An Incremental Flow- and Context-sensitive Pointer Aliasing Analysis*

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

†Department of Computer Science
Rutgers University
Piscataway NJ 08855, USA
+1 732 445 4070
{yur,ryder}@cs.rutgers.edu

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

ABSTRACT

Pointer aliasing analysis is used to determine if two *object names* containing dereferences and/or field selectors, (e.g., `*p,q->t`), may refer to the same location during execution. Such information is necessary for applications such as data-flow-based testers, program understanding tools, and debuggers, but is expensive to calculate with acceptable precision. *Incremental algorithms* update data flow information after a program change rather than recomputing it *from scratch*, under the assumption that the change impact will be limited. Two versions of a practical incremental pointer aliasing algorithm have been developed, based on Landi-Ryder flow- and context-sensitive alias analysis. Empirical results attest to the time savings over exhaustive analysis (a six-fold speedup on average), and the precision of the approximate solution obtained (on average same solution as exhaustive algorithm for 75% of the tests.)

Keywords

Interprocedural pointer aliasing, incremental analysis, interprocedural side effect analysis.

1 INTRODUCTION

An *alias* occurs at some program point during program execution when two or more names refer to the same location. The precision of pointer alias information greatly affects its utility in applications such as data-flow-based testers, semantic program browsers, debuggers, and compilers. For example, *modification side effect analysis* [LRZ93, YRLS97] requires pointer aliasing information in the presence of assignments through dereferenced pointers, to find the set of variables that may be modified. Data-flow-based testers utilize definition-use information, needing aliases at uses of values obtained through dereferenced pointers.

Unfortunately, computing pointer aliasing information is expensive for languages with general-purpose pointers (such as C). The Precise solution of the pointer aliasing problem requires propagation of data flow information on interprocedural paths in a program. Intuitively, *flow sensitivity* means preserving statement order within a procedure, while *context sensitivity* means propagating information through a called procedure and then back again to the correct call site [LRS+98]. Many flow- and context-sensitive pointer aliasing analyses for C have been proposed [Coo89, LR92, CBC93, MLR+93, EGH94, WL95, Ruf95, HA90, SFRW90]. [1]

Incremental algorithms update data flow information after a program change rather than recomputing it *from scratch*, with the belief that the change impact will be limited and that the update cost will be proportional to the size of the impact. Given the expense of calculating aliases with sufficient accuracy to be useful, our goal was to develop an incremental aliasing algorithm with acceptable cost for use with interactive software tools. Empirical results show partial achievement of that goal.

Two incremental versions of the Landi-Ryder algorithm [LR92], a flow- and context-sensitive pointer aliasing analysis, are presented. Empirical tests used single statement deletions as sample source code changes on 13 moderately-sized C programs, obtaining updated alias solutions with a proof-of-concept prototype. Results show (i) on average over all the statement deletion which does cause impact on the aliasing solution, a *6-fold speedup* of incremental over exhaustive analysis (ii) in general, the *small impact* on the alias solution of a single source statement deletion (iii) *acceptable precision* (about 75% of tests had same solution from incremental and exhaustive algorithms). Our incremental approach allows the approximation of the Landi-Ryder algorithm to accumulate, and thus may compute a less precise solution than an exhaustive approach. These results show that incremental analysis will be substantially cheaper than exhaustive reanalysis with a limited sacrifice of the precision.

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

ICSE 99 Los Angeles, CA USA

[1] For other pointer aliasing algorithms, see [YRL98].

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], and some are a combination of these two techniques [MR90]. When a change is made, restarting iteration is a good mechanism for reusing the previously computed information. However, a precise solution for the updated problem instance will not be obtained unless some sufficient conditions are met [RMP88].

2 PROBLEM SPECIFICATION

The subset of C handled by our prototype includes sophisticated pointer usage and heap data structures, but not function pointers, and arrays are treated as aggregates.

A program is represented as an *Interprocedural Control Flow Graph* or $ICFG$ [LR91] which is the union of statement-level control flow graphs for each function in the program; each function has a unique *entry* and *exit* node. Each *call site* is split into *call* and *return* nodes. Edges are added to connect a call node to the entry node of the called function, and the exit node to the corresponding return node.

Memory locations are referred to through *object names.* An object name is a variable name with a (possibly empty) sequence of pointer dereferences (*) and/or field accesses (.field). Two object names, say o_1 and o_2, are aliased to each other, if they both may refer to the same memory location some time during execution; an unordered pair (o_1,o_2) is used to represent an alias, as in [LR91].

The *Interprocedural May Alias* problem computes the following set:

$\{< N, (o_1, o_2) > \mid \exists$ an interprocedural executable path, $\rho N_1 N_2 \cdots N_{i-1} N$, in the $ICFG$ on which o_1 and o_2 may be aliased at node $N\}$, where ρ is the entry of the program.

The Landi-Ryder aliasing algorithm conceptually divides this path into two paths, $\rho \cdots entry(N)$ and $entry(N) \cdots N$, where $entry(N)$ is the entry node of the function containing N. By doing this, the algorithm can solve separately for aliases at N under different calling contexts; thus, it obtains a context-sensitive solution by keeping calling context information with each alias and matching calling contexts during alias propagation. A calling context is approximated by a single *reaching alias,* AA, at $entry(N)$.

Only an approximate solution can be expected from a polynomial-time algorithm for pointer aliasing [LR91, Lan92]. We calculate *may_hold* [LR91] at each node of the $ICFG$ as an approximation of Interprocedural May Alias; that is, $may\_hold(N, AA, PA)$ is YES, if there exists a path in the $ICFG$ from ρ to $entry(N)$, which introduces reaching alias AA at $entry(N)$ **and** there also exists a path from $entry(N)$ to N which introduces alias PA at node N. The *may_hold* function is calculated by the Landi-Ryder algorithm and by our incremental algorithm.

A *safe solution* to Interprocedural May Alias is one which contains all the possible aliases that may occur at execution time and possibly, some spurious aliases.

---

procedure *exhaustive_aliasing*(G)
    G: an interprocedural control flow graph ($ICFG$);
begin
    /* 1. only performed implicitly */
  1. initialize *may_hold* with a default value NO;
     create an empty *worklist*;
  2. **for** each node N in G
     2.1 **if** N is a pointer assignment
           $aliases\_intro\_by\_assignment(N, YES)$;
     2.2 **else if** N is a call node
           $aliases\_intro\_by\_call(N, YES)$;
  3. **while** *worklist* is not empty
     3.1 remove (N, AA, PA) from *worklist*;
     3.2 **if** N is a call node
           $alias\_at\_call\_implies(N, AA, PA, YES)$;
     3.3 **else if** N is an exit node
           $alias\_at\_exit\_implies(N, AA, PA, YES)$;
     3.4 **else for** each $M \in successor(N)$
           3.4.1 **if** M is a pointer assignment
                   $alias\_implies\_thru\_assign(M, AA, PA, YES)$;
           3.4.2 **else**
                   $make\_true(M, AA, PA)$;
end

Figure 1: Exhaustive algorithm for *pointer aliasing*

---

3 EXHAUSTIVE ALIASING ANALYSIS

We intuitively and briefly explain the Landi-Ryder approximate algorithm [LR92] for interprocedural pointer aliasing in Figure 1 as having three main steps.[2] Firstly, the function *may_hold* is initialized, and an empty *worklist* for iteration is created. Secondly, new aliases are introduced through pointer assignments and parameter bindings at a call site. Finally, a fixed point iteration using a worklist is performed in step 3 to find more aliases that may be generated at assignments based on the introduced aliases, or interprocedurally through function calls/returns. During one step in the iteration, an alias tuple is removed from the worklist, and propagated to successor nodes by invoking appropriate

[2] For a more complete description of the Landi-Ryder algorithm, see [LR92].

handling procedures[3], which correspond to applying the semantics of a respective node to an alias to obtain the set of aliases it implies at successor nodes. Procedures *alias_at_call_implies* and *alias_at_exit_implies* handle the matching of the calling contexts and the mapping of the aliases between the calling and called functions. When a new alias is generated at a node, function *make_true* (in step 3.4.2) will set its *may_hold* value to YES, and put it on the worklist. After the iteration terminates, $may\_hold(N, AA, PA) = YES$ indicates that (N, AA, PA) is contained in the solution of the *Approximate Interprocedural May Alias* problem.

4 INCREMENTAL ALIASING ANALYSIS

When a change is made to the program, an incremental algorithm based on the iterative technique usually has the following two phases in order to obtain the *incrementally precise*[4] solution [BR90, RMP88]: *reinitialize* the solution at potentially affected program points, and *restart iteration*. That is, before restarting iteration, a reinitialization of the region of the program to which the effects of a source change can extend, typically called the *affected region*, is taken to retain a *safe initial estimate*[5] for the changed program. The effectiveness of an incremental algorithm relies on the belief that the affected region rarely covers the whole program, and is frequently limited to a small part of the program. In practice, since the actual affected region is not known *a priori*, a too conservative approximation could result in *unnecessary* reinitialization to a large part of the program, and then cause redundant re-computation at those program points which are not actually affected. An incremental algorithm may achieve a better performance improvement by reducing unnecessary reinitialization, and reusing the old solution as much as possible.

A high level description of our incremental aliasing algorithm for handling a small source change, such as adding or deleting a single statement, is given in Figure 2. The algorithm is composed of four main steps. When a source change (statement addition or deletion) is made, the first step (step 1 in Figure 2), called *alias falsification*, removes (or $falsifies$) the *affected aliases* that are either generated at that statement or dependent on other affected aliases. This step basically performs the task of reinitialization so as to obtain a safe initial

procedure *incremental_aliasing*(G,N)
  G: an $ICFG$;
  N: a statement to be changed;
begin
  1. falsify the affected aliases, which are either generated at N, or depend on other affected aliases.
  2. update G to reflect the change to statement N;
  3. *worklist*=*reintroduce_aliases*(G);
  4. *reiterate_worklist*(*worklist*,YES);
end

Figure 2: Incremental aliasing algorithm for handling addition/deletion of a statement

estimate for the subsequent iteration that computes the updated solution. The structure of the $ICFG$ is updated (in step 2) to reflect the change. The third step (step 3 in Figure 2), called *alias reintroduction*, adds the relevant aliases onto a worklist, and the *reiteration* phase (step 4) will then iterate to the final aliasing solution.

For this approach, the strategy of falsification directly affects the performance of the incremental algorithm. A straight-forward way of falsification is to falsify the entire old aliasing solution, but then the incremental algorithm degenerates to an exhaustive algorithm. We propose two strategies of falsification: *naive falsification* and *selective falsification*. *Naive falsification* falsifies aliases by only following control flow. *Selective falsification* further uses data dependences to reduce the set of aliases it falsifies.

Naive Falsification

The idea behind the naive falsification is to falsify all the aliases at the control flow successors (intra- and interprocedurally) of the changed statement. In procedure $naive\_falsification$ in Figure 3, a forward walk in the $ICFG$ is taken from the changed statement. At each node visited, we assume that all the aliases at the visited node may be affected, and just discard them (step 2) and mark the node *TOUCHED* (step 3). When an exit node of a function is encountered, the walk is resumed at the return node of each call to the function (step 4), and we keep discarding aliases until we reach the exit of the main program. In order to "save" some aliases which may be reusable in the later updating phase, when a call is visited in the falsification, instead of discarding the aliases in the called function, function *disable_aliases* in Figure 3 just *disables* all the reaching aliases at the entry of the function (step 2), and marks the function *INFLUENCED* (step 3). That is, those aliases that are disabled are still *memorized*. The alias disabling will proceed recursively to other functions called by the current function (step 4).

Since a calling context (i.e., a reaching alias at function

[3] These procedures have been augmented with one extra parameter which is the value of the *may_hold* function to be set for the implied aliases. With YES as the argument (as in Figure 1), these handling functions will *generate* aliases by setting their *may_hold* to YES.

[4] A solution obtained by an incremental algorithm is *incrementally precise* if it is the same as the one obtained by the corresponding exhaustive algorithm applied to the changed program.

[5] For the pointer aliasing problem, an initial estimate is *safe*, if we can be sure that it should be contained in any safe aliasing solution.

```
/* Alias falsification corresponding to step 1 in Figure 2 */
procedure naive_falsification(N)
    N: a statement to be changed;
begin
    1. if N is marked TOUCHED, return;
       /* Falsify aliases at the changed node N */
    2. set all may_hold(N, AA, PA) to NO;
    3. mark N TOUCHED;
    4. if N is an exit node
          for each call node C which calls the function
          containing N;
              naive_falsification(corresponding return of C);
    5. else if N is a call node
       5.1  disable_aliases(entry of the function called by N);
       5.2  naive_falsification(corresponding return of N);
    6. else for each M ∈ successor(N)
                naive_falsification(M);
end

procedure disable_aliases(E)
    E: entry of the function whose aliases will be disabled;
begin
    1. if E is marked INFLUENCED, return;
    2. set all may_hold(E, AA, AA) to FALSIFIED;
    3. mark E INFLUENCED;
    4. for each call node C in function E;
           disable_aliases(entry of the function called by C);
end
```

Figure 3: Naive falsification

```
/* Alias reintroduction corresponding to step 3 in Figure 2 */
procedure reintroduce_aliases(G)
    G: an ICFG;
return
    a worklist for keeping the reintroduced aliases;
begin
    1. create an empty worklist;
       /* Inter-procedural propagation */
    2. for each call node C in G
       2.1  if C is TOUCHED or its called function is
              INFLUENCED,
              2.1.1   aliases_intro_by_call(C, YES);
              2.1.2   repropagate_aliases(C, worklist);
       /* Intra-procedural propagation */
    3. for each TOUCHED node N in G
       3.1  if N is a pointer assignment statement,
               aliases_intro_by_assignment(M,YES);
       3.2  for each M ∈ predecessor(N)
                 repropagate_aliases(M,worklist);
    4. return worklist;
end

procedure repropagate_aliases(N,worklist)
    N: a program node in the ICFG;
    worklist: a worklist for keeping the reintroduced aliases;
begin
    for each may_hold(N, AA, PA) = YES
         add (N, AA, PA) to worklist;
end
```

Figure 4: Reintroduce aliases for naive falsification

entry) may now be disabled, whether or not a tuple (N, AA, PA) is in the aliasing solution can not be determined solely by the value of *may_hold*(N, AA, PA), but in additional by *may_hold*(E, AA, AA) where E is the entry of the function containing N. That is, a tuple (N, AA, PA) is in the aliasing solution if both *may_hold*(N, AA, PA) and *may_hold*(E, AA, AA) are YES. On the other hand, if *may_hold*(E, AA, AA) is $FALSIFIED$ (i.e., alias AA is disabled at entry E), then all the aliases with AA as the reaching alias at nodes within the corresponding function are *implicitly* disabled, and will not appear in the aliasing solution.

After the naive falsification, all the aliases at $TOUCHED$ nodes are discarded (i.e., falsified), and those reaching the entries of *INFLUENCED* functions are disabled. The $ICFG$ then is modified to reflect the change to the source program. Iteration may now be restarted to update the aliasing solution at the $TOUCHED$ and *INFLUENCED* nodes. Before reiteration, we have to reintroduce aliases (step 3 in Figure 2) to the worklist. Aliases will be reintroduced within functions and/or across functions. For reintroducing aliases across functions, procedure *reintroduce_aliases* (in Figure 4) examines each call site. If either a call site is $TOUCHED$ or its called function *INFLUENCED*, the aliases at the call site need to be repropagated; that is, the effect of the call is reintroduced (step 2.1.1), and the aliases at the call site are repropagated (step 2.1.2). For reintroducing aliases within functions, the effect of a $TOUCHED$ pointer assignment is reintroduced (step 3.1) and aliases are repropagated from each of its predecessors (step 3.2).

Next, iteration is restarted over the worklist obtained by the alias reintroduction step. Procedure *reiterate_worklist* in Figure 5 is quite similar to the iteration (step 3) in Figure 1, except it will try to reuse the previously existing but disabled aliases in a *INFLUENCED* function by calling *aliases_propagated_at_call* instead. Since procedure *reiterate_worklist* will also be used for the selective falsification strategy (mentioned later), parameter *value* is set YES (or $FALSIFIED$) to generate (or to falsify) aliases. Here, *value* is set to YES.

When a call node is encountered in the reiteration, procedure *aliases_propagated_at_call* in Figure 6 will check whether the implied alias reaching the entry of the called function existed before. If not, it will be handled as a new reaching alias (step 2.1). Otherwise, the alias is "enabled" (step 2.2.1), and this *implicitly* enables all

/* Reiteration corresponding to step 4 in Figure 2 */
procedure *reiterate_worklist*(*worklist*,*value*)
  worklist: a worklist for keeping the aliases to process;
  value: value that will be given to (N,AA,PA);
begin
  1. **while** *worklist* is not empty **do**
    1.1 remove (N,AA,PA) from *worklist*;
    1.2 **if** N is a call node
        aliases_propagated_at_call(N, AA, PA, *value*);
    1.3 **else if** N is an exit node
        alias_at_exit_implies(N, AA, PA, *value*);
    1.4 **else for** each $M \in$ *successor*(N)
        1.4.1 **if** M is a pointer assignment
            alias_implies_thru_assign(M, AA, PA, *value*);
        1.4.2 **else if** *value* is YES
            make_true(M, AA, PA);
        1.4.3 **else** /* *value* is $FALSIFIED$ */
            make_false(M, AA, PA);
end

Figure 5: Reiteration for the incremental algorithm

the other aliases at the *un*$TOUCHED$ nodes of the function which depend on that reaching alias. Procedure *inter_proc_propagate* is then called (step 2.2.2) to recursively enable other reaching aliases of those functions called by the function. This will save the incremental algorithm from "recomputing" those aliases which existed before and are still reusable. In presence of recursion, when a reaching alias is enabled at a function containing $TOUCHED$ nodes, the previously disabled aliases which depend on that reaching alias are repropagated to those $TOUCHED$ nodes in the procedure.

The reuse of previously disabled aliases may introduce imprecision in the updated aliasing solution, which will be discussed at the end of Section 4.

Selective Falsification

Next we are going to refine our incremental algorithm by using a better falsification strategy, *selective falsification*, which only falsifies aliases that are actually generated by the changed statement and other aliases implied by those aliases. This strategy reduces the set of aliases to be falsified, but at the price of extra computation. The procedures corresponding to steps 2-4 of the incremental algorithm (in Figure 2) do not require changes.

In selective falsification, the set of initially affected aliases which are directly generated at the changed node is computed. Then an iteration finds other aliases which may depend on those initial aliases. Finding the set of initially affected aliases is not a trivial task. Including too many irrelevant aliases will cause unnecessary computation not only in the falsification phase, but also in the reiteration phase. Next, we consider two kinds

procedure *aliases_propagated_at_call*(N,AA,PA,*value*)
  N: a call node;
  AA: reaching alias at the entry of the function containing N;
  PA: possible alias at N;
  value: value to set the propagated aliases;
begin
  1. **let** E be the entry of the function called by N, and R the corresponding return node of N;

  /* aliasing effect propagated to the entry node E */
  2. **for** each AA' in *bind*(N, E, PA)
    /* *bind* uses parameter bindings to map PA to the entry E of the called function */
    2.1 **if** (E, AA', AA') has not been seen before
        make_true(E, AA', AA');
    2.2 **else if** *may_hold*(E, AA', AA') $\neq$ *value*
      2.2.1 set *may_hold*(E, AA', AA') to *value*;
          /* Recursively enable (or disable) all the reaching aliases implied at the entry of other functions reachable from E */
      2.2.2 *inter_proc_propagate*(E, AA', *value*);

  /* aliasing effect propagated to the return node R */
  3. (Same as what is done for propagating aliases to the return node in procedure *alias_at_call_implies*, except it will make_true or make_false the implied aliases, depending on what *value* is)
end

procedure *inter_proc_propagate*(E,AA,*value*)
  E: a function entry;
  AA: a previously existing reaching alias at E;
  value: new value for AA at E, and the reaching aliases implied at other reachable functions from E;
begin
  1. **for** each call node C in the function
    1.1 **let** E' be the entry of the function called by C;
    1.2 **for** each (C, AA, PA)
        for each AA' in *bind*(C, E', PA), such that *may_hold*(E', AA', AA') $\neq$ *value*
          1.2.1 set *may_hold*(E', AA', AA') to *value*;
          1.2.2 *inter_proc_propagate*(E', AA', *value*);

  /* Repropagate the aliases implicitly enabled by the reaching alias AA to the $TOUCHED$ nodes */
  2. **if** *value* $= YES$ /* AA is enabled at E */
      for each $TOUCHED$ node N in the function
        for each $M \in$ *predecessor*(N)
          for each *may_hold*(M, AA, PA') $= YES$
            add (M, AA, PA') to the *worklist*;
end

Figure 6: Procedures for propagating aliases among functions

of atomic source change respectively: deleting a statement, and adding a statement.

Deleting A Statement
The computation of the initially affected aliases depends on the type of statement being deleted.

```
/* Alias falsification for deleting a pointer assignment
   corresponding to step 1 in Figure 2 */
procedure falsify_for_deleting_assign(N)
   N: a pointer assignment to be deleted;
begin
   1. create an empty worklist;
      /* Falsify the aliases introduced at statement N. */
   2. aliases_intro_by_assignment(N, FALSIFIED);
   3. for each M ∈ predecessor(N)
         for each may_hole(M, AA, PA = (o1, o2)) = YES
            if the left-hand side of N is a prefix6 of either
            o1 or o2, or both
               alias_implies_thru_assign(N, AA, PA,
               FALSIFIED);
   4. reiterate_worklist(worklist, FALSIFIED);
end

procedure falsify_for_deleting_call(N)
   N: a function call to be deleted;
begin
   1. create an empty worklist;
      /* Falsify the aliases introduced by the call */
   2. let E and X be the corresponding entry node and
      exit node of the function called by N respectively;
   3. aliases_propagated_at_call(N, ∅7, ∅, FALSIFIED);
   4. for each may_hold(N, AA, PA) = YES
         /* If the called function may generate new aliases
            from the reaching aliases implied by PA */
         if ∃ AA' ∈ bind(N, E, PA), such that some
         PA' (≠ AA') is generated from AA' at exit X
            aliases_propagated_at_call(N, AA, PA,
            FALSIFIED);
   5. if a function becomes unreachable from the main pro-
      gram after the call node is deleted, steps 3 and 4
      are repeated on those calls within each of the
      reachable functions.
   6. reiterate_worklist(worklist, FALSIFIED);
end
```

Figure 7: Procedures for falsifying aliases which are potentially affected by deleting a statement

Deleting a pointer assignment If a pointer assignment is to be deleted, procedure *falsify_for_ deleting_assign* (in Figure 7) is taken to falsify the aliases that may be affected by the assignment. First, step 2 falsifies the aliases introduced at the pointer assignment (i.e., generated independently of the reaching aliases). With *FALSIFIED* as the last argument, the procedure *aliases_intro_by_assignment* will falsify (i.e., set the *may_hold* value to *FALSIFIED*) the aliases the pointer assignment introduces. Step 3 then passes all the reaching aliases to the statement from its predecessors, and falsifies the implied aliases, except those which are preserved. An alias $PA = (o_1, o_2)$ is preserved if the left-hand-side of the pointer assignment statement is not a *prefix*[6] of either o_1 or o_2; this is what the *if* statement in step 3 does. After steps 2 and 3, the *worklist* will contain the aliases that the pointer assignment affects initially. Step 4 is an iteration step to falsify more aliases at other program nodes which depend on those initially affected aliases. With *FALSIFIED* as the argument for *value*, procedure *reiterate_worklist* in Figure 5 will selectively *make_false* (step 1.4.3) aliases it finds during its traversal in the *ICFG*. Procedure *make_false*(M, AA, PA) not only sets *may_hold*$((M, AA, PA)$ to *FALSIFIED*, but also marks node M as *TOUCHED* (which is not mentioned in the code). If a call node is visited during the traversal (step 1.2 in Figure 5), the procedure *aliases_propagated_at_call* (in Figure 6) will selectively disable the implied aliases at the entry of the called function and mark the function *INFLUENCED*. The alias-disabling [8] will be performed recursively on other functions called by this function by execution of *inter_proc_propagate* at step 2.2.2 in Figure 6.

Deleting a function call To determine which aliases may be directly affected by the call and need to be falsified, procedure *falsify_for_ deleting_call* (in Figure 7) examines each of the aliases reaching the call (in step 4). If the called function does imply new aliases based on that alias, then procedure *aliases_propagated_at_call* is called to disable the reaching aliases implied at the entry of the called function, and to falsify the implied aliases at the corresponding return node. Since the deletion of a function call may cause some functions which are reachable originally to become unreachable, those calls within these unreachable functions to other reachable functions should also be treated as deleted. Steps 3 and 4 are repeated on those calls (in step 5).

Adding A Statement
While the alias falsification for deleting a statement is to falsify the aliases that may be generated at the statement, the alias falsification for adding a statement is to falsify the aliases that will not be preserved by the statement and other aliases implied by those initially affected aliases.

Adding a pointer assignment To be able to falsify the aliases that will not be preserved by the to-be-added pointer assignment, we first need to obtain its alias-

[6] An object name o_1 is a *prefix* of o_2 iff o_1 can be transformed to o_2 by a sequence of dereferences and field accesses.

[7] $\emptyset$ indicates that no aliases are assumed to reach at the entry.

[8] The disabling step is the same in both falsification strategies; they only differ how to determine which reaching aliases to disable.

procedure $update\_for\_adding\_assign(N,M)$
N: a pointer assignment to be added;
M: the statement after which statement N is added;
begin
1. make N as a successor of M, and leave N without any successors;
2. create an empty *worklist*;
3. $aliases\_intro\_by\_assignment(N, YES)$;
4. $repropagate\_aliases(M, worklist)$;
5. $reiterate\_worklist(worklist, YES)$;
6. **for** each $may\_hold(M, AA, PA = (o_1, o_2)) = YES$, and $may\_hold(N, AA, PA) = NO$
   add (M, AA, PA) to *worklist*;
7. $reiterate\_worklist(worklist, FALSIFIED)$;

end

Figure 8: Procedure for falsifying aliases that are potentially affected by adding a pointer assignment

ing effect. Thus the first part (step 1 to 5) of procedure *update_for_adding_assign* in Figure 8 is to compute the effect of the newly added pointer assignment on the aliases propagated from its predecessor. When a pointer assignment N is to be added between node M and all its successors, in step 1, node N is added as a successor of node M, but left without successors. An empty *worklist* is created at step 2. The aliases which will be introduced at node N and those propagated from its predecessor M are added to *worklist* in steps 3 and 4 respectively. Though step 5 calls for an iteration, it is basically for computing the effect of node N on the aliases that are propagated to it. An alias PA at node M that is not preserved at node N is added to *worklist* (in step 6) for falsification. An iteration is then performed at step 7 to falsify more aliases at other nodes which directly or indirectly depend on those initially affected aliases found at step 6.

Adding a function call When adding a function call, we first compute the aliases that will be introduced by the call and those that will be generated from the aliases propagated to the call. Then we compare the previous and new aliasing solutions at the to-be-added call to find the initially falsified aliases, and an iteration is performed to falsify other aliases which depend on the initially falsified aliases.

If the called function of the to-be-added call recursively calls the changed function, the aliasing effect of the call and the changed function will be interdependent on each other. That is, those aliases the call preserves will depend on the return effect of the changed function, and which recursively depends on the alias-preserving effect of the call. Such interdependence may make the falsifying procedure miss some aliases that are supposed to be falsified, and thus introduce more spurious aliases in the final solution. Thus, if the call to be added recursively calls the changed function, we will just use the naive falsification to ensure the precision of the updated aliasing solution. Further details and complete algorithms are presented in [YRL98].

Program	LOC	ICFG nodes	Alias time	#*may_hold*	#Ptr tests	#Func tests
loader	1219	1536	3.4	24255	29	22
sim	1439	3034	8.0	75622	63	26
cdecl	1015	3196	17.3	148722	46	30
diff	1726	3300	12.7	111360	60	62
unzip	4106	3416	8.4	72892	51	32
livc	1891	4101	3.3	22624	27	8
lharc	3303	4250	12.6	111768	87	137
patch	2672	4608	11.8	95420	83	186
simulator	3733	5574	10.2	72724	47	167
arc	7507	5856	19.8	168474	104	118
triangle	1930	6119	6.1	48913	75	13
tbl	2511	6162	55.8	530880	120	97
moria	24596	38572	52.8	271725	535	1756

Figure 9: Experiment dataset

Potential Imprecision
One source of approximation in the Landi-Ryder's algorithm for the Interprocedural May Alias problem is that the *may_hold* function at a node takes one alias, instead of a set, as the reaching alias to the entry of the function containing that node. Thus, if two aliases, say PA_1 and PA_2, are propagated to node n from its predecessor m (that is, both $may\_hold(m, AA_1, PA_1)$ and $may\_hold(m, AA_2, PA_2)$ hold), and they together imply PA at node n, then either $may\_hold(n, AA_1, PA)$ or $may\_hold(n, AA_2, PA)$, but not both, will be generated [LR92]. Since the incremental algorithm tries to reuse as much of the old aliasing solution as possible, for the above example, if AA_1 is chosen as the reaching alias for the generated PA at node n, and after changes, AA_1 is still implied at the entry of the containing function, then $may\_hold(n, AA_1, PA)$ will hold in the updated solution obtained by our incremental algorithm even if the other reaching alias AA_2 may not exist at the function entry after changes. This is an overestimating approximation that is a safe solution. In the following section, we will show that such overestimation is very limited.

5 EMPIRICAL EFFECTIVENESS OF INCREMENTAL ANALYSIS

To demonstrate the effectiveness of the incremental approach, we studied 13 programs and measured the execution time of the incremental algorithm implemented in a prototype on a 75 MHz Sun SPARCstation-20 running Solaris 2.5.1 with 352 Mbyte physical memory and 250 Mbyte swap space. For each program in our dataset, we first determined the *interesting* source statements of two types, pointer assignment[9] statements and *affecting*

[9] If a library function call can have aliasing side effects, it is encoded as a sequence of assignment statements.

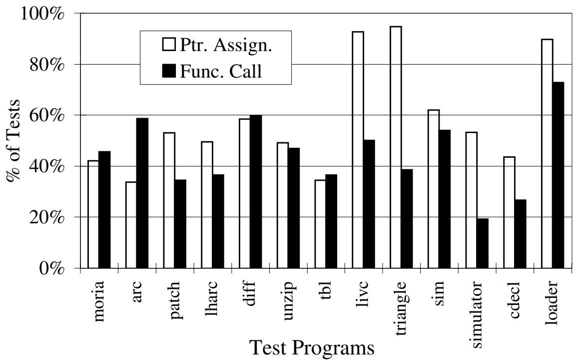

Figure 10: Percentage of tests that are active

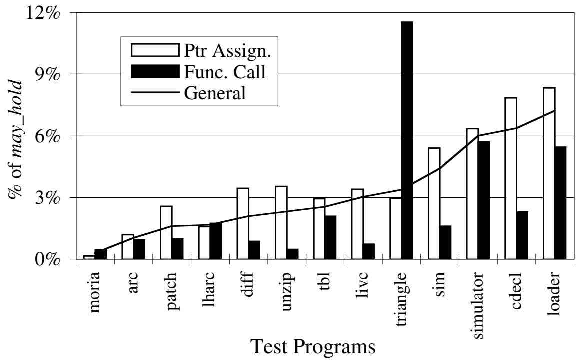

Figure 11: Impact on the aliasing solution

function calls, which are calls to functions containing pointer assignments directly or indirectly, which thus can potentially affect aliases. Each test corresponded to deletion of one interesting source statement from the original program. In each test, we calculated the aliasing solution for the program exhaustively, recalculated it incrementally after deleting the statement, and then also recalculated it exhaustively on the changed program. The programs we tested are listed in Figure 9, ordered by the number of $ICFG$ nodes, where LOC denotes lines of code; *Alias time* the execution time (in seconds) for exhaustively computing the conditional aliasing solution; *#may_hold* the solution size; and $\#Ptr$ and $\#Func$ the numbers of tests on pointer assignments and affecting functions calls respectively. These data are based on the original (i.e., unchanged) program.

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 obtain statistically significant results.

Not every interesting statement deletion will cause changes to the aliasing solution; a test is called *active* if it causes changes to the aliasing solution at some program points other than the deleted statement. Deletion of an affecting function call may cause those functions called directly or indirectly to become unreachable. Since the aliasing solution at the unreachable functions can simply be discarded due to their unreachability, a test on an affecting function call is *active* only if it causes changes to the aliasing solution in some reachable function. Figure 10 shows the percentages of tests which are active for pointer assignments and affecting function calls respectively. Deletion of an affecting function call does not appear more likely to be active than deletion of a pointer assignment; this may be because most of the aliasing effect of a called function is on its local variables, and is not propagated to the calling functions.

For each program, we measured the average impact on the aliasing solution of deleting one interesting statement by calculating the average number of changed aliases as a percentage of the total number of aliases for the original program. Figure 11 gives the percentages for active tests on the two types of interesting statements as bars. It also shows a *general average* across all programs as a polygonal line; each of the 13 points on the line represents the average percentage impact over all interesting tests for a particular program. For example, for `triangle`, 3% of the aliases are affected on average by deletion of an active pointer assignment, whereas 11.5% are affected by deletion of an active affecting function call, and the general average is 3.4%. The *overall average*, or average of the 13 points mentioned above, is about 3.2%; this indicates that a minor source code change, such as deleting a statement, generally does not have great impact on the aliasing solution. By averaging over each interesting statement type separately, we find that deleting a pointer assignment usually causes a greater impact (3.6%) on the aliasing solution than deleting an affecting function call (2.4%). Usually, the impact on the aliasing solution of a source change in a small program is greater than that in a large program. However, another factor is sometimes more dominant than the program size, namely how the aliases are created and propagated in the program. In `triangle`, most of the active tests on affecting function calls are calls to some functions which create or manipulate aliases on some global data structures in the `main` function. Thus, deleting such a function call will cause greater impact on the aliasing solution.

To demonstrate the time savings of our incremental algorithm, we calculated the *relative execution time* spent by the incremental algorithm using the two falsification strategies (naive and selective) to update the aliasing solution as a percentage of the execution time for recomputing the updated aliasing solution exhaustively. The bar chart in in Figure 12 shows the average relative execution times of the naive and selective incremental algorithms in an active test, respectively. The selective incremental algorithm is superior to the naive

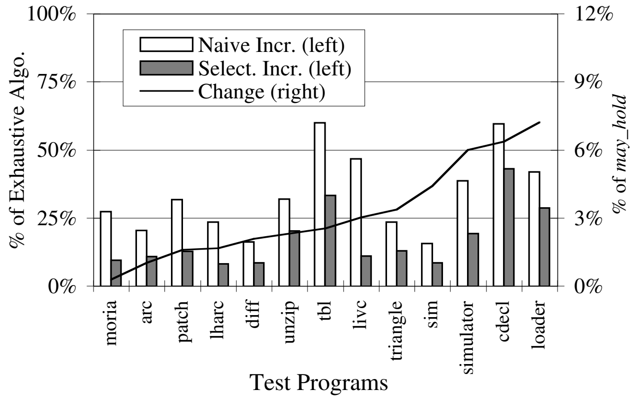

Figure 12: Execution time of the incremental algorithms

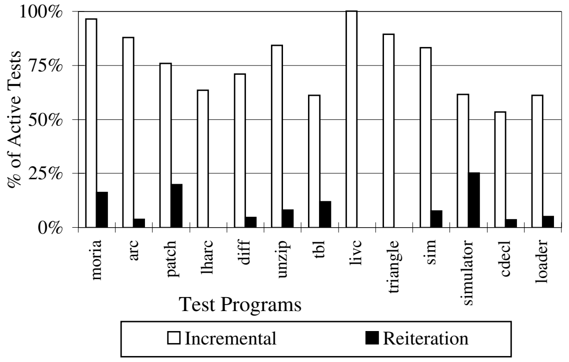

Figure 13: Percentage of active tests which are hits

one. While the overall average for the naive incremental algorithm is 33%, that for the selective incremental algorithm is about 17%, which is a 6 times speedup over the exhaustive algorithm. For large test programs like `moria`, the average speedup is as high as 11. In the same figure, the average impact on the aliasing solution (shown as a line scaled with the right y-axis) is superimposed for comparison. We have observed that the execution time of our incremental algorithms does not show strong correlation with the size of impact, but instead, with how the aliases are used and propagated.

A solution obtained by an incremental algorithm is a *hit*, if it is the same as the solution computed by the exhaustive one; the *hit rate* for a program is the percentage of tests in which the incremental algorithm achieves a hit, a measure of precision. Figure 13 shows the hit rates for our proposed incremental algorithm[10] versus reiteration without any reinitialization (which also computes a safe but less precise solution); the average hit rates are 75% and 8% respectively. In most of the tests, the incremental algorithm computes the same solution as the exhaustive one.

The empirical results demonstrate that the incremental algorithm achieves a multi-fold speedup over the exhaustive algorithm, and the sacrifice of the precision is limited compared with the reiteration without reinitialization.

[10] The naive and selective incremental algorithms will demonstrate the same hit rate.

6 DISCUSSIONS AND CONCLUSIONS

Multiple Changes The incremental algorithms mentioned in this paper are for handling addition/deletion of one single statement. Although any source change can be handled by applying the relevant algorithm, in the presence of multiple source changes, better efficiency can be achieved by processing multiple changes as a whole. That is, instead of applying the relevant algorithms to handle the changes individually, we can combine the falsification phase of the required algorithms, reintroduce aliases from the affected nodes, and then update the aliasing solution in one single iteration. Since multiple changes usually exhibit locality, the sets of affected nodes for these changes may have a great overlap, and combining corresponding phases may avoid multiple passes of falsification and reiteration of the common affected aliases.

Demand-Driven Analysis While the complete data-flow analysis computes the information at all program points, a demand data-flow analysis determines whether a given data-flow fact holds at a given point [BJ78, Rep94, DGS95]. The Landi-Ryder aliasing algorithm is a whole-program analysis; we are not seeking to derive a demand-driven version from it, but to handle changes and queries on a demand-driven basis. After a source change is made, if we are interested in the possible impact of the change on the aliasing solution at a particular program node, the incremental algorithms may meet the need with minor modifications. Details can be found in [YRL98].

Applicability to Other Flow- and Context-sensitive Algorithms Although, our incremental algorithm is based on the Landi-Ryder algorithm, the framework can be adapted to incrementalize other flow- and context-sensitive data flow algorithms, which tag the data flow information with a representation of calling context. The general approach involves (i) obtaining a safe initial estimate for the to-be-updated solution, (ii) using an efficient worklist algorithm for calculating the updated solution and (iii) performing delayed updates within called functions, by disabling possibly affected calling contexts and sometimes subsequently reusing memorized values.

Conclusions and Future Work We have incrementalized a flow- and context-sensitive pointer aliasing algorithm to handle deletion and addition of single statements in a C program. The technique can be extended to sets of statement changes as well. The studies of deleting pointer assignments and affecting function calls using 13 C programs, show the promise of the incremental algorithm. On average, a 6-fold speedup of incremental over exhaustive analysis was observed. The relatively small impact (overall average 3.2% of all aliases in unchanged program) of a single source statement deletion was documented. Acceptable precision (on average solution agreement on 75% of tests) was demonstrated.

Our next step is to study the behavior of real-world changes and to improve the efficiency of the incremental algorithm in handling multiple changes. Further, since the incremental algorithm may not always save time, especially when the effect of a change is large, we want to find a measurement which can be used to determine when using the incremental algorithm will be worthwhile.

REFERENCES

[BJ78] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis, part II: Demand analysis. *Acta Informatica*, 10:265–272, 1978.

[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.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[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.

[Coo89] B. G. Cooper. *Ambitious Data Flow Analysis of Procedural Programs.* Master's thesis, University of Minnesota, May 1989.

[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.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1995.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–257, June 1994. Published as SIGPLAN Notices, 29 (6).

[HA90] W. L. Harrison III and Z. Ammarguellat. Parcel and Miprac: parallelizers for symbolic and numeric programs. In *Proceedings of International Workshop on Compilers for Parallel Computers*, pages 329–346. Ecole des Mines de Paris - CAI, UPMC - Laboratoire MASI, December 1990. Paris, France.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[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.

[LRS+98] W. Landi, B. G. Ryder, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report DCS-TR-336, Department of Computer Science, Rutgers University, May 1998. submitted for journal publication.

[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.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 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.

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

[Rep94] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the Fifth International Conference on Compiler Construction*, pages 389–403, April 1994. Appeared as Lecture Notes in Computer Science, Vol 786.

[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.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis. In *Proceedings of the Second International Workshop in Programming Language Implementation and Logic Programming*, pages 277–292, August 1990. Volume 456 of Lecture Notes in Computer Science.

[WL95] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. also available as SIGPLAN Notices, 30(6).

[YRL98] Jyh-shiarn Yur, Barbara G. Ryder, and William. A. Landi. Incremental algorithms and empirical comparison for flow- and context-sensitive pointer alias analysis. Department of Computer Science Technical Report DCS-TR-348, Department of Computer Science, Rutgers University, August 1998. in preparation.

[YRLS97] J. Yur, B. G. Ryder, W. Landi, and P. Stocks. Incremental analysis of side effects for C software systems. In *Proceedings of the Nineteenth International Conference on Software Engineering*, pages 422–432, May 1997.

[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.