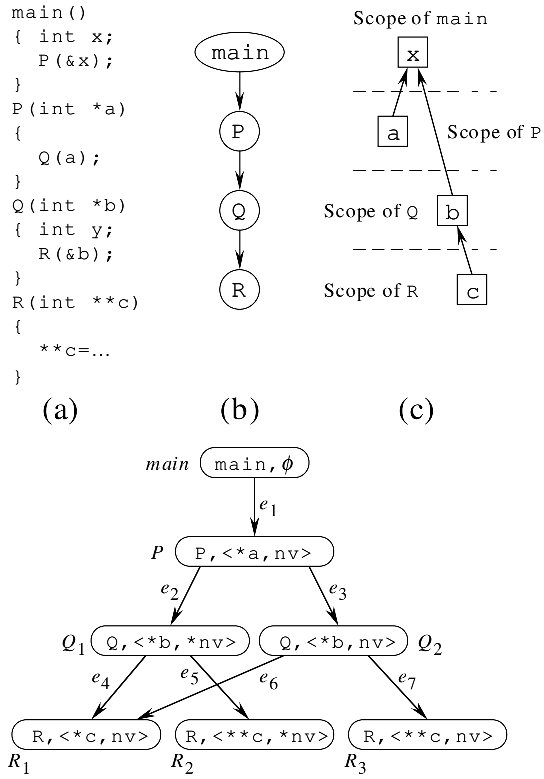
For the *MOD* problem for Fortran-like languages, efficient flow-insensitive algorithms are proposed in [CK84, Coo85], and the incremental MOD problem for Fortran was handled in [Bur90]. The incremental *MOD* problem for C is more complicated than its counterpart for Fortran mainly because the pointer aliasing information for C may change inter- and intra-procedurally.

BACKGROUND AND DEFINITIONS

MOD Problem for C

The *MOD* problem ascertains how program execution may affect the values of variables in that program. Statement execution can change the value of a variable either by direct assignment or by indirect assignment through an alias of the lefthand-side variable. Figure 1 shows the relevant parts of the problem decomposition of *MOD* for C programs with general-purpose pointers [LRZ93]. In this decomposition, there are only two data flow problems: *ALIAS* and *PMOD*. All others shown are just set combinations of their subproblems.

$DIRMOD(n)$ contains the set of variables which may experience direct side effects through assignments at program point n. This can be calculated while parsing the program during compilation. $ALIAS(n, RA)$ is the solution of the interprocedural may alias problem (IPMay_Alias) at statement n in the presence of general purpose pointers [LR92]. If `<*x,*y>` is in $ALIAS(n, RA)$, this means that `x` and `y` may point to the same object on exit from statement n. *Reaching alias RA* approximates the calling context under which we are performing analysis at statement n.[1] For example, at statement `n: *p=`$\cdots$, $ALIAS(n, RA)$ yields a safe estimate of those variables to which p may point, that is, those variables which may experience side effects at n. By widening of direct side effects to include indirect effects, we obtain $CondLMOD(n, RA)$ from $DIRMOD(n)$ and $ALIAS(n, RA)$.



Edge	e_1	e_2	e_3	e_4	e_5	e_6	e_7
nv_backbind	{x}	{a}	{nv}	{b}	{nv}	{b}	{nv}

Source,Destination	P,R_1	P,R_2	P,R_3
nv_backbind *	{ }	{a}	{nv}

(d)

(a) Source program (b) Call graph
(c) Memory map (d) Call-RA graph

Figure 2: An example of a call-RA graph

We only report side effects to *fixed locations*, variables which have fixed addresses during each invocation of a procedure; `p` is a fixed location but `*p` is not.[2]

[1] We will refer to a specific reaching alias or calling context interchangeably. We have proved previously that for programs limited to a single level of dereference, a single reaching alias precisely represents calling context for IPMay_Alias [LR92]. Empirical evidence indicates that this also closely approximates calling context for the side effect calculation in programs with multiple levels of dereferencing [LRZ93].

[2] We name heap variables by their creation site, obtaining an approximate fixed location for those locations allocated [LRZ93].

$CondIMOD(P, RA)$ is the union of $CondLMOD(n, RA)$ sets for all statements n in procedure P. *CondIMOD* summarizes all side effects in P except those due to call statements.

$PMOD(P, RA)$ is the entire set of fixed locations modified by a procedure P in calling context RA, including the effects of calls from P. The *PMOD* solution is formed from local *CondIMOD* solutions at call sites and *PMOD* information propagated from procedures called by P. Our analysis makes the usual assumption of static analysis, namely that all intraprocedural paths are executable. The RA's in our analysis enable a good approximation of the possibly executable interprocedural paths (i.e., *realizable* paths) [LRZ93]. $MOD(P)$ summarizes possible side effects over all executions of procedure P (i.e., for all calling contexts which occur).

New Program Representation

Interprocedural static analyses often use a *call multigraph*: a graphical representation of the possible calling relations between procedures in a program. A node represents a procedure and an edge from P to Q represents a call from P to Q. In order to distinguish calling contexts during our *PMOD* analysis, we introduce the *call-RA graph*, a variant of a call multigraph.[3] In a call-RA graph, each node corresponds to a procedure and a reaching alias pair. Each edge represents information at a specific call site flowing on a path from the entry of the calling procedure to the entry of the called procedure. The size and precision of the *call-RA graph* depend on the precision of the algorithms for computing the call graph and pointer aliases.

Figures 2(b) and (d) show the call multigraph and call-RA graph for the simple program in Figure 2(a). Figure 2(c) shows the memory locations pointed to by the parameters at the entry of procedure `R`. Procedure `main` calls procedure `P` by passing the address of `x` to formal `a` of `P`. The parameter binding creates alias pair `<*a,x>`; this alias is used to represent this calling context. Note how the alias captures the relationship in Figure 2(c). Since `x` is a local in `main`, and thus not accessible in `P`, we use a special symbol, `nv`, instead of `x`, as an abstraction of the variables in the calling procedure which are *non_visible*, but still accessible through their aliases in the called procedure [LR92]. Note that `nv` in different nodes can represent different things.

We also keep track of this binding information for `nv` along the edge $e_1 = ((\texttt{main}, \emptyset), (\texttt{P}, < *\texttt{a}, \texttt{nv} >))$ by defining $nv\_backbind_{e_1}$ as the set of all names in node $(\texttt{main},\emptyset)$ to which `nv` in node `(P,<*a,nv>)` can bind, (namely, $\{\texttt{x}\}$). Given `<*a,nv>` at the entry of procedure `P`, the call of `Q` creates two alias pairs, `<*b,*nv>` and `<*b,nv>` and thus generates two call-RA graph nodes Q_1 and Q_2, with corresponding edges e_2 and e_3. The `nv` in node Q_1 represents local `a` in procedure `P`, and the `nv` in node Q_2 represents `nv` in node P, which itself represents `x` in *main*. Thus use of a formal of the calling procedure as an actual argument can create `nv` as a name in $nv\_backbind_e$. Figure 2(d) shows the *nv_backbind* sets for other edges and paths in the call-RA graph. $nv\_backbind^*$ is explained in the next section.

[3] The call-RA graph is not a multigraph, however.

Reformulation of ***PMOD***

Figure 3 shows the original *PMOD* equation (equation (1)), and our equivalent new formulation in equations (2) and (3), using $nv\_backbind^*_e$, which makes it easier to apply the hybrid algorithm [MR90]. E' is the edge set of the call-RA graph. The function b maps names between scopes, mapping globals to themselves, discarding locals (and formal parameters), and mapping *non_visibles* to their corresponding names in scope. In equation (1), *PMOD* was calculated as the union of indirect and direct side effects within the procedure and those side effects due to calls, on globals and *non_visible* variables (through their aliases). $nv\_backbind_{((P,RA),(Q,RA'))}$ is used by a binding function from an immediate successor (Q, RA') to (P, RA) to account for *non_visibles*. In equation (3), we use $nv\_backbind^*_{((P,RA),(R,RA''))}$ to account at (P, RA) for *non_visibles* from any node (R, RA'') reachable through a non_visible binding chain from a successor of (P, RA). A *non_visible binding chain* is a path in the call-RA graph along each edge of which *nv_backbind* contains `nv`. $nv\_backbind^*$ keeps track of *non_visibles* through multiple levels in the call-RA graph. For example, in Figure 2(d), $nv\_backbind^*_{(P,R_2)}$ is $\{\texttt{a}\}$, and $nv\_backbind^*_{(P,R_1)}$ is $\emptyset$, because there is a non_visible binding chain (e_5) connecting Q_1 and R_2, but there is no such chain between a successor of P and R_1. Also, (e_3 e_7) form a non_visible binding chain from P to R_3. Thus, $nv\_backbind^*_{(main,R_3)} = \{\texttt{x}\}$. *PMOD* is formulated in equation (3) as a reachability problem in that $PMOD(P, RA)$ collects $CondIMOD(R, RA'')$ of all reachable (R, RA'') and maps them into the scope of procedure P.

Incremental Hybrid Algorithm

The hybrid data flow analysis algorithm [MR90] is based on a graph decomposition (usually by strongly connected components) which divides the flow graph into single entry regions. Local variants of the data flow problem are defined and solved on each region. These problems make it possible to calculate the data flow solution within the region as a function of incoming information from the region exits (for backward data flow problems such as *MOD*) or the region entry (for forward problems such as reaching definitions). Thus, we can perform a propagation of global data flow information forwards or backwards in topological order on the

$$PMOD(P,RA) = CondIMOD(P,RA) \cup \bigcup_{e=((P,RA),(Q,RA'))\in E'} b(PMOD(Q,RA'), nv\_backbind_e) \quad (1)$$

$$nv\_backbind^*_{((P,RA),(R,RA''))} = \bigcup_{\substack{e=((P,RA),(Q,RA'))\in E', \text{ and} \\ \exists \text{ a non\_visible binding chain con-} \\ \text{necting } (Q,RA') \text{ and } (R,RA'')}} nv\_backbind_e \quad (2)$$

$$PMOD(P,RA) = CondIMOD(P,RA) \cup \bigcup_{\substack{(R,RA'') \text{ is reach-} \\ \text{able from } (P,RA)}} b(CondIMOD(R,RA''), nv\_backbind^*_{((P,RA),(R,RA''))}) \quad (3)$$

Figure 3: Reformulation of *PMOD*

reduced graph, followed by a propagation within each region to solve the data flow problem. The benefit of a hybrid approach is that it enforces locality on a data flow problem and thus contains the extent of iteration. Thus, it is also very suitable for adapting to an incremental approach.

The solution procedure can be made incremental in the following manner for backward problems. Given a source code change, map it into the corresponding changes to the flow graph. For changes that do not invalidate the current graph decomposition or topological order, first update the solutions to the local problems, where necessary, in the regions corresponding to source code changes. Secondly, if some region entry node solution has changed, then propagate this change backwards on the reduced graph to region ancestors, as far as necessary. For each region whose exit node(s) have a changed solution, recalculate the global solutions for nodes within that region. If a source code change affects the region decomposition or topological order, then these must be adjusted before these other steps and recalculation occurs from change points [MR90].

The hybrid algorithm must be specialized to specific instances of data flow problems, such as *PMOD*. The key step in specializing a problem is factoring the global data flow problem into a set of local problems that are solved separately on the region. In the next section we describe incremental *PMOD* in terms of the hybrid algorithm schema for different source code changes.

INCREMENTAL *PMOD*

For our incremental *PMOD* algorithm, we use a partitioning of the call multigraph to induce a partitioning on the call-RA graph. For all nodes N in the same region of the call graph as node P, $Region(P)$ denotes the region of the call-RA graph containing nodes $\{(N,*)\}$. The topological order is calculated on the reduced call-RA graph (which is isomorphic to the reduced call multigraph).

Factorization of *PMOD*

Observing equation 3 in Figure 3, we can see that the global *PMOD* solution at a node n in a region depends on the local *PMOD* information of the region, the global *PMOD* information propagated from the other regions, and the mapping of the global information to the scope of node n. Thus, we factor the *PMOD* problem into three local problems: LOC_1, LOC_2, and LOC_3, which will be used to capture those three types of information. The global *PMOD* solution at a node in a region can then be recovered from the solutions to these local problems.

In contrast with $PMOD(P,RA)$ in equation (1), $LOC_1(P,RA)$ (Figure 4, equation (4)) at node (P,RA) is a restricted instance of $PMOD(P,RA)$ to $Region(P)$, which summarizes the effects of *CondIMOD* of all reachable nodes from (P,RA) within $Region(P)$. $LOC_2(P,RA)$, defined by equation (5), calculates the exit nodes of $Region(P)$ reachable from node (P,RA). Global information arriving at these exit nodes is then gathered and mapped back to the scope of (P,RA). In computing $nv\_backbind^*$ (Figure 3, equation (2)), we need to determine reachability through a non_visible binding chain. Thus, we define another local problem $LOC_3(P,RA)$, using equation (6) in Figure 4, as the set of exit nodes which are reachable from (P,RA) through a non_visible binding chain. Using $LOC_3(P,RA)$, equation (7) defines $loc\_nv\_backbind^*_{((P,RA),(X,RA''))}$, which is similar to $nv\_backbind^*_{((P,RA),(X,RA''))}$ but restricted to $Region(P)$.

Combining the solutions to those local problems and the global information propagated from other regions, we are able to recover the global *PMOD* solution for a node (P,RA) using equation (8). In this equation, $LOC_1(P,RA)$ captures the local effect of the region containing (P,RA). For every exit (X,RA') reachable from node (P,RA), the *PMOD* information $PMOD(H,RA'')$ from a head node (H,RA'') of one of

$$LOC_1(P,RA) = CondIMOD(P,RA) \cup \bigcup_{\substack{e\,=\,((P,RA),(Q,RA''))\,\in\,E' \\ \text{and } Q\,\in\,Region(P)}} b(LOC_1(Q,RA'), nv\_backbind_e) \tag{4}$$

$$LOC_2(P,RA) = \bigcup_{\substack{((P,RA),(Q,RA'))\,\in\,E' \\ \text{and } (P,RA)\,\neq\,(Q,RA')}} \begin{cases} \{(P,RA)\} & \text{if } Q \notin Region(P) \\ LOC_2(Q,RA') & \text{otherwise} \end{cases} \tag{5}$$

$$LOC_3(P,RA) = \bigcup_{\substack{e\,=\,((P,RA),(Q,RA'))\,\in\,E' \\ \text{and } (P,RA)\,\neq\,(Q,RA')}} \begin{cases} \{(P,RA)\} & \text{if } Q \notin Region(P) \\ LOC_3(Q,RA') & \text{else if } \mathtt{nv} \in nv\_backbind_e \\ \emptyset & \text{otherwise} \end{cases} \tag{6}$$

$$loc\_nv\_backbind^*_{((P,RA),(X,RA''))} = \overline{\bigcup_{\substack{e\,=\,((P,RA),(Q,RA'))\,\in\,E' \\ \text{and } Q\,\in\,Region(P) \text{ and} \\ (X,RA'')\,\in\,LOC_3(Q,RA')}} nv\_backbind_e} \tag{7}$$

$$\begin{aligned} PMOD(P,RA) &= LOC_1(P,RA) \cup \bigcup_{(X,RA')\in LOC_2(P,RA)} b(EXT(X,RA'), loc\_nv\_backbind^*_{((P,RA),(X,RA'))}) \\ &\quad \text{where } EXT(X,RA') = \bigcup_{\substack{e\,=\,((X,RA'),(H,RA''))\,\in\,E' \\ \text{and } H\,\notin\,Region(P)}} b(PMOD(H,RA''), nv\_backbind_e) \end{aligned} \tag{8}$$

Figure 4: Factorization of *PMOD*

its successor regions is propagated to (X, RA'). All the global information arriving at an exit node (X, RA') is collected and mapped back to node (P, RA) using $loc\_nv\_backbind^*_{((P,RA)(X,RA'))}$.

Next, we will consider two classes of changes on the call-RA graph, non-structural and structural changes, and analyze how to update the solutions to reflect these changes.

Non-structural Changes
Non-structural changes only change local information, like *CondIMOD* and *nv_backbind*; the structure and region decomposition of the call-RA graph remain the same.

Change to *CondIMOD* When the values of *CondIMOD* at some nodes change, simply restarting fixed point iteration from the old solution does not always yield a correct solution. Restarting iteration can get a correct solution only if the old solution is a *safe*[4] initial estimate; otherwise, all or part of the solutions must be re-initialized to a safe initial value [RMP88]. Changes that can be accommodated by restarting iteration are called *additive*.

[4]For the *PMOD* problem, an initial estimate is safe, if we can be sure that it will be contained in the final *PMOD* solution.

Figure 5 shows the algorithm for handling changes of *CondIMOD*. Only the LOC_1 information is affected by the change of *CondIMOD*. Step 2 checks which changes are additive and does the necessary re-initialization. A *CondIMOD* change is additive if the set of global and *non_visible* variables of the new *CondIMOD* is a superset of that of the old *CondIMOD*. Thus, for an additive change of *CondIMOD*, the old LOC_1 solution is a safe initial estimate, and restarting iteration will yield a correct solution. Otherwise, a simple and safe new initial estimate can be made by re-initializing LOC_1 at the changed node and its ancestors in the same region to the corresponding *CondIMOD*. Iteration using a worklist is then applied to compute the new local solutions in step 3 of Figure 5. Step 4 updates the global *PMOD* solutions in the regions needing to be changed based on the updated local solutions.

Change to *nv_backbind* Change of $nv\_backbind_e$ associated with an edge e is another kind of non-structural change, which affects LOC_1 and LOC_3. If the changed edge connects two regions, since it only changes the incoming information to the source region of edge e, the local solutions are still valid. After recalculating the incoming global *PMOD* information to the source region, we can just invoke subroutine *PropagatePMOD* (step 4 in Figure 5) to update the *PMOD* solutions in

Algorithm ModifyCondIMOD

1. $Worklist = \{(P, RA) \mid CondIMOD(P, RA) \text{ has changed}\}$
2. For those nodes in $Worklist$ whose *CondIMOD* change is not additive, re-initialize LOC_1 of those nodes and all their ancestors in the same region to their respective *CondIMOD*.
3. While $Worklist$ is not empty
   - 3.1 Remove (P, RA) from $Worklist$.
   - 3.2 Re-compute $LOC_1(P, RA)$ using equation (4).
   - 3.3 If $LOC_1(P, RA)$ changes, then add predecessors of (P, RA) in *Region*(P) to $Worklist$.
4. *PropagatePMOD*, starting at the changed region with the largest topological order number.
   - 4.1 In a reverse topological order of the reduced graph, compute the *PMOD* solutions at entry nodes using equation (8), and propagate the changed solutions to their predecessor regions until there are no changes at entry nodes.
   - 4.2 For each region which receives updated global *PMOD* information, recalculate the global solutions for nodes within that region.

Figure 5: Algorithm for $ModifyCondIMOD$

the regions with changed global *PMOD* information. If the changed edge is within a region, before information propagation, we have to update the solutions of LOC_1 and LOC_3 within that region, analogously to steps 1-3 in Figure 5. Details about the algorithm $ModifyBackbind$ are given in [YR95].

Structural Changes

Structural changes are those which can change the shape of the call-RA graph or its decomposition, and thus are harder to handle than non-structural changes. An edge may be added or deleted if the aliases reaching the entry of a procedure change. Figure 7 shows structural changes made by changing the aliases reaching a call (i.e., deletion of edge e_5 and addition of edge e_8).

In the following discussion, we will consider edge deletion and addition. Addition or deletion of nodes can be converted to a sequence of edge changes.

Delete an Edge Case 1 of algorithm *DeleteAnEdge* in Figure 6 deals with deleting an inter-region edge[5], which is similar to changing the *nv_backbind* of an inter-region edge.

Case 2 in Figure 6 summarizes what is done for deleting an edge $e = ((P, RA), (Q, RA'))$ within a region[6]. In this case, LOC_1, LOC_2, and LOC_3 at (P, RA) and its ancestors in the same region are invalidated because of the edge deletion, and thus need to be re-initialized as in step 2.1 before restarting iteration. After re-initialization, we update the local solutions, and then the global solution.

Add an Edge Adding an edge within a region is an additive change with respect to our local problems, and can be accommodated by restarting iteration from previous solutions. But, if the edge addition is caused by adding a new function call in the source program, it will also cause an edge addition to the call multigraph. Then, we have to check if it induces a cycle in the reduced graph and the single-entry property for regions in the call multigraph. If either of these two properties is violated, we have to do a region merge in order to recover the violated properties. Then, we are able to handle the inter-region edge as an edge within the resulting region. The case for edge addition without region merge is similar to edge deletion. Details about algorithm *AddAnEdge* are given in [YR95].

IMPACT OF SOURCE CHANGES

We have shown how to handle changes to the call-RA graph. The incremental algorithm is effective when the impact of the source code changes on the call-RA graph and the *PMOD* solution is small. In this section, we discuss how a source code change may affect the *PMOD* solution through changes to the call-RA graph. The primary statements of interest are assignment statements and function calls, and both may change the set of aliases reaching their statement exits. A change on the aliasing information may change the indirect side effect of the assignment, and thus change *CondIMOD* for the procedure. In addition, changes reaching a call site may change the aliasing information reaching the entry of the called procedure and the parameter binding

[5] Edge deletion can sometimes render a region exit node no longer an exit, and then the node should be removed from LOC_2 and LOC_3 for all nodes in the region. In addition, deleting an inter-region edge may also make a region unreachable from the root of the call-RA graph.

[6] Sometimes, edge removal within a region can make the region further decomposable. In parallel data flow analysis, decomposition of large regions is beneficial to load balancing among processors.

Algorithm DeleteAnEdge(e)

1. If e is an inter-region edge $((X, RA), (H, RA'))$:
   1.1 Recalculate the *PMOD* information arriving at exit node (X, RA).
   1.2 Call *PropagatePMOD* (step 4 in Figure 5), starting at *Region*(X).
2. If e is an intra-region edge $((P, RA), (Q, RA'))$:
   2.1 For every node (R, RA'') in the set of (P, RA) and its ancestors in *Region*(P), re-initialize $LOC_1(R, RA'')$ and $LOC_2(R, RA'')$ to $CondIMOD(R, RA'')$ and $\emptyset$, respectively. If $nv\_backbind_e$ contained `nv`, re-initialize $LOC_3(R, RA'')$ to $\emptyset$.
   2.2 Iterate to the solutions of LOC_1, LOC_2, and LOC_3 on *Region*(P) using equations (4), (5), and (6) respectively.
   2.3 Call *PropagatePMOD* (step 4 in Figure 5), starting at *Region*(P).

Figure 6: Algorithm for *DeleteAnEdge*

information.

Assignment Statements An assignment statement which may assign a pointer value is called a *pointer assignment*; otherwise, it is called a *non-pointer assignment*. A non-pointer assignment statement cannot affect aliasing. Thus, changes to a non-pointer assignment cause changes to *CondIMOD* of the containing procedure only, which are non-structural. If the statement altered is a pointer assignment, then it may cause non-structural and/or structural changes to the call-RA graph.

The impact on the *PMOD* solution made by the change of an assignment depends not only on the type of the statement (pointer or non-pointer assignment), but also on what variables are modified and how the containing procedure is used. For example, if we remove a non-pointer statement that modifies only a local variable, then such a source change only will affect the *PMOD* solutions for the containing procedure. However, if the statement we remove modifies a global variable that is not modified elsewhere in the program, then doing so may cause changes to the *PMOD* solutions for all procedures calling it directly or indirectly; the larger the set of such procedures, the greater the impact.

Actual-Formal Bindings Changing actual-formal bindings at a call site may change the set of variables to which a *non_visible* variable in the called procedure can bind, and thus change the values of *nv_backbind* associated with the edges corresponding to the call. Of course, it may also cause changes in the reaching aliases for the called procedure. This could result in edge deletion and/or edge addition in the call-RA graph.

Procedure Calls Insertion of a call may cause new aliases to reach the entry of the called procedure; that is, new edges to new nodes in the call-RA graph may be created. On the other hand, deletion of a call may cause the opposite effect (i.e., edge deletion in the call-RA graph.) Basically, the impact on the call-RA graph and the *PMOD* solution of inserting/deleting a function call depends on how the called function is used in the program. For example, if we remove the only call to a key function in the program, it sometimes disconnects many other functions invoked by that function, and thus causes lots of edges to be deleted. On the other hand, deleting a call to a function that is a leaf node in the call-RA graph causes much less impact on the graph. Any change at a call site may also kill and/or create aliases reaching those statements after the call site, so it may cause all kinds of call-RA graph changes.

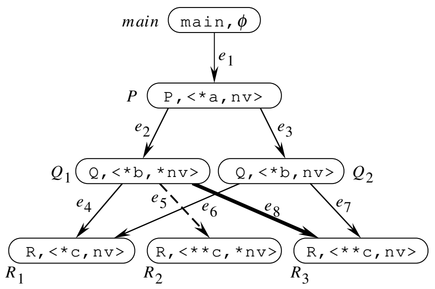
Example In general, a change of the source program may cause a sequence of changes, structural and/or non-structural, to the call-RA graph. For example in Figure 7(a), we add a pointer assignment statement `b=&y` to the source program, and it causes the following call-RA graph changes:

(i) *CondIMOD* changes at nodes Q_1 and Q_2,
(ii) an *nv_backbind* change of edge e_7,
(iii) deletion of edge e_5, and
(iv) addition of edge e_8.

The first two changes are non-structural, and the others are structural. Figure 7(b) shows the resulting call-RA graph. The algorithms mentioned in the previous section can then be invoked to handle these changes of the call-RA graph, one by one. Thus, algorithm *ModifyCondIMOD* is called to handle changes in *CondIMOD* for Q_1 and Q_2, algorithm *ModifyBackbind* to handle the *nv_backbind* change of edge e_7, algorithm *DeleteAnEdge* to handle deletion of edge e_5, and algorithm *AddAnEdge* to handle addition of edge e_8. Figure 7(c) also gives the *PMOD* solutions before and after the source change. We can see that the source change affects the *PMOD* solutions of four call-RA graph nodes:

```
main()
{ int x;
  P(&x);
}
P(int *a)
{
  Q(a);
}
Q(int *b)
{ int y;
  b=&y;
  R(&b);
}
R(int **c)
{
  **c=...
}
```

(a)



Edge	e_1	e_2	e_3	e_4	e_6	e_7	e_8
nv_backbind	{x}	{a}	{nv}	{b}	{b}	{y}	{y}

(b)

Node	*main*	*P*	Q_1	Q_2	R_1	R_2	R_3
Old *PMOD*	{x}	{nv}	{ }	{nv}	{ }	{ }	{nv}
New *PMOD*	{ }	{ }	{b, y}	{b, y}	{ }	{ }	{nv}

(c)

(a) Source program (b) Call-RA graph
(c) *PMOD* solutions

Figure 7: An example of source change

main, *P*, Q_1, and Q_2.

Multiple Changes In this section, we have described the updates as separate steps for ease of understanding. By examining the algorithms, we can find that each of our proposed updating algorithms is composed of two major steps: local solution updating and inter-region information propagation. Although any change to the call-RA graph can be handled by applying the relevant algorithm, in the presence of multiple changes to the call-RA graph, we can achieve better efficiency by processing multiple changes as a whole. That is, instead of applying the relevant algorithms to handle the changes individually, for intra-region changes, we can combine the local solution updating steps of the required algorithms, and update the local solutions for each region in one single iteration. Then, one pass of inter-region information propagation can be applied to handle inter-region changes and propagate updated information among regions. In handling multiple changes, this method avoids multiple iterations within each region and multiple passes of inter-region information propagation.

Predicting Impact As we can see, an arbitrary, even small, change of the source program could result in a wide range of changes to the call-RA graph. The impact of a source change is not solely dependent on the kind of change made. So, to make general statements about kinds of source code changes we need an empirical study of impacts of many source changes. Our study is discussed in the next section.

Program	lines of code	# inter-mediate code	# procs	# CRAG nodes	# CRAG edges	# intr. stmts	intr. stmt. coverage
allroots	215	420	8	29	59	32	36.34%
fixoutput	401	615	7	43	72	36	13.66%
diffh	268	644	15	114	214	48	24.19%
travel	862	696	16	47	78	134	38.74%
ul	541	1026	16	908	1646	133	33.63%
plot2fig	1435	1079	27	123	193	176	42.57%
lex315	719	1300	18	117	511	143	35.52%
compress	1490	1318	16	80	99	190	29.97%
loader	1220	1563	31	791	1581	168	31.84%
mway	700	1576	23	376	523	173	31.12%
stanford	887	1769	48	238	332	225	39.27%
pokerd	1120	1915	29	161	281	223	32.31%
dixie	2129	2339	37	331	589	267	28.52%
learn	1461	2622	39	1198	1827	201	17.92%
xmodem	1705	2686	29	239	439	311	28.87%
compiler	2232	3006	39	131	346	527	44.00%
sim	1422	3019	16	557	725	376	34.34%
cdecl	3623	3193	33	1751	2499	131	10.05%
assembler	2693	3602	53	4539	18753	470	34.75%
gnugo	2901	3651	30	94	139	352	21.00%
lharc	3296	4247	88	3454	6487	618	38.47%
patch	2736	4603	56	2041	4373	604	29.55%
simulator	3735	5574	107	3813	13643	674	37.00%
arc	9573	5874	102	4863	9464	791	33.04%
triangle	1925	6117	19	268	337	557	37.50%
tbl	2588	6138	86	11464	29772	479	19.75%
football	2222	7313	59	925	2202	434	16.81%

Figure 8: Experiment dataset

EMPIRICAL EFFECTIVENESS OF INCREMENTAL ANALYSIS

We studied 27 programs to show the impact of source changes on the call-RA graph and the *PMOD* solution. To determine the effectiveness of incremental analysis, we measured the impact of deleting single source statements. One reason we chose this approach is that it was reasonably simple to implement, which enabled us to collect data on many potential changes across many programs. Ideally, we would like to study change histories of programs, so that the changes we test more accurately reflect modifications likely to be made by programmers. Unfortunately, we don't have enough change history data to be statistically significant. We did not analyze the impact of statement addition by the reverse process of adding statements back to the programs because it's hard to automatically generate a sensible addition, and because this is a complementary change to the deletion yielding essentially the same results.

For each program in our dataset, we determined the *interesting* source statements (assignment statements and function calls), calculated the *PMOD* solution for the program, and then recalculated it after deleting each interesting source statement one by one (Each test is on

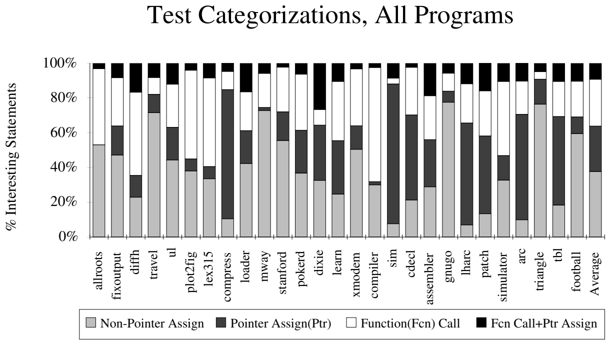


Figure 9: Test categorizations

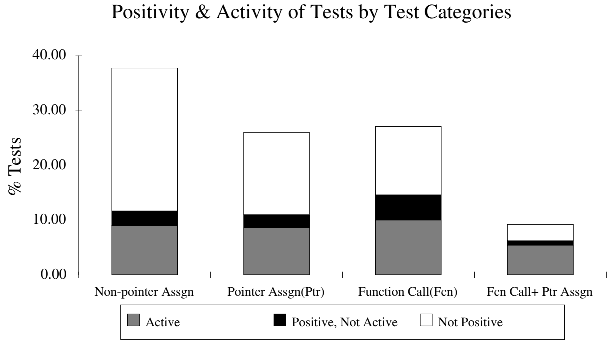


Figure 10: Percentages of tests having impact on the call-RA graph and the *PMOD* solution

the original program with a single statement deleted). We call each such interesting statement a *test*.

The programs are listed in Figure 8, ordered by the number of statements in our intermediate representation [LRZ93, YR95]. We also show the number of nodes and edges of the corresponding call-RA graphs. The rightmost two columns show the number of tests (i.e., the number of interesting source statements) and the percentage of the intermediate code corresponding to the set of interesting source statements. Over all programs, this percentage was on average about 30%.

We categorize the interesting source statements into four categories: non-pointer assignments, pointer assignments, function calls without pointer assignments, and functions calls with pointer assignments (e.g., `str3=strcat(str1,str2)`). Deleting a non-pointer assignment will be the simplest source change and cause the least impact on the call-RA graph and the *PMOD* solution. On the other hand, deleting a function call with pointer assignments will result in the most changes on the call-RA graph, and the *PMOD* solution. Figure 9 shows the relative percentages of these interesting statements found in each program, and the rightmost bar represents the average across all programs. It is easy to see that the non-pointer assignments occupy a great proportion (about 40%) of the interesting statements. Function calls with pointer assignments do not occur very often (only 9% of the interesting source statements).

Not every statement deletion will cause changes to the call-RA graph, and not every change to the call-RA graph will affect the *PMOD* solution at some node. A test is called *positive* if it causes changes to the call-RA graph or information associated with nodes or edges of the call-RA graph, and is called *active* if it causes changes to the *PMOD* solution; all active tests are also positive. Deletion of different types of statements may have varying degrees of impact on the call-RA graph and the *PMOD* solution.

Figure 10, shows the corresponding percentages of positive and active tests respectively, for each statement category across all our programs, with the height of a bar representing the percentage of all tests for that category. In each program, we calculated the average percentage of tests of each test type (e.g., positive, active, no call-RA graph change) for each interesting statement category; then we averaged these averages across all programs. This calculation of an average of averages, repeated in several of our measurements reported in this section, we will call an *overall average*. Deleting a non-pointer assignment statement may not cause changes to the call-RA graph, because its lefthand-side variable could be assigned by other assignment statements in the same procedure. Deleting a call site may not affect the call-RA graph, if the changed procedure has more than one call site to the same called procedure. We can see that removing a non-pointer assignment statement is less likely to affect the call-RA graph and the *PMOD* solution than removing a statement of another category. Although function calls with pointer assignments only occupy a small percentage of the interesting statements, removing such a function call is most likely to cause changes to the call-RA graph and the *PMOD* solution.

The table in Figure 11 gives the overall average of the percentages of call-RA graph nodes with changed *CondIMOD*, call-RA graph edges with changed *nv_backbind*, call-RA graph edges deleted, and those added in a positive test. Since a non-pointer statement will not affect aliasing information, deleting such a statement changes only the set of variables modified by the containing procedure (i.e., *CondIMOD*). As for deleting a statement of the other three categories, edge deletion is the common structural change. Since a function call with pointer assignments is a combination of a function call and a pointer assignment, deleting such a call statement causes structural changes of the largest scale on the call-RA graph, as expected.

Rep. changes \ Test kinds	Non-pointer assignments	Pointer assignments	Fun. calls w/o ptr. assign.	Fun. calls w/ ptr. assign.
% CRAG nodes w/ diff. CondIMOD	0.64%	0.50%	0.50%	0.57%
% CRAG edges w/ diff. nv_backbind	0.00%	0.06%	0.03%	0.09%
% CRAG edges deleted	0.00%	4.67%	6.52%	7.11%
% CRAG edges created	0.00%	0.04%	0.08%	0.09%

Figure 11: Impact on the call-RA graph ($CRAG$) by test categories

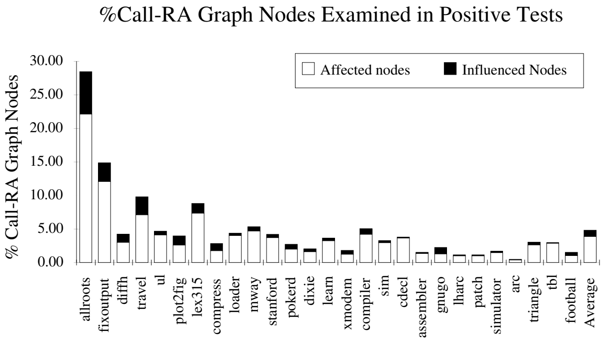


Figure 12: Impact on the *PMOD* solution of a positive test for each experiment program

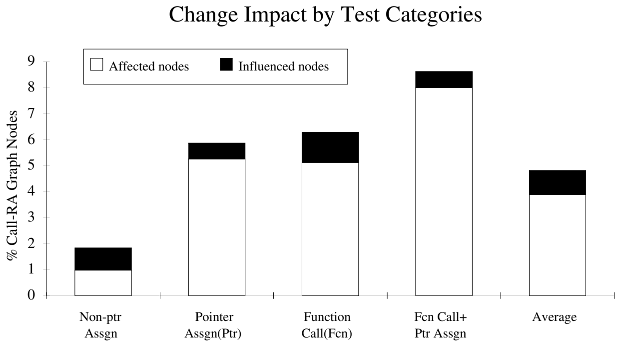


Figure 13: Impact on the *PMOD* solution of a positive test for each test category

In a test, a call-RA graph node is *affected* if its *PMOD* solution changes, and a call-RA graph node is *influenced* if one of its successors is affected, but the node itself is not. Figure 12 shows the the average percentages of call-RA graph nodes affected and influenced in a positive test for each program. The sum of these two percentages (i.e., the height of each bar) gives an estimate of the minimum percentages of call-RA graph nodes that must be examined by any safe incremental algorithm in order to update the *PMOD* solution. Except for the first few small programs, this value for most programs is below 6%, and the overall average is 4.80%(shown by the rightmost bar in the figure). This indicates that a minor source code change, such as deleting a statement, generally does not have great impact on the call-RA graph or the *PMOD* solution.

We have further examined the distribution of the number of affected and influenced nodes for each program, and found that it is not a normal distribution, but instead has a steep peak around a small number of nodes (i.e., 2-3). This indicates that a great percentage of the tests only affect a small number of call-RA graph nodes. Furthermore, the values given in Figure 12 are obtained by taking averages over positive tests only; thus if we included all tests in the averages, the height of the bars would be reduced by more than 50%. This result reveals great opportunities for our incremental technique.

Next, we examine the average impact of a positive test by test category. Figure 13 gives the overall average percentages of call-RA graph nodes affected and influenced, respectively, in a positive test for each test category. The impact of deleting a non-pointer statement is quite limited. Actually, if the variable modified by a non-pointer statement is a local variable, then it will just affect the *PMOD* solution of the containing procedure. However, if the modified variable is a global, the effect can be large, depending on the depth of the corresponding call-RA node in the graph. Deleting another category of statement may affect the *PMOD* solution to a larger extent.

Comparing Figures 11 and 13, we can see that a larger number of call-RA graph changes usually implies a larger number of affected or influenced call-RA graph nodes. This proportionality may open a door to finding an index that can be used to determine when it is worthwhile to update the solution incrementally, and when this will be tantamount to using an exhaustive approach.

CONCLUSIONS AND FUTURE WORK

We have presented an incremental algorithm for *PMOD*, a major subproblem of *MOD* analysis for C systems; our method includes updates for non-structural and structural changes and is based on the hybrid algorithm. Our instrumented studies of statement deletion using 27 C programs, show the greatly reduced cost of our incremental technique compared to an exhaustive analysis for side effects. In these studies, we have distinguished different types of assignments and function calls, showing their relative impact on solution updating. We have measured impact in terms of effect on our new program representation, the call-RA graph.

We are now implementing the incremental algorithm to profile its performance. We also are working on incremental approaches to the aliasing problem.

REFERENCES

[BR90] M. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[CK84] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258, June 1984. SIGPLAN Notices, Vol 19, No 6.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1985.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, January 1988.

[LR92] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[LRZ93] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[MR91] T. J. Marlowe and B. G. Ryder. Hybrid incremental alias algorithms. In *Proceedings of the Twentyfourth Hawaii International Conference on System Sciences, Volume II, Software*, January 1991.

[PS89] L. Pollock and M. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), December 1989.

[RMP88] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11:1–15, 1988.

[RP88] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[YR95] J. Yur and B.G. Ryder. Incremental analysis of the MOD problem for C. Laboratory for Computer Science Research Technical Report LCSR-TR-254, Department of Computer Science, Rutgers University, August 1995.

[Zad84] F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 132–143, June 1984. SIGPLAN Notices, Vol 19, No 6.