Incremental and Demand-driven Points-To Analysis Using Logic Programming

Diptikalyan Saha
dsaha@cs.sunysb.edu

C. R. Ramakrishnan
cram@cs.sunysb.edu

Department of Computer Science
University of Stony Brook
Stony Brook, NY 11794-4400.

ABSTRACT

Several program analysis problems can be cast elegantly as a logic program. In this paper we show how recently-developed techniques for incremental evaluation of logic programs can be refined and used for deriving practical implementations of incremental program analyzers. Incremental program analyzers compute the changes to the analysis information due to small changes in the input program rather than re-analyzing the program. Demand-driven analyzers compute only the information requested by the client analysis/optimization. We describe a framework based on logic programming for implementing program analyses that combines incremental and demand driven techniques. We show the effectiveness of this approach by building a practical incremental and demand-driven context insensitive points-to analysis and evaluating this implementation for analyzing C programs with 10-70K lines of code. Experiments show that our technique can compute the changes to analysis information due to small changes in the input program in, on the average, 6% of the time it takes to reanalyze the program from scratch, and with little space overhead.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis; D.1 [**Programming Techniques**]: Logic Programming

General Terms

Algorithms, Performance, Design, Languages, Theory

Keywords

Pointer analysis, Incremental analysis, Demand-drive analysis, Logic programming

1. INTRODUCTION

Many program analysis problems can be naturally formalized by a set of inference rules. For instance, Anderson's points-to analysis [3] can be represented by a set of inference rules [20] (see Figure 1). In many cases (Anderson's analysis being one of them), the inference rules can be readily encoded as a logic program (e.g. see Figure 2). The solution to the analysis problem can then be obtained from the least (or in some cases, the greatest) relation satisfying its defining clauses.

While Horn clause logic has been used as a framework to specify many analysis problems, logic programming systems have been seldom used to derive practical analyzers directly from these specifications. *Tabled evaluation* [38, 10], which is complete for Datalog programs, has offered the promise to derive implementations of analyzers from specifications [41]. However, declarative programming technology has not been used to offer benefits, other than the elegance and simplicity, for the construction of program analyzers.

In this paper we describe a concrete instance where declarative programming technology significantly improves the state of the art in the implementation of program analyzers. We combine our recent development of an incremental evaluation technique for logic programs with the demand-driven query evaluation capabilities of tabling to derive incremental program analyzers. We refine our incremental evaluation technique so that the resultant analyzers are efficient and scalable. For instance, our implementation of Anderson's points-to analysis for C programs, which is a flow- and context-insensitive analysis[1], scales well to programs with nearly 70K lines of code. We show the effectiveness of demand-driven and incremental computation for this analysis. We demonstrate the generality of this approach by deriving an incremental and demand-driven version of a context-sensitive points-to analysis.

Incremental Program Analysis. Incremental algorithms for program analysis and optimization have long been a subject of research (e.g. [8, 5, 28]). These algorithms efficiently compute the changes in results due to small changes in the input, and are of importance in software development environments. Information such as global points-to sets (to determine the set of memory locations that may be accessed through pointer dereferences) have been critical for ensuring the soundness of other program analyses and transformations (e.g. program slicing). However, such information is not routinely gathered due to the lack of practical incremental techniques and frameworks for computing it.

Previous works on incremental program analysis have reported significant time and space savings. However, these works provide solutions that are specific to a program analysis technique or problem. For instance, [44] gives an efficient incremental data-

[1] *Flow-insensitive* analyses ignore the control flow in a program, considering it as a *set* of unordered statements. *Context-insensitive* analyses ignore the calling context of procedures [21].

PPDP'05, July 11–13, 2005, Lisbon, Portugal.

flow analysis technique which performs flow-sensitive and context-sensitive points-to analysis; but the technique cannot be readily generalized to other program analyses including other points-to analyses (such as flow-insensitive or context-insensitive analyses). Demand-driven program analysis [13, 12], which efficiently gathers only that information which is needed by its client analysis/ optimization, is also seen as a way to reduce the cost of analysis. To date, there have been no method that combines the benefits of demand-driven and incremental computation. We address this problem in this paper.

Contributions. In this paper, we describe efficient techniques for evaluating program analyses specified as Horn clause rules incrementally and in a demand-driven fashion. Note that many program analysis problems can be readily encoded into deductive framework [2, 31, 11]. By developing techniques that apply to the general setting of this deductive framework, we can generate incremental and demand-driven analyzers for different analysis problems. We illustrate this approach by deriving a practical demand-driven and incremental analyzer for context-insensitive points-to analysis. Experiments show that the incremental analyzer is 20 times faster (on the average) than the comparable from-scratch analyzer for small changes in the analyzed program, with negligible space overhead. We also describe preliminary results on incremental-context sensitive analysis derived using the same framework.

An Overview of Our Solution. We formulate the program analysis problems in terms of query evaluation over logic programs. The analysis is performed by evaluating queries over a logic program which contains clauses defining the analysis itself, as well as a representation of the program to be analyzed (Intensional and Extensional Databases, resp., in database terminology). When we use a goal-directed query evaluation mechanism (such as tabled resolution [38, 10] or magic set transformations [29]), we naturally obtain demand-driven analysis.

Using this formulation, the problem of incremental analysis is reduced to that of evaluating the changes to the derived relation (i.e. materialization of the intensional database) corresponding to the changes to the program (the extensional database). In this paper, we consider changes to the program in terms of addition and deletion of tuples to its relational representation. For example, if an assignment `x=*y` is changed to `x=*z`, this change is represented by the deletion of the tuple corresponding to `x=*y`, and insertion of a new tuple representing `x=*z`.

We have developed efficient techniques for incremental, goal-directed query evaluation over definite logic programs [34]. The incremental evaluation technique keeps an auxiliary data structure, called the *support graph*, to quickly identify the changes to the derived relation when tuples in the base relation are deleted. However, the support graph may consume a large amount of space, and is especially impractical for program analysis problems. In this paper, we describe a hybrid approach which maintains a limited amount of auxiliary support information in order to bound the space usage, and yet exhibits very good time behavior.

The rest of the paper is organized as follows. We describe a logic-programming-based formulation of pointer analysis in Section 2, and its demand-driven version in Section 3. We describe techniques for incremental evaluation in Section 4. Section 5 contains experimental results that show the characteristics of our demand-driven and incremental analyses. A detailed discussion of related research appears in Section 6, and concluding remarks appear in Section 7.

$$\frac{}{u \longrightarrow v}\ \texttt{u = \&v} \qquad \frac{v \longrightarrow x}{u \longrightarrow x}\ \texttt{u = v}$$

$$\frac{v \longrightarrow x,\ x \longrightarrow y}{u \longrightarrow y}\ \texttt{u = *v} \qquad \frac{u \longrightarrow x,\ v \longrightarrow y}{x \longrightarrow y}\ \texttt{*u = v}$$

Figure 1: Anderson's rules for pointer analysis.

```
points_to(U,V) :-
      assign(plain(U),addr(V)).
points_to(U,X) :-
      assign(plain(U),plain(V)),
      points_to(V,X).
points_to(U,Y) :-
      assign(plain(U),star(V)),
      points_to(V,X), points_to(X,Y).
points_to(X,Y) :-
      assign(star(U),plain(V)),
      points_to(U,X), points_to(V,Y).
```

Figure 2: Logic program using Prolog notation corresponding to Anderson's rules

2. A DEDUCTIVE FORMULATION OF POINTER ANALYSIS

We consider Anderson's [3] inclusion-based context-insensitive and flow-insensitive pointer analysis. To simplify our presentation, we assume that the program is decomposed to a set of primitive assignment statements of the following form:

$$u = \&v \mid u = v \mid u = *v \mid *u = v$$

Figure 1 shows the points-to analysis rules for each assignment statements (from [19]). In the figure $x \longrightarrow y$ denotes that x may point to y.

These rules can be readily written as a logic program which is shown in Figure 2. Following Prolog's notational convention, identifiers beginning with uppercase letters denote variables, and identifiers beginning with lowercase letters denote relation names and data constructors. For instance `points_to` denotes the binary may-points-to relation, and the term `points_to`(x,y) denotes that x may point to y. The terms `plain`(x), `addr`(u) and `star`(u) represent pointer variable x and pointer expressions $\&u$ and $*u$ respectively. Also `assign`(U,V) represents the assignment statement with left hand side and right hand side contains pointer expressions corresponding to the terms U and V. For example, the second rule should be read as "U may point to X if there exists an assignment statement of the form $U = V$ in the code and V may point to X."

Based on this formulation the set of all variables that a given variable v may point to can be computed as answers to the query[2] `points_to`(v, X). For instance, given set of assignment statements {`u=&v`, `p=u`, `u=p`}, the query `points_to(p, X)` has one answer `X=v`, meaning that `p` may point to `v`. While the points-to analysis can be succinctly encoded in Prolog syntax, most Prolog systems will fail to evaluate the program correctly. This results from Prolog's inability to compute the least models of programs with left recursion— even for Datalog programs (i.e. logic programs without data structures). For instance, consider the evaluation of the query `points_to(p,X)` w.r.t. the set of assignments

[2]In this paper, we use the terms "query", "goal", "subgoal" and "call" interchangeably.

{u=&v, p=u, u=p, t=&u}. In resolving the original goal, Prolog will issue the query points_to(u,X) (due to p=u), whose resolution gives an answer X=v. To find more answers for the latter goal, we again encounter the goal points_to(p,X). This causes Prolog to loop.

Tabled resolution is a goal-directed evaluation technique [38, 10] which removes this shortcoming by using memoization as follows. Tabled resolution maintains the set of subgoals encountered so far (called the *call table*) and their associated answers (called *answer tables*, one table per call). During resolution, if a subgoal occurs in the call table, its answers can be found by simply retrieving answers from the corresponding answer table. This process is called *answer clause resolution*. When a subgoal is encountered for the first time, it is added to the call table. Answers to this subgoal are computed by resolving the goal w.r.t. program clauses (called *program clause resolution*), and are added to its answer table. Consider again the query points_to(p,X) w.r.t. the set of assignments {u=&v, p=u, u=p, t=&u}. Upon first encountering the original query, tabled resolution adds the goal to the call table and creates an empty answer table for it. Program clause resolution will then produce the goal points_to(u,X), which, in turn is entered in the call table. Further resolution will produce one answer X=v, which is entered in both answer tables. Continuing with resolution, we will once again get the goal points_to(p,X). Instead of doing program clause resolution which makes Prolog loop, tabled resolution resolves this goal using answers in the first goal's table. In this example, this produces X=v, an answer that was already generated. No further answers can be generated, and hence evaluation terminates.

By remembering the past resolution steps and avoid repeating them, memoization helps tabled resolution terminate for datalog programs, and moreover, evaluate queries with polynomial data complexity. Moreover, unlike the semi-naive algorithm used in the deductive database literature [39], tabled resolution is goal-directed, and hence is naturally suited for demand-driven analysis.

Pragmatics. We apply the above analysis to C programs by transforming all assignment expressions into a set of primitive assignments. Nested uses of & and * are handled by introducing temporary variables. For each static call site we introduce assignment statements where formal parameter is assigned to actual parameter. Dynamic call sites are resolved matching with functions having same number and types of parameters. If the returned value is assigned to a variable then we generate an assignment statements which assigns a temporary variable (same as the function name) to that variable. For each function the return expression is assigned to the same temporary variable generated from function name. In this paper we consider field-independent analysis which ignores field information for accessing structures and unions. Each array is treated as a single variable and index information is ignored. Relaxing these restrictions adds complexity to the analysis but does not reveal any more insight into the problem of incremental analysis, and hence we describe only an analysis with these restrictions.

3. DEMAND-DRIVEN POINTER ANALYSIS

Recall that the evaluation of points_to(p,X) w.r.t. the set of assignments {u=&v, p=u, u=p, t=&u} did not make use of the assignment statement t=&u. This is due to the goal-directedness of the evaluation technique: only calls and answers that are needed to resolve the given goal are used. However, note that implementations of tabled resolution follow Prolog's literal selection strategy: at each program clause resolution step, the left-most subgoal is selected for resolution. Hence, the order of literals in a clause affects the propagation of demand. For example, consider the same query points_to(p, X) with the set of assignments {u=&v, p=u, u=p, t=&u, *s=r}. Due to the statement *s=r and the fourth rule for points-to, we will generate new queries points_to(s,p) and points_to(r,X). Thus we evaluate the points-to relations of variables that are unrelated to p.

```
points_to(X,Y):- assign(plain(X),addr(Y)).
points_to(X,Y):- assign(plain(X),plain(Z)),
                 points_to(Z,Y).
points_to(X,Y):- assign(plain(X),star(Z1)),
                 points_to(Z1,Z),points_to(Z,Y).
points_to(X,Y):- pointed_to_by(X,U),
                 assign(star(U),plain(Z)),
                 points_to(Z,Y).
pointed_to_by(X,Y):- assign(plain(Y),addr(X)).
pointed_to_by(X,Y):- pointed_to_by(X,Z),
                     assign(plain(Y),plain(Z)).
pointed_to_by(X,Y):-  pointed_to_by(X,Z),
                      pointed_to_by(Z,Z1),
                      assign(plain(Y),star(Z1)).
pointed_to_by(X,Y):-  pointed_to_by(X,V),
                      assign(star(U),plain(V)),
                      points_to(U,Y).
```

Figure 3: Logic program for points-to analysis specialized with respect to calling modes

Consider the case when we are interested in evaluating, for different variables v, what variables v may point to. This can be done by issuing queries of the form points_to(v, A) for different v: queries where the first argument is bound and second argument is free. The pattern of boundedness of query arguments is known as the *calling mode* of the query; the calling mode of the above query is bound-free. Note that for bound-free queries, the fourth rule defining points-to uses the bound argument (X) only in the second literal. Thus, we will backtrack through every assignment of the form *U=V, without regard to whether it is related to the original query.

We can avoid this by reordering the literals in the fourth rule to as follows:

```
points_to(X,Y) :- points_to(U,X),
    assign(star(U),plain(V)), points_to(V,Y).
```

Due to the first literal in the body of the above rule, we now get queries to points-to with calling mode free-bound. Note that different calling modes require different literal orders. For instance, for free-bound queries, it is better to reorder the body of the fourth rule with points_to(V,Y), as the first literal. We hence specialize the points-to relation with respect to the two calling modes, as shown in Figure 3. In the figure, points_to handles bound-free queries to the original points-to relation; pointed_to_by, the inverse of points_to, handles free-bound queries to the original points-to relation. Note that if we issue only bound-free queries to points_to from the top-level, then all queries to points_to as well as pointed_to_by will be bound-free.

Note that the queries to points_to and pointed_to_by are distinct (bound-free queries are distinct from free-bound queries) and their answers are not shared in the current tabling infrastructure. Subsumptive tabling [30] can be used to share the answers, but only provided a more general (in this case, a free-free) query is issued first; however, the general query will mean that the analysis will no longer be demand-driven.

The rules in Figure 3 are similar to the inference rules for demand driven analysis in [19]. The mode based specialization presented here is straightforward. However since it involves goal re-

[f0] h = &b;	[f3] c = &e;	[f6] j = c;
[f1] j = h;	[f4] c = d;	[f7] g = d;
[f2] c = &b;	[f5] d = j;	[f8] g = c;

(a) Assignments

Answers	Supports	
[a0] points_to(h,b)	s0 {f0}	
[a1] points_to(j,b)	s1 {f1, a0}	s7 {f6,a3}
[a2] points_to(d,b)	s2 {f5,a1}	
[a3] points_to(c,b)	s3 {f2}	s8 {f4,a2}
[a4] points_to(c,e)	s4 {f3}	s9 {f4,a6}
[a5] points_to(j,e)	s5 {f6,a4}	
[a6] points_to(d,e)	s6 {f5,a5}	
[a7] points_to(g,b)	s10 {f7,a2}	s11 {f8,a3}
[a8] points_to(g,e)	s12 {f7,a6}	s13 {f8,a4}

(b) Answers and their corresponding supports

Figure 4: Examples of supports

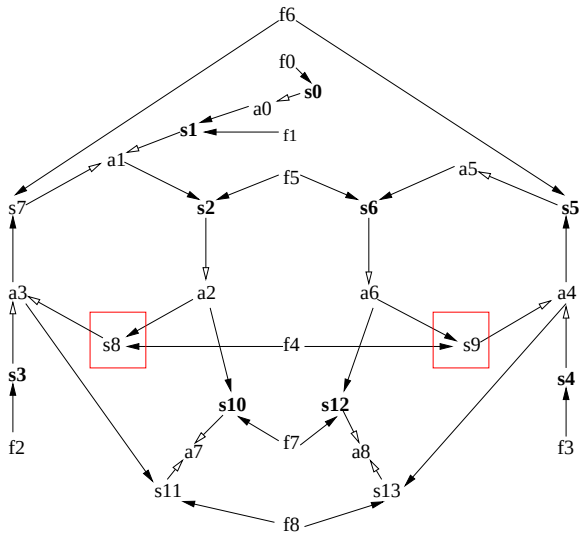

Figure 5: Support graph for program in Figure 4

ordering, it is not clear how this specialization can be automated for general logic programs. The program can be further optimized by grouping the literals on the right hand sides and tabling intermediate results. The details are omitted here, but appear on this paper's web page [33].

4. INCREMENTAL POINTER ANALYSIS

In this section we describe space-efficient techniques for incremental evaluation of logic programs; we describe this technique using the points-to analysis in Figure 3 as an example. Our algorithms compute changes to the points-to set due to the addition or deletion of one or more facts representing the program.

Our incremental evaluation algorithm to handle *addition* of facts is based on program transformation [34]. This technique is relatively straightforward, and has been studied extensively in the literature (e.g. [17]). For every relation p defined in the program, we derive a new relation, called its *delta* relation (denoted by δ_p), which captures changes to the relation due to the addition of facts. For every rule of the form $p :- q_1, q_2, \ldots q_n$ in the original program, we add rules of the form $\gamma_p :- (q_1; \delta_{q_1}), \ldots, (q_{i-1}; \delta_{q_{i-1}}), \delta_{q_i}, q_{i+1}, \ldots, q_n$ for each $i \in [1, n]$, where δ_p is defined as $\gamma_p - p$. For example, corresponding to the second rule given in Figure 3 we generate two rules as shown below.

$$\gamma_{points\_to}(X,Y) :- \delta_{assign}(plain(X), plain(Z))\,, points\_to(Z,Y)\,.$$
$$\gamma_{points\_to}(X,Y) :- (\delta_{assign}(plain(X), plain(Z)); assign(plain(X), plain(Z)))\,, \delta_{points\_to}(Z,Y)\,.$$

In the above, δ_p is the exact set of changes to relation p due to changes in the program, and γ_p is an over-approximation of δ_p. In [34] we proposed data structures and algorithms to efficiently evaluate these delta relations. Program analysis problems do not raise new issues in incremental addition, and the results of [34] are directly applicable.

Efficient incremental computation in the presence of deleted facts is a more complex problem. Techniques independently developed for incremental program analysis [27, 44], incremental view maintenance in deductive databases [17] and incremental model checking [35] are remarkably similar, and have the following two phases. In the first phase (called the *deletion* phase in [17]) the set of answers that could be affected by the deletion of the given set of facts is computed. For instance, consider the C program in Figure 4(a). When the statement `c=&b` is deleted, since `j=c` is a statement in the program, the answer that `j` may point to `b` may be affected, and is marked. The second phase (called the *rederivation* phase in [17]) attempts to rederive the marked answers without using the deleted facts. Affected answers which cannot be rederived are deleted. Note that in the above example, `j` still points to `b` since `h` points to `b`, and `j=h` is a statement in the program. We observe that such deletion followed by rederivation is often wasteful, and most answers marked in the first phase are often rederived (see Section 5 for experimental results).

Our incremental evaluation algorithm considerably improves this situation by keeping an auxiliary data structure, called the *support graph* [34], to quickly identify the set of answers to be marked in the first phase. At a high level, a *support* for an answer is an immediate reason for its truth. More precisely, s is a support for an answer a, if $a :- s$ is an instance of a clause in the program, and all literals in s are true. Note that a support is, in general, a conjunction of answers and facts. For instance, in the above example, the answer `points_to(j,b)` has two supports: (1) `assign(plain(j), plain(h))` and `points_to(h,b)` ({`f1,a0`} in Figure); and (2) `assign(plain(j),plain(c))` and `points_to(c,b)` ({`f6,a3`}).

A support graph is a bipartite graph with vertices drawn from the set of all answers and facts in one partition, and the set of all supports in the other. Figure 4 illustrates supports and support graphs: part (a) shows an example set of primitive assignments in C code form; part (b) shows the answers to queries of the form `points_to(`v`,X)` for all v, and their corresponding supports. We refer to each fact, answer and support with names of the form $f0, f1, \ldots$; $a0, a1, \ldots$; and $s0, s1, \ldots$ respectively. Figure 5 shows the support graph corresponding to the program given in Figure 4. An answer vertex a and its support s are connected by an edge from s to a (called a "answer-of" edge, shown using a white arrowhead in the figure). If a fact or an answer a is in a support s then there is an edge (called a "uses-of" edge, shown using a black arrowhead in the figure) from a to s.

Support graphs make it easy to mark affected answers in the first phase. For example, if we delete the assignment `j=c` (fact $f6$) then we mark all the supports containing that fact, i.e. $s5$ and $s7$ as affected. Since $s5$ supports $a5$, we then mark $a5$ as affected. We continue to propagate the marks, marking support $s6$ and then answer $a6$. Thus we infer that answers `points_to(j,e)` and `points_to(d,e)` are possibly affected by the deletion of statement `j=c`.

Now, note that answer $a6$ is in support $s9$ and $s12$, and hence $s9$ and $s12$ will also be marked. Since $s9$ is a support of $a4$ (`points_to(c,e)`), that answer will also be marked as affected. However, this answer will be rederived (due to the assignment `c=&e`) in the second phase. Note that $a4$ has two supports: $s9$ and $s4$, and that truth of support $s4$ is not dependent on the truth of $a4$. Thus support $s4$ has a derivation that is independent of answer $a4$. Since $s4$ is not marked, we can infer that $a4$ will be unaffected and hence need not be marked. Hence, *if we can quickly identify independence of supports and answers, we can limit the marking of affected answers*, and consequently reduce the rederivation effort also.

In any least fixed point computation, the *first* support used to derive an answer will itself be independent of the answer. We call such a support as the *primary* support for an answer. In Figure 4(b) we list the primary support before any other supports for an answer. All the primary supports are shown in bold in the support graph in Figure 5. In our incremental algorithm for deletion [34], we mark an answer only if its primary support is marked. In the second phase, if a marked answer a has an unmarked support s, we know that s is independent of a, and hence can make s as the new primary support of a. We can now remove the mark on a, remove the marks on supports that were marked due to a, and continue to remove marks by traversing the support graph. For example, deletion phase would mark $a8$ since $s12$ is marked and rederivation phase would remove the mark on $a8$ based on its other unmarked support $s13$.

Note that the above algorithm checks if an answer is rederivable solely based on the support graph, and hence requires us to store the complete support graph. Although this permits us to reduce incremental evaluation time, the support graph size grows quickly enough to make it impractical, especially for program analysis problems. For instance, in our encoding of points-to analysis, we observe many hundreds of supports for each answer (the sizes of complete support graphs for all benchmarks is given in the last column of Table 2 in Section 5).

Below, we describe three new algorithms that keep the space usage bounded by keeping only a part of the support graph, yet exhibit acceptable time performance. The first is a memory efficient algorithm which keeps only the primary support for each answer; the subsequent algorithms expand on the earlier ones, trading off a bounded amount of space to obtain better time performance.

We use the following notations in the description of the algorithms. The answer a supported by s is denoted by `answer_of`(s). The set of all supports $\{s_1, s_2, \ldots, s_k\}$ that contain a given answer or fact a is denoted by `uses_of`(a). We begin with the description of `PS`, the primary-support-based algorithm.

Algorithm PS. In this algorithm we record only the primary support for each answer. The resultant support graph is called primary support graph or PSG. The PSG at the beginning of incremental phase is shown in Figure 6(a). With each support vertex in the support graph, we keep an integer field called `false_count` that records the number of marked answers in that support; this field is initialized to zero at the beginning of each incremental phase. With each answer vertex, we keep three fields: `call`, a pointer to the call

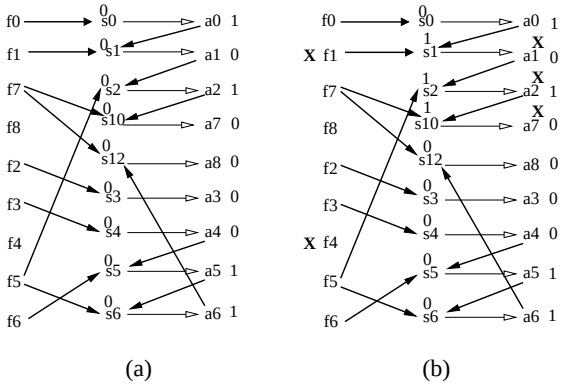

Figure 6: Primary support graphs

```
deletion_phase()
  ∀deleted fact f
    ∀support s ∈ uses_of(f)
      mark_support(s);

mark_support(support s)
  s.false_count++;
  if(s.false_count==1)
    ans=answer_of(s);
    mark_answer(ans);

mark_answer(answer ans)
  mark_delete(ans);
  if(ans.all_support==false)
    ans.call.dirty_count++;
    ∀s ∈ uses_of(ans)
      mark_support(s);
```

Figure 7: Primary support based deletion

to which it is an answer, and `all_support`, a boolean to record whether the support we have for the answer is its only support. Since we have only incomplete support information, rederivation of an answer may be performed using program clause resolution (PCR) of the corresponding call (see below for details). For each call we keep a count of the number of marked answers in a field called `dirty_count`. This field is also initialized to zero before each incremental phase.

When an answer is derived for the first time we generate its primary support, and set its `all_support` to true. If the answer is derived again we discard the new support and make `all_support` false. The incremental deletion algorithm consists of two phases, described below:

Phase 1: Deletion phase. The algorithm for this phase is given in Figure 7. Consider the deletion of assignment statements `c=d` and `j=h` in the example given in Figure 6(a). The support graph after the deletion phase is shown in Figure 6(b). In the figures, we have annotated each support vertex with the value of its `false_count` field, and each answer vertex with the value of its `all_support` field. The deletion phase marks the answers $a1$, $a2$ and $a7$ and it makes `dirty_count=1` for call `points_to(j,X)` and `points_to(g,X)` but not for call `points_to(d,X)` since the `all_support` field of $a2$ is true (see the rederivation phase, below).

Phase 2: PCR Rederivation phase. To check if an answer marked for deletion can be rederived, we can per-

form program clause resolution (PCR) for the corresponding call. PCR, however, is expensive since it may rederive answers not even marked for deletion in the first place. For instance, computing `points_to(j,X)`, `points_to(d,X)` and `points_to(g,X)` by PCR to rederive $a1$, $a2$ and $a7$ will also derive answers $a5$, $a6$ and $a8$. Hence we devise two ways to avoid PCR-based rederivation whenever possible.

The first is based on `all_support` field of an answer. If this field is true, then the recorded supports (in algorithm PS only the primary support) for an answer are the only supports for this answer. Hence the only way the answer can be true is if the deletion mark on the primary support is removed. For instance, we do not have to execute call for `points_to(d,X)` to know whether `points_to(d,b)` (answer $a2$) is true, as we know that if `points_to(j,b)` (answer $a1$) is true then that will subsequently make support $s2$ and answer $a2$ true by removal of marks through support graph. Thus for a marked answer with `all_support` bit set we do not increment the `dirty_count` for its call.

The second heuristic uses the notion of *derivation length* (denoted by *dl*) to determine the order in which marked answers are processed. Intuitively, if an answer a_1 has a lower *dl* than answer a_2, then a_1 is independent of a_2. Thus if answers with lower *dl*s are rederived earlier, then PCRs for rederiving higher *dl*s may be avoided altogether. Given a vertex v in the support graph, *dl*(v) is defined as

$$\begin{cases} 0 & \text{if } v \text{ is a fact} \\ \max\{dl(a) \mid v \in \texttt{uses\_of}(a)\} + 1 & \text{if } v \text{ is a support} \\ dl(s) \mid s \text{ is the primary support of } v & \text{if } v \text{ is an answer} \end{cases}$$

To ensure that a call is issued only when necessary, we keep the total number of its marked answers in `dirty_count`, and issue a call only if its `dirty_count` is non-zero. Whenever an answer is rederived using the support graph and its `all_support` is false, we decrement the `dirty_count` of its call. In the above example, since answer $a1$ has lower derivation length (2) compared to that of answer $a7$ (4), we issue the call `points_to(j,X)` first. This rederives the answer $a1$ along with generation of new support $s7$. By calling `rederive_answer(`$a1$`)` and `rederive_support(`$s2$`)` we rederive the answer $a2$ and, subsequently, calling `rederive_answer(`$a2$`)` and `rederive_support(`$s10$`)` we rederive $a7$. This decrements the `dirty_count` for the call `points_to(g,X)` to 0 and thereby we save the PCR rederivation for the call `points_to(g,X)`.

The algorithm for Phase 2 is in Figure 8.

Algorithm AS. In the above example we mark the answer $a1$ (`points_to(j,b)`) since its primary support is marked. Note that, its other support $s7$ is also independent of $a1$: i.e. if $s7$ is derivable, so is $a1$. We call these supports as *acyclic supports*.

An acyclic support subgraph (ASG) corresponding to a support graph (SG) is defined as a *directed acyclic subgraph* of SG which contains all the facts and answers of SG and at least one support for each answer. Note that, there can be multiple acyclic support subgraph corresponding to a support graph. Supports in an ASG are called acyclic supports.

We can now generalize Algorithm PS by keeping acyclic supports with each answer, and marking an answer only if all its acyclic supports are marked. In the above example, if we keep $s7$ as an acyclic support of $a1$ then $a1$ will not be marked, and consequently, there will be no rederivation. This generalization leads to Algorithm AS. The size of the support graph is limited by the specifying the maximum number of acyclic supports stored for an answer (called the Maximum Acyclic Support Count, or MASC).

```
rederivation_phase()
  queue = set of all marked answers
  while(queue is not empty)
    remove ans from queue with minimum dl;
    call=ans.call;
    if((!considered_for_rederivation(call))
          && dirty_count(call)>0)
      consider_for_rederivation(call)=true;
      execute_query(call);
      ∀newly generated answers ans
        if(marked_deleted(ans))
          rederive_answer(ans);

rederive_answer(answer ans)
  remove_delete_mark(ans);
  if(ans.all_support==false)
    ans.call.dirty_count−−;
  ∀s ∈ uses_of(ans)
    rederive_support(s)

rederive_support(support s)
  s.false_count−−;
  if(s.false_count==0)
    ans=answer_of(s);
    rederive_answer(ans);
```

Figure 8: PCR rederivation for algorithm PS

The key problem in Algorithm AS is to determine whether a support is acyclic. If the derivation length of a support s is no greater than that of an answer a, then s has a derivation that is independent of a, and therefore s is acyclic. We use this as a conservative test to determine the acyclicity of a support:

$$s \text{ is acyclic if } dl(s) \leq dl(\texttt{answer\_of}(s))$$

For the program in Figure 4, the above heuristic deems supports $s8$ and $s9$ as potentially cyclic supports since $dl(a3) = 1$, $dl(s8) = 4$ and $dl(a4) = 1$, $dl(s9) = 4$. These supports are marked by boxes in the support graph in Figure 5.

Algorithm AS is derived from Algorithm PS as follows. With each answer we now keep the number of unmarked acyclic supports recorded for that answer in `support_count`. In `mark_support`, when `false_count` of a support becomes 1, we decrement the `support_count` of its answer. We mark an answer (and propagate this mark) only when its `support_count` falls to zero. It is easy to see that our Algorithm PS is a special case of Algorithm AS with MASC=1. Consider the example in Figure 4. With MASC=2, we will store $s7$, $s11$ and $s13$ as additional acyclic supports (apart from the primary supports); When statements `j=h` and `c=d` are deleted, Algorithm AS will not mark any answer.

Note, however, that an acyclic support for an answer may not be remain acyclic after rederivation. For instance, in Figure 4, consider the deletion of statement `c=&b` ($f2$). Using Algorithm AS with MASC=2, we will mark $s3$, $a3$, $s7$, and $s11$. The new primary support of $a3$ is $s8$ whose derivation length is 4. Thus $dl(a3) = 4$ and hence $dl(s7) = dl(s11) = 5$. Consequently, $s7$ and $s11$ are no longer acyclic and will have to be removed from the graph. The PCR rederivation phase of Algorithm AS is obtained from that of Algorithm PS as follows. Whenever a new answer a is generated by PCR, we store the derivation length of its first support as a.`dl`. When we rederive an answer a based on an existing support s (in `rederive_support`) and the answer is not marked (i.e. it had been already rederived), we check the cyclicity of the s with respect to a, and delete s if it is not acyclic. The modified procedure

```
rederive_support(support s)
  s.false_count−−;
  if(s.false_count==0)
    ans=answer_of(s);
    dl(s)= maximum{dl(a)| s ∈ uses_of(a)}+1;
    if(ans.support_count==0)
      ans.support_count++;
      dl(ans) = dl(s);
      rederive_answer(ans);
    else
      if(dl(s)≤dl(ans))
        ans.support_count++;
```

Figure 9: PCR rederivation for algorithm AS

```
gred()
  rederive_list={}
  ∀marked answers ans
    if (ans.total_support_count>0)
      ans.support_count=ans.total_support_count;
      dl(ans) = max {dl(s)| s is a support of ans,
                          s.false_count=0}
      rederive_list=rederive_list + ans
  ∀ans ∈ rederive_list
      rederive_answer(ans);
```

Figure 10: Support graph based rederivation

`rederive_support` of Algorithm PS is shown in Figure 9.

Algorithm MS. Algorithm AS improves on PS by reducing the number of answers marked in the first phase. We now describe Algorithm MS, which builds on AS, and aims to reduce the number of PCR rederivations. In AS we bound the number of acyclic supports for each answer by the constant MASC. For some answers we might not fill this "quota" as the number of acyclic supports may be less than MASC. Algorithm MS fills the rest of the "quota" for each answer with other (possibly cyclic) supports, while giving preference to acyclic supports.

For instance consider the example in Figure 4. Let MASC=2, and assume that we keep supports $s8$ and $s9$ in the support graph (as supports to $a3$ and $a4$ respectively) even though they are do not meet the acyclicity criterion described before. Now consider the deletion of statement `c=&b`. In the deletion phase we will mark $s3, a3, s7$ and $s11$. Since $s8$ remains at the end of this phase, we know that the support $s8$ has a derivation that is independent of $a3$. Thus $s8$ now qualifies as an acyclic support of $a3$. Hence we can remove the mark on $a3$ without PCR rederivation. We derive Algorithm MS from Algorithm AS by invoking `gred` (Figure 10) which does a support graph based rederivation before doing PCR rederivation. In addition to the data for Algorithm AS, this algorithm maintains the total number of current supports for an answer, and also a structure to access all supports for a given answer. The details of this additional bookkeeping are straightforward and omitted. This strategy does not affect the number of answers marked, but reduces the number of PCR rederivations.

Discussion. We now describe the space and time complexity of from-scratch as well as incremental pointer analysis. These complexity measures are given in terms of the number of variables of the program to be analyzed (denoted by N). The worst-case running time for Anderson's pointer analysis is $O(N^3)$. By optimizing the analysis program in Figure 3 as described at the end of Section 3 and evaluating the program using tabling, our implementation has the same worst-case time complexity. The space complexity of the analysis as well as its implementation is $O(N^2)$.

Programs	LOC	Fun. Ptr	Prim. Assign	From Scratch All Points-to		
				Avg. Size	Time(s)	Space
smail	3850	-	664	24.5	0.09	1.5M
gzip	8620	1	391	0.7	0.01	0.8M
parser	11391	-	2190	5.8	0.01	2.6M
vpr	17729	10	2708	1.8	0.03	3M
m88ksim	19093	2	1406	6	0.03	2M
twmc	24951	3	7065	16.7	2.48	12.8M
nethack	33993	11	4875	35.0	1.04	9.7M
vortex	67110	14	14387	69.8	13.34	40M

Table 1: Benchmark characteristics

The size of the *complete* support graph for pointer analysis is $O(N^3)$. However, the size of the *partial* support graph described in this section is $O(N^2)$ since the number of supports for an answer is bounded by a constant. Thus, the *space complexity* of the incremental pointer analysis has been made to match the space complexity of from-scratch analysis.

The time taken by our incremental insertion algorithm is, in worst case, the time taken for from-scratch analysis of the changed program. For incremental deletion, the time taken by the deletion phase is proportional to the size of the partial support graph, and hence bounded by the time taken for from-scratch analysis of the original program. Rederivations done on the basis of the partial support graph is also proportional to the size of the graph. Now consider the rederivations that require program clause resolution. Note that PCR rederivation invokes calls that had been made for the original analysis, and hence the PCR rederivation time is bounded by the from-scratch analysis time. Therefore, incremental deletion, which comprises of deletion and rederivation phases, takes time proportional to that of from-scratch analysis of the original program.

The correctness of our incremental insertion and deletion algorithms follows from the correctness of DRed algorithm [17]. Additionally our deletion algorithm identifies a set of acyclic supports for an answer. It is easy to construct a proof for an answer using an acyclic support, by recursively building proofs of answers in the support. Since the support is acyclic, this recursive process will terminate, yielding a proof. Hence it follows that any answer with an unmarked acyclic support has a derivation and hence is not deleted.

5. EXPERIMENTAL RESULTS

Experimental Setup. We measured the performance of our algorithms for demand driven and incremental points-to analysis on programs taken from C benchmarks available with PUF compiler suite and SPEC95 benchmarks. The incremental evaluation algorithms were implemented by extending the XSB logic programming system [43] (v2.6). Our incremental points-to analysis system, the benchmarks, and detailed experimental results are available at [33].

We preprocessed the C source code using CIL [26] into Prolog facts representing the primitive assignment statements. Each library function was replaced by a stub representing the data flow between its formal parameters and return value and preprocessed in the same manner. Performance measurements were taken on a PC with 1.4Ghz Pentium M processor with 512MB of physical memory running Linux (Debian) 2.6.7. We present the benchmark characteristics in Table 1.

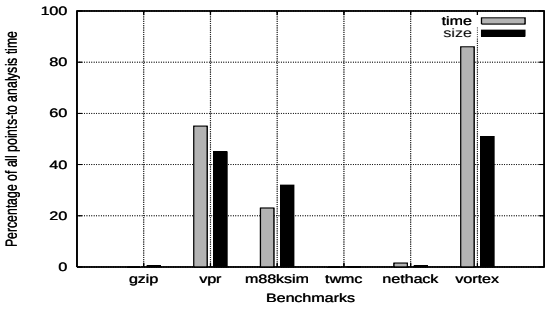

Figure 11: Relative performance of Function Pointer Analysis w.r.t. All Points-to Analysis

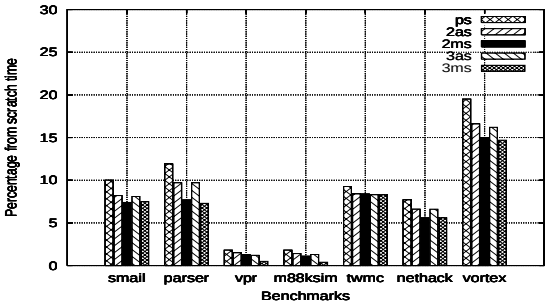

Figure 12: All Points-To Analysis: Incremental deletion time relative to from-scratch time

Demand Driven Analysis. We measured the effectiveness of the demand-driven analysis for resolving dynamic call sites in the benchmark programs. Function Pointer Analysis (FPA) uses the logic program encoding in Figure 3, and computes all answers to queries of the form `points_to(`f`, X)` for each function pointer f occurring in a given benchmark program. All Points-To Analysis (APA) also uses the same logic program, but computes the entire points to relation (i.e. over all program variables). Figure 11 shows the time taken and the size of the points-to relation computed by FPA relative to those of APA. Observe from the figure that, for some benchmarks (`gzip`, `twmc`, and `nethack`) FPA takes less than 1.8% of the time taken by APA; and in others (`vpr`, `vortex`) the time taken for FPA is a significant fraction of that for APA. The latter benchmarks keep function pointers in structures, and the higher analysis times appear to be the artefact of performing field-insensitive analysis, which results in a large number of spurious points-to tuples for the function pointers.

Incremental Analysis. To measure the effectiveness of our incremental evaluation algorithms, we performed All Points-to Analysis (APA) and Function Pointer Analysis (FPA) using the logic program encoding in Figure 3. For single statement-level changes to the benchmark programs, we measured the time and space taken to redo the analyses from scratch and to maintain the points-to relations using our incremental techniques. We first report on the performance of incremental deletion.

Effectiveness of incremental deletion. We compare the average time taken for single assignment statement deletion in source (over 105 deletions) for incremental and from-scratch APA. Note that each assignment statement in the source may correspond to multiple primitive assignment statements in the preprocessed code. Figure 12 shows the average time taken for incremental APA *per source statement deletion* as a percentage of the time taken for from-scratch APA. That figure shows the relative performance of the different incremental deletion algorithms PS, AS and MS, and for the latter, with different values of maximum support set size (MASC=2 and 3). Recall that PS is identical to AS and MS with MASC=1.

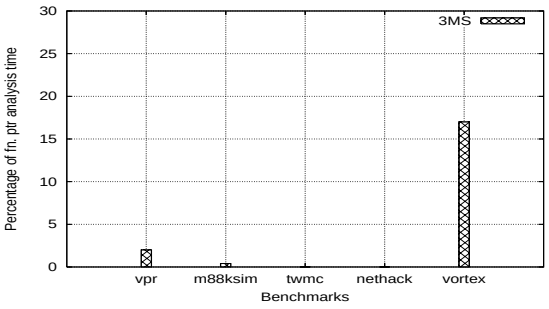

Figure 13: Function Pointer Analysis: Incremental deletion time relative to from-scratch time

	PS	2AS	2MS	3AS	3MS	ALL
small	26K	29K	37K	30K	47K	211K
parser	24K	29K	34K	32K	42K	91K
vpr	9.9K	11.1K	11.6K	11.7K	12.7K	14.4K
m88ksim	11K	12K	14K	13K	15K	21K
twmc	279K	389K	397K	490K	506K	5728K
nethack	205K	239K	270K	252K	317K	2075K
vortex	1202K	1564K	1714K	1810K	2099K	33444K

Table 2: Support graph sizes

Observe from the figure that, even the simplest of our primary support-based algorithms, PS, takes 1.8%(m88ksim) to 19%(vortex) of the from-scratch time. Time decreases with increasing MASC, and the MS algorithm performs better than AS. This is because most of the time (more than 95%) is taken by the PCR rederivation phase and keeping extra supports considerably reduces the number of calls made for PCR rederivation.

Figure 13 shows the average time taken for incremental *FPA* per source statement deletion as a percentage of the time taken for from-scratch FPA. The figure shows that the performance gains due to demand-driven analysis can be further improved by incremental analysis. Note that the gains for FPA are consistent with those for APA (Figure 12).

Space Behavior. In Table 2 we compare the space overhead of our different incremental deletion algorithms by comparing the total number of supports recorded for APA. The last column in the table shows the total number of supports in complete support graph for each benchmark. Each support vertex takes at most 6 words, and the total number of supports is a very good measure of the space overheads due to incremental evaluation[3]. Note that the support graph space overheads are very small when the number of supports per answer is bounded. However, note that the size of full support graphs is prohibitively large for the bigger examples (e.g. 30M vertices for `vortex`), and hence the simpler incremental algorithm of [34] is impractical. It is interesting to observe that the support size for MS with MASC=2 is smaller than AS with MASC=3, but the average evaluation time is longer for the latter (Figure 12), showing the advantage of keeping non-acyclic supports in reducing evaluation time.

[3]The amount of space each support takes varies from Algorithm PS (2 words), AS and MS (6 words).

	MASC=0	1	2	3
smail	15375	890	636	618
parser	16311	72.6	71.4	70
vpr	1400	20	10	9.8
m88ksim	283	73	45.8	45.6
twmc	51417	2770	2172	1933
nethack	54490	337	208	208
vortex	493445	6273	614	614

Table 3: No. of answers marked in Deletion Phase

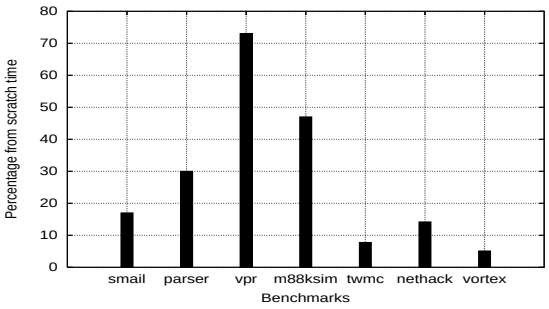

Figure 14: All Points-To Analysis: Incremental insertion time as percentage of from-scratch time

The importance of supports. To measure the effectiveness of primary and acyclic supports in reducing the number of answers marked and later rederived, we collected the average number of answers marked in the first phase for APA. Table 3 shows the number of answers marked using the AS algorithm for various benchmarks (rows) and for different MASC values (columns). Note that AS and MS are identical w.r.t. number of marked answers, and that AS with MASC=0 is identical to the non-support-based (e.g. [17, 44, 35]) algorithms. The comparison of MASC=0 and MASC=1 columns shows the substantial advantage (4–160 times) of keeping primary support for restricting the marking and rederivation of answers. The number of marked answers decreases as we increase the number of acyclic supports stored. However, the decrease tapers off after MASC=2, and MASC=2 appears to be a good balance between evaluation time and space overhead.

Tables 2 and 3 show the importance of keeping a *partial* support graph for deriving scalable incremental analyses. Note that the PCR rederivation phase is needed only when the support set is partial. We observe that this phase takes more than 95% of the incremental evaluation time. Hence, if it is if the entire support graph is kept, the incremental evaluation time will be less than 2% of the from-scratch evaluation time. The algorithms presented in this paper hence trade off incremental evaluation time to lower the space requirements.

Incremental Insertion. We measured the performance of incrementally maintaining the points-to sets for APA when new statements are added, with the same random assignment statements used for incremental deletion. We computed the points-to relation after deleting a selected statement, then added back the deleted statement and ran the transformed program for incremental addition. The average time taken to run the incremental insertion query as a percentage of the from-scratch re-evaluation time for each benchmark is shown in Figure 14. Observe that, for `m88ksim` and `CPR` incremental insertion takes about 45-73% of the from-scratch time. However, but for large benchmarks (`nethack`, `twmc` and `vortex`) it takes only 5% of the from-scratch evaluation time.

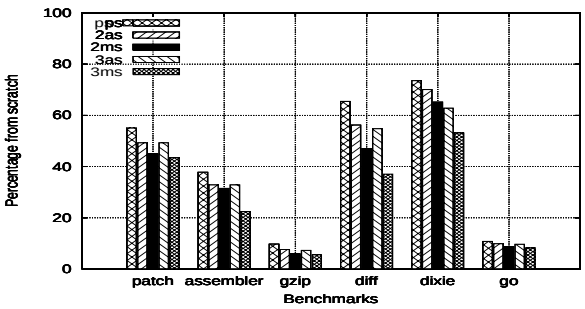

Figure 15: Relative time for incremental deletion in context sensitive analysis

Note that the time reflects the effect of insertion on *all* derived relations.

Context Sensitive Analysis. In addition to the context insensitive analysis described in this paper, we also encoded a context sensitive (summary-based) analysis in our framework. This analysis builds a parametric summary for each procedure and instantiates the summary at the call sites; its details are beyond the scope of this paper. The average time to perform incremental evaluation after a single source statement deletion is given in Figure 15 as a percentage of the from-scratch analysis time. We observe with MS algorithm with MASC=3 takes about 50% of the from-scratch time, though with only primary support we could take as high as 73% of the from-scratch time. This high percentage is due to the fact that deletion of each assignment statement marks a large number of answers (compared to context in-sensitive analysis) which in turn triggers a large number of expensive PCR rederivations of answers.

Adding field-sensitivity to our existing pointer analysis would make the analysis more precise does not change the essential nature of the analysis problem and is unlikely to affect the relative performance of incremental analysis with respect to the from-scratch analysis. However, a flow-sensitive analysis cannot be efficiently evaluated with our current implementation. Flow-sensitive analysis needs ability to handle of stratified negation (due to kill sets). Our current implementation handles insertions in a top-down manner (by issuing queries for the delta relations) and deletions in a bottom-up manner. Although negation can be handled in principle, a more efficient implementation would need to propagate both insertions and deletions along the same direction; this is a topic of current research.

6. RELATED WORK

The problem of incremental evaluation of logic programs has been extensively researched, especially in the context of deductive databases[4]. Few techniques however deal with incremental evaluation in the presence of recursive rules as well as deletion of facts. The most general of these, Delete-Rederive (DRed) algorithm [17] is the closest to our incremental evaluation technique. DRed computes the dependencies between answers at the time of incremental evaluation and does not maintain a support graph. The MCI algorithm [35] was proposed in a different context, namely incremental model checking, and maintains an analogue of the support graph. However both MCI and DRed mark an answer as deleted if *any*

[4]See [34] for a detailed comparison of our basic incremental evaluation technique with those in deductive database literature.

of its supports is deleted— thereby over-propagating the effects of a deletion. In [34] we introduced the notion of primary supports and showed its effectiveness in reducing recomputation. In that algorithm, *all* supports of an answer were recorded, and hence the expensive PCR rederivation operation is never needed. Consequently, the algorithm in [34] shows better time behavior than the one described in this paper. However, the maintenance of complete support graphs leads to space overheads that are prohibitive for program analysis, rendering the algorithm in [34] impractical.

Points-to analysis has been studied extensively (see [21] for a survey), and continues to attract significant attention (e.g. [15, 16, 18, 42, 24]). The aim of this work is to present techniques for making program analysis incremental. Although we do not directly address the accuracy-time tradeoffs that are at the core of much of the points-to analysis work, ability to perform incremental analysis will enable us to deploy more accurate analyses that may otherwise be deemed impractical.

Among the many works on points-to analysis, Heintze and Tardieu [20, 19] encode flow-insensitive and context-insensitive subset-based pointer analysis due to Anderson [3] using deductive rules. Our encoding in Section 2 follows [19]. We derive demand-driven as well as incremental analyses directly based on these rules. Several graph based optimization techniques [14, 37] cannot be declaratively encoded showing the limitations of rule-based techniques. However, the idea of deduction is present at least implicitly in every worklist-based algorithm, and we believe the analogues of support graphs in those settings will make it possible to derive incremental versions of such algorithms as well.

Demand-driven program analysis using logic-programming based formulation has been studied before [31, 19]. For instance, our encoding of demand driven context-insensitive Anderson's rules in terms of Horn clauses (Figure 3) is similar in nature to the rules obtained in [19]. The main difference between the earlier works and the one presented in this paper lies in the way the rules are evaluated. Heintze et. al. use CLA [20] infrastructure to implement a set-based algorithm corresponding to the rules by using a technique similar to magic set transformation to bring goal-directedness to bottom-up evaluation [31, 22]. Tabled resolution [38] is naturally goal directed, and as observed in [31] this strategy ensures that it accesses only those assignment statements and generates only those intermediate queries which are relevant to answer the top level query [11]. Although the implementation by Heintze et. al is considerably faster than our from-scratch APA analysis, our support graph based incremental algorithm can be incorporated into CLA framework for yielding better performance.

In [13] the authors presented a method for the construction of precise demand-driven algorithms for the class of distributive finite data flow problems. As also explained in [19] demand-driven pointer analysis falls outside the scope of [13]. In [4] a demand-driven pointer algorithm has been presented for Steengard's Algorithm [36] and program slicing is used to show effectiveness of demand driven analysis. Demand-driven call graph construction for the Java programs has been investigated in [1].

The closest related work on incremental pointer analysis we are aware of is that of Yur et. al. [44]. The authors developed incremental pointer aliasing algorithm based on Landi-Ryders's flow- and context-sensitive alias analysis [23]. They update points-to information after a program change rather than computing it from scratch. Their incremental algorithm is not *complete* in the sense that it may compute less precise solution than the exhaustive technique. In contrast, we show that our incremental algorithms produce exactly same relations as their exhaustive counterpart. The incremental algorithm in [44] also has the two phases of alias falsification (deletion) and alias introduction (rederivation) as our algorithm. Their selective falsification strategy degenerates to falsification strategy of DRed [17] where all directly and indirectly generated aliases due to the deleted statement are falsified. Our work substantially reduces the unnecessary falsification followed by rederivation. The experimental results presented in Section 5 show the effectiveness and the necessity of our techniques. Perhaps more importantly, our memory-efficient support graph based incremental algorithm is not confined to alias analysis but can be readily applied to any analysis specified using deduction rules.

In [40] an incremental algorithm is presented which analyzes part of the program assuming no previous analysis result. That algorithm monitors the analysis results incrementally in each phase to direct the analysis in those parts of the program which offer highest expected optimized return. That work does not consider the problem of updating existing analysis results to reflect the effect of program changes.

Many incremental algorithm have been developed for data-flow analysis problems. Some incremental analyses use the elimination method [6, 9, 32]; some are based on the technique of restarting iterations [27] and some are combination of the two techniques [25]. A comparison of incremental iterative algorithm can be found in [7]. Effectiveness of incremental analysis has been shown for MOD analysis of C programs [45]. Pollock and Soffa [27] presented precise incremental iterative algorithm using change classification and reinitialization for bitvector problems. They employ two phase solution where the *exaggerate* and *adjust* phases correspond to our delete and rederive phases respectively. Adding support information to the data flow sets will reduce the effort in the adjust phase of this algorithm also.

7. CONCLUSION AND FUTURE WORK

In this paper we presented incremental and demand driven algorithms for program analyses formulated using deductive rules. We demonstrate the effectiveness of our demand driven approach by first encoding context insensitive Anderson's pointer analysis using deductive rules and later deriving a set of rules more suitable for demand driven query. For some benchmarks we observe that demand driven analysis takes less than 1% time for resolving function pointers compared to its exhaustive query. Averaged over all benchmarks, our primary-support based incremental algorithm for single source statement deletion takes about 9% of time of computing from scratch with negligible space overhead. The time can be further reduced to 6% by keeping more supports. We also presented preliminary experimental results of incremental evaluation of a context-sensitive subset-based points-to analysis. These results shows that our incremental evaluation framework can readily accommodate more complex analysis.

Many data-flow analysis problems can be cast in terms of query evaluation over logic programs. The presence of *kill* sets however gives rise to programs with negation. Our techniques can, in principle, be extended to handle logic programs with stratified negation (i.e. no cycles involving negation). We are currently extending our implementation to accommodate such programs.

Although our support-based deletion algorithm is implemented on a top-down goal-directed memoized logic programming framework, the idea of supports can be used to derive incremental counterparts of other algorithms which compute least fixed points. However, when only partial support sets are maintained, the rederivation step would need goal-directed evaluation. Designing an effective rederivation algorithm that can be used in a bottom-up evaluation framework (e.g. CLA [20]) is an interesting open problem.

Note that, for any demand-driven analysis, incremental inser-

tion and PCR rederivation part of incremental deletion can be done lazily: i.e. only when a related query is issued by the client analysis. However, deletion of tuples from relations is an inherently bottom-up process and is best done eagerly. We are currently investigating techniques to perform lazy insertion and rederivation.

8. ACKNOWLEDGEMENTS

This research was supported in part by NSF grants CCR-0205376, CCR-0311512, and ONR grant N000140110967. We thank Wei Xu for his help in building an interface to C front end.

9. REFERENCES

[1] G. Agrawal, J. Li, and Q. Su. Evaluating a demand driven technique for call graph construction. In *Computational Complexity*, volume 2916 of *LNCS*, pages 29–45. Springer-Verlag, 2002.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*, pages 585–718. Addison-Wesley, 1986.

[3] L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Unversity of Copenhagen, 1994.

[4] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Foundations of Software Engineering*, pages 46–55, 1998.

[5] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Principles of Programming Languages*, pages 384–396. ACM Press, 1993.

[6] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst.*, 12(3):341–395, 1990.

[7] M. G. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Trans. Softw. Eng.*, 16(7):723–728, 1990.

[8] M. Carroll and B. G. Ryder. An incremental algorithm for software analysis. In *ACM software engg. symp. on practical software development environments*, pages 171–179, 1987.

[9] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute update. In *Principles of Programming Languages*, pages 274–284. ACM Press, 1988.

[10] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1):20–74, 1996.

[11] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems — a case study. In *Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.

[12] E. Duesterwald, R. Gupta, and M. L. Soffa. A demand-driven analyzer for data flow testing at the integration level. In *International Conference on Software Engineering*, pages 575–584, 1996.

[13] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, 1997.

[14] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial Online Cycle Elimination in Inclusion Constraint Graphs. In *Programming Language Design and Implementation*, pages 85–96. ACM Press, 1998.

[15] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Programming Language Design and Implementation*, pages 253–263. ACM Press, 2000.

[16] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *Static Analysis Symposium*, volume 1824 of *LNCS*, pages 175–198, 2000.

[17] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[18] S. Guyer and C. Lin. Client-driven pointer analysis. In *Static Analysis Symposium*, pages 214–236, 2003.

[19] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *Programming Language Design and Implementation*, pages 24–34. ACM Press, 2001.

[20] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *Programming Language Design and Implementation*, pages 254–263. ACM Press, 2001.

[21] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[22] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering*, pages 104–115, 1995.

[23] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Programming Language Design and Implementation*, volume 27, pages 235–248. ACM Press, 1992.

[24] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Static Analysis Symposium*, pages 279–298. Springer-Verlag, 2001.

[25] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Principles of Programming Languages*, pages 184–196. ACM Press, 1990.

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[27] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Softw. Eng.*, 15(12):1537–1549, 1989.

[28] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Trans. Program. Lang. Syst.*, 14(2):173–200, 1992.

[29] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programming. In *JICSLP*, 1988.

[30] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *Joint International Conference/Symposium on Logic Programming*. MIT Press, 1996.

[31] T. W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases, ILPS*, pages 163–196, 1993.

[32] B. G. Ryder and M. C. Paull. Incremental data-flow analysis algorithms. *ACM Trans. Program. Lang. Syst.*, 10(1):1–50, 1988.

[33] D. Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. Downloads are available at `www.lmc.cs.sunysb.edu/~dsaha/incr_pta`.

[34] D. Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In *International Conference on Logic Programming*, volume 2916 of *LNCS*, pages 389–406, 2003.

[35] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, volume 818 of *LNCS*, pages 351–363, 1994.

[36] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, pages 32–41. ACM Press, 1996.

[37] Z. Su, M. Fahndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Principles of Programming Languages*, pages 81–95. ACM Press, 2000.

[38] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98, 1986.

[39] J. D. Ullman. *Principles of Database and Knowledge-base Systems, Volume II*. Computer Science Press, 1989.

[40] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *Programming Language Design and Implementation*, pages 35–46. ACM Press, 2001.

[41] D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.

[42] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation*, pages 131–144. ACM Press, 2004.

[43] XSB. The XSB logic programming system. Available from `http://xsb.sourceforge.net`.

[44] J. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *International Conference on Software Engineering*, pages 442–451, 1999.

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