Demand-Driven Pointer Analysis

Nevin Heintze
Research, Agere Systems
(formerly Lucent Technologies' Microelectronics Division)
nch@agere.com

Olivier Tardieu
Ecole Des Mines, Paris
olivier.tardieu@mines.org

ABSTRACT

Known algorithms for pointer analysis are "global" in the sense that they perform an exhaustive analysis of a program or program component. In this paper we introduce a demand-driven approach for pointer analysis. Specifically, we describe a *demand-driven* flow-insensitive, subset-based, context-insensitive points-to analysis. Given a list of pointer variables (a query), our analysis performs just enough computation to determine the points-to sets for these query variables. Using deductive reachability formulations of both the exhaustive and the demand-driven analyses, we prove that our algorithm is correct. We also show that our analysis is optimal in the sense that it does not do more work than necessary. We illustrate the feasibility and efficiency of our analysis with an implementation of demand-driven points-to analysis for computing the call-graphs of C programs with function pointers. The performance of our system varies substantially across benchmarks - the main factor is how much of the points-to graph must be computed to determine the call-graph. For some benchmarks, only a small part of the points-to graph is needed (e.g *povray*, *emacs* and *gcc*), and here we see more than a 10x speedup. For other benchmarks (e.g. *burlap* and *gimp*), we need to compute most ($> 95\%$) of the points-to graph, and here the demand-driven algorithm is considerably slower, because using the demand-driven algorithm is a slow method of computing the full points-to graph.

1. INTRODUCTION

1.1 Pointer analyses

In programming languages like C, pointers introduce aliasing: different data access paths may denote the same runtime location [1]. While aliasing is very convenient for representing sharing and cycles in complex datastructures, it interferes with data-flow analysis and program optimization. To address this problem, many program analyses have been developed that compute the sets of variables to which a pointer may point; these sets are called points-to sets [2]. Such analyses are said to perform may-alias analysis of pointers [1] or simply pointer analysis.

Recent work on pointer analysis has focused on efficiency [3, 4, 5], tradeoffs between efficiency and accuracy (e.g. the choice of a subset-based [6] or unification-based [7] analysis [8, 9]) or on the cost and benefits of context-sensitivity and polymorphism [10, 11]. Techniques for incremental analyses have been proposed, so that small modifications of the source code require only small recomputation [9, 15]. Although these methods can reduce the cost of typical end-to-end applications of pointer analyses, in the end they perform an exhaustive analysis i.e. they compute the points-to sets for the entire program considered, and cannot utilize a given query to limit the computation.

In this paper we consider the use of demand-driven techniques to reduce the effective cost of points-to analysis. It is natural to ask: why do we need demand-driven points-to analysis – aren't the current points-to systems already fast enough? For example, unification based systems with precision enhancements can perform exhaustive analyses of multi-million lines in about a minute or two [9], and subset-based analyses can perform exhaustive analyses of about a million lines in a few seconds [16].

One answer is that any improvement in speed (and especially space) is welcome – points-to analysis is still too slow and takes too much space for many applications. Another issue is that the fast points-to analysis systems implemented to date have all required substantial implementation effort with innovative techniques and/or sophisticated data-structures. One promise of the demand-driven approach is that we can build small light-weight points-to analyses systems from simple, maintainable, easy-to-implement (or borrow) building blocks, and yet obtain satisfactory performance and accuracy on large programs.

Our view is that the most important applications of demand-driven points-to analysis are in the development of context-sensitive points-to analysis. We see demand-driven points-to analysis as an important building block for constructing efficient, accurate, context-sensitive points-to analysis for multi-million line programs. It is well known that aggressive duplication of contexts can lead to prohibitive performance degradation for a context-sensitive analysis. By restricting attention to contexts that are relevant to a particular query, a demand-driven analysis could potentially support much more aggressive context-duplication (and flow-duplication) policies e.g. beyond the polymorphic points-to analysis considered in [11]. Moreover, we expect that aggressive context-duplication will improve the query-independence of the points-to graph, and that degenerate behavior, where one query requires computation of the entire points-to graph, will be much less likely.

As a first step towards demand-driven points-to analysis, we shall restrict our attention in this paper to context-insensitive, flow-insensitive, Andersen-style points-to analysis. To simplify our presentation, we start with a minimal subset of C. In this setting, points-to analysis can be described by a simple set of deduction rules that define a points-to relation. Each statement of the program is translated into either a fact or a deduction rule. Figure 1 gives a program and the corresponding deduction system. Using this system, we can derive "p may point to x" from fact (1) and rule (2). We present this analysis in more details in Section 2.

1.2 Demand-driven analyses

A demand-driven algorithm is an algorithm that is structured around answering queries. The central design criteria of such algorithms is: perform only computation that contributes to answering the given query. By eliminating unnecessary computation (w.r.t. a given query), significant time and space savings can be achieved. The use of demand-driven techniques for program analysis was introduced by [12]. Many analysis applications can be very naturally formulated around the notion of queries – typically we do not require all information about all program pieces, but instead we only need some portion of it. For example, building the call graph of a C program only requires the points-to sets of function pointer variables, a small subset of the set of pointer variables.

[12] introduces a method for the construction of precise demand-driven algorithms for a class of dataflow analyses. As an application of their method, they introduce a demand-driven analysis for copy constant propagation. Their query language consists of a fact and a program point. For example, if line 42 of a program contained the assignment $x = exp$, then we might ask the query "is x the constant 5 immediately after line 42?". The algorithm would reply `true` if exp was the constant 5. It would reply `false` if exp was a constant other than 5. If exp was the variable y, then it would recursively call itself on the query "is y the constant 5 immediately before line 42?". In the case of copy constant propagation, the answer to any query is either trivial (can be obtained by just looking at the current instruction) or else there is an equivalent query if we go one step backward in the execution of the program. Thus, it is possible to check the truth value of a given query by executing the program in "reverse mode".

1.3 Demand-driven pointer analyses

Consider Figure 1 and the query "what may p point to?". A demand-driven analysis might proceed as follows: (1) to determine what p points to, we process the assignment `p=q` and initiate a new query "what may q point to?"; (2) to find what q points to, we process `q=&x` and determine that q may point may to x; (3) returning to "what may p point to?", we re-process `p=q` and determine that since q may point to x, p may point to x. Note that we have not considered the assignment `r=&x`, which is irrelevant to the query at hand.

The purpose of this work is to study the feasibility of a demand-driven pointer analysis – too see if it is possible and if it is practical. To the best of our knowledge neither has been done before, and moreover, this problem appears to fall outside the scope of [12]. The generic construction described in [12] is restricted to certain classes of interprocedural data-flow problems. Quoting from [12]: "If the problem's flow functions are distributive, the derived demand algorithms provide as precise information as the corresponding exhaustive analysis." In other words, if the flow functions are not distributive, the demand-driven analysis might return results that are *less accurate* than the exhaustive analysis. Again quoting from [12]: "Copy constant propagation is a distributive version of the non-distributive constant propagation analysis with expression evaluation." Thus, the method cannot handle general constant propagation. Concretely suppose that line 42 of a program is $z = x + y$, and we have the query "is z the constant 5 after line 42". This query cannot be expressed using conjunctions of independent queries about x and y immediately before line 42. Pointer analysis is even more problematic. Suppose that line 43 contains the assignment $*u = *v$. The query "p may point to x after line 43" is equivalent to the disjunction of queries "p may point to x before line 43" or "u may point to p and there exists a variable t such that v may point to t and t may point to x, before line 43". As we try to backtrack in the execution of a program, the queries we have to deal with become more and more complex. In contrast, in a copy constant propagation analysis, they can be expressed in a very simple manner. The kind of reasoning by equivalences implicit in the approach of [12] is clearly inadequate for addressing problems such as general constant propagation or pointer analysis.

Our approach is different, and is not based on equivalences of queries before and after statements. Instead, we separate the notion of query and answer: in our setting a query can have the form "what may x point to?" and answers have the form "x may point to y". Central to our work is a complete redesign of the deduction rules for aliasing analysis in which conditions are added to the rules to ensure that we always derive something that is relevant to the pending query. We also add deduction rules to introduce new queries, as illustrated in Figure 2. We remark that this example is a very trivial one. As we shall see when we present our analysis in Section 3, we must enrich the query language to add queries of the form "which variable may point to x?". This is essential for obtaining a true demand-driven analysis, and for our optimality results.

In Section 4 we show that our analysis is equivalent to the exhaustive analysis in the following sense: the results it computes for a query are exactly what an exhaustive analysis would compute for that query. In Section 5 we establish an optimality result: the answer to a query depends on all program components processed by our demand-driven algorithm in the sense that if one of the processed components were to change, it could potentially change the results of the query. The remainder of the paper addresses practicality:

`int x, *p, *q, *r;`	Rules:
`q = &x;`	(1) q may point to x.
`p = q;`	(2) For any variable v, if q may point to v then p may point to v.
`r = &x;`	(3) r may point to x.

Figure 1: Rules for an exhaustive pointer analysis.

`int x, *p, *q, *r;`	Some rules (incomplete list):
`q = &x;`	If we want to know what q may point to then q may point to x.
`p = q;`	If we want to know what p may point to, then we need to know what q may point to. If we want to know what p may point to, for any variable v, if q may point to v then p may point to v.
`r = &x;`	If we want to know what r may point to then r may point to x.

Figure 2: Rules for a demand-driven pointer analysis.

what is the overhead of the demand-driven algorithm, and can the demand-driven algorithm result in significant analysis savings e.g. can we answer interesting queries without inspecting the entire program. To this end, we have designed and implemented two demand-driven call-graph analyses for C (described in Section 6). We present benchmark results in Section 7.

2. ANDERSEN'S POINTER ANALYSIS

To simplify our presentation, we consider a tiny language consisting of primitive assignments involving just the operations * and &. (We note that nested uses of * and & or statements such as `*x=*y` can be reduced using temporary variables.) A programs is a finite set of *assignments* A:

$$A \ ::= \texttt{x=y} \ | \ \texttt{*x=y} \ | \ \texttt{x=*y} \ | \ \texttt{x=\&y}$$

We use a, b, $\cdots$, x, y, z to denote program variables; p and q are typically pointer variables. We write $var(P)$ to denote the set of variables in P. Our representation of programs ignores control constructs and statement order. To define context-insensitive flow-insensitive aliasing analysis, we construct deduction rules from a program as specified in Figure 3. The binary relation $\cdot \hookrightarrow \cdot$ has domain $var(P) \times var(P)$. Intuitively, $x \hookrightarrow y$ means that x may point to y during an execution of P.

In the first rule, the side condition "if `x=&y` in P" indicates that there is an instance of this rule for each occurrence of an assignment of the form `x=&y` in P. The side conditions in the other rules are similarly interpreted. The last rule deals with assignments of the form `*x=y`. It states that for any z and u, if "x may point to z" and "y may point to u" then "z may point to u". The pointer analysis can now be defined as: x may point to y if we can derive $x \hookrightarrow y$. The exhaustive analysis computation proceeds by finding all of the points-to relations that can be derived from the deduction rules.

3. DEMAND-DRIVEN ANALYSIS

In exhaustive pointer analysis, we compute the points-to sets of all program variables. To define a demand-driven pointer analysis, we must first establish the query language. In this paper we use queries of the form "what may p point to?". An alternative might be to use queries of the form "may p point to x?", however this appears to be less useful for compiler or software engineering applications.

3.1 A First Attempt

As we have seen in Figure 2, answering a query such as "what may p point to?" can necessitate answering another query "what may q point to?". More generally, just as points-to relations are accumulated during the exhaustive pointer analysis, the demand-driven analysis accumulates both points-to relations and queries. We shall write $p \hookrightarrow \cdot$ for "what may p point to?", and any points-to relation of the form $p \hookrightarrow z$ is called an *answer* for $p \hookrightarrow \cdot$ (we sometimes say $p \hookrightarrow z$ answers $p \hookrightarrow \cdot$). Figure 4 presents a first attempt at adding queries to the deduction rules for pointer analysis; quantifiers are implicit. Consider the rule corresponding to the assignment `x=&y`. This rule says if we have the query "what may x point to?", then we may derive "x may point to y".

Given one or more initial queries, demand-driven computation proceeds by finding all points-to relations (and queries) that can be derived from the deduction rules and the initial queries. For the example of Figure 2 with the initial query $p \hookrightarrow \cdot$, we derive $p \hookrightarrow x$, $q \hookrightarrow x$, $p \hookrightarrow \cdot$ and $q \hookrightarrow \cdot$. This includes an additional query ("what may q point to?") and the points-to information for p and q. Importantly, nothing is derived about r. Thus, this new set of rules produces a strictly better result than the exhaustive analysis, in the sense that less points-to relations are derived.

However, a fundamental inefficiency arises with these simple rules. Consider the rules corresponding to `*x=y`, and suppose we have the query "what may z point to ?". Now, if x may point to z and y may point to u, then z may point to u. The problem is that in order to determine whether x points to z, we have to initiate the query "what may x point to?" (this reasoning step is capture by rule ($\star$)). Now, computing the points-to set for x is much more than we actually require – in this case we just need to know whether z is in the points-to set for x. This kind of reasoning can lead to useless computation, as illustrated by Figure 5. In this example, the query "what may p point to?" leads to the points-to relation "w may point to r" which is irrelevant to the initial query. In contrast "w may point to p" and "q may point to x" are relevant results for the query – without them the analysis is not correct. We present a formalization of the notion of "relevant" when we discuss optimality in Section 5.

$$\frac{}{x \hookrightarrow y} \quad \text{if } \texttt{x=\&y} \text{ in } P.$$

$$\forall z, \quad \frac{y \hookrightarrow z}{x \hookrightarrow z} \quad \text{if } \texttt{x=y} \text{ in } P.$$

$$\forall z, \forall u, \quad \frac{y \hookrightarrow z \quad z \hookrightarrow u}{x \hookrightarrow u} \quad \text{if } \texttt{x=*y} \text{ in } P.$$

$$\forall z, \forall u, \quad \frac{x \hookrightarrow z \quad y \hookrightarrow u}{z \hookrightarrow u} \quad \text{if } \texttt{*x=y} \text{ in } P.$$

In this example...

```
int x, *p, *q, *r;
q = &x;
p = q;
r = &x;
```

...there are three rules:

$$\frac{}{q \hookrightarrow x} \; ; \; \forall z, \frac{q \hookrightarrow z}{p \hookrightarrow z} \; ; \; \frac{}{r \hookrightarrow x}$$

Figure 3: Deduction rules for Andersen's pointer analysis.

$$\frac{x \hookrightarrow \cdot}{x \hookrightarrow y} \quad \text{if } \texttt{x=\&y} \text{ in } P.$$

$$\left.\begin{array}{c} \dfrac{x \hookrightarrow \cdot}{y \hookrightarrow \cdot} \\[2ex] \dfrac{y \hookrightarrow z \quad x \hookrightarrow \cdot}{x \hookrightarrow z} \end{array}\right\} \text{if } \texttt{x=y} \text{ in } P.$$

$$\left.\begin{array}{c} \dfrac{x \hookrightarrow \cdot}{y \hookrightarrow \cdot} \\[2ex] \dfrac{y \hookrightarrow z \quad x \hookrightarrow \cdot}{z \hookrightarrow \cdot} \\[2ex] \dfrac{y \hookrightarrow z \quad z \hookrightarrow u \quad x \hookrightarrow \cdot}{x \hookrightarrow u} \end{array}\right\} \text{if } \texttt{x=*y} \text{ in } P.$$

$$\left.\begin{array}{c} \dfrac{}{x \hookrightarrow \cdot} \; (\star) \\[2ex] \dfrac{x \hookrightarrow z \quad z \hookrightarrow \cdot}{y \hookrightarrow \cdot} \\[2ex] \dfrac{x \hookrightarrow z \quad y \hookrightarrow u \quad z \hookrightarrow \cdot}{z \hookrightarrow u} \end{array}\right\} \text{if } \texttt{*x=y} \text{ in } P.$$

Figure 4: Deduction rules for a partial demand-driven pointer analysis.

3.2 Fully demand-driven analysis

To avoid the inefficiencies discussed in the last section, we introduce a new kind of query of the form "what variables may point to p?", denoted $\cdot \hookrightarrow p$. Any points-to relation of form $z \hookrightarrow p$ is an *answer* for $p \hookrightarrow \cdot$. Now, when we have a query "what may p point to?", we generate the query "what variables may point to p?". Then, if we find that x may point to p, and there is an assignment of the form `*x=y`, then we generate the query "what may y point to?", and use results from that query to update the points-to relation for p. The full rules are detailed in Figure 6. The main changes are the addition of the rule (+) and rules to answer queries of the form $\cdot \hookrightarrow p$. There are now 15 rules, and no facts – i.e. every rule has a hypothesis. In other words, these rules are "fully demand-driven". Returning to the example from Figure 5, we show two derivations using the fully demand-driven rules in Figure 7. The first derivation shows how, given the query "what can p point to" ($p \hookrightarrow \cdot$), we can derive that $p \hookrightarrow x$. The second derivation shows how, given the query "what variables may point to x" ($\cdot \hookrightarrow x$), we can derive that $r \hookrightarrow x$. We can also derive that $p \hookrightarrow x$ and $q \hookrightarrow x$ from $\cdot \hookrightarrow x$; these are not shown in Figure 7.

4. EQUIVALENCE

The results that we establish in this section and the next are for a single initial query; however they can be easily adapted to deal with multiple queries in a single computation. We first prove that the demand-driven pointer analysis is equivalent to the exhaustive pointer analysis. Specifically, we will show (a) when given the query $x \hookrightarrow \cdot$, the demand-driven analysis computes $x \hookrightarrow y$ iff the exhaustive analysis computes $x \hookrightarrow y$, and (b) when given the query $\cdot \hookrightarrow x$, the demand-driven analysis computes $x \hookrightarrow y$ iff the exhaustive analysis computes $x \hookrightarrow y$. We use $\vdash_{ex(P)} x \hookrightarrow y$ to denote that $x \hookrightarrow y$ can be derived from the set of rules associated with the program P in the exhaustive framework. We use $Q \vdash_{d(P)} x \hookrightarrow y$ to denote when $x \hookrightarrow y$ can be derived from the query Q using the set of rules associated with the program P in the demand-driven framework. We write $\vdash_{ex}$ and $\vdash_d$ when the program P is clear from context.

Lemma 1.
$$\begin{cases} \textit{If } a \hookrightarrow \cdot \vdash_d x \hookrightarrow y \textit{ then } \vdash_{ex} x \hookrightarrow y \\ \textit{If } \cdot \hookrightarrow a \vdash_d x \hookrightarrow y \textit{ then } \vdash_{ex} x \hookrightarrow y \end{cases}$$

Proof: We proceed by induction on the proof of $Q \vdash_d x \hookrightarrow y$ where Q is either $a \hookrightarrow \cdot$ or $\cdot \hookrightarrow a$. The last used rule can be one of the eight rules that add a new points-to relation. Each of these rules reduces to a rule of the exhaustive analysis with less hypotheses, as described in Figure 8.

Let's suppose $Q \vdash_d y \hookrightarrow x$ is derived using the rule $\frac{z \hookrightarrow x \quad u \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}$`*z=u`. In this case we know $Q \vdash_d z \hookrightarrow x$ and $Q \vdash_d u \hookrightarrow y$. By the induction hypothesis we have $\vdash_{ex} z \hookrightarrow x$ and $\vdash_{ex} u \hookrightarrow y$. We can apply the rule $\frac{z \hookrightarrow x \quad u \hookrightarrow y}{x \hookrightarrow y}$`*z=u` of the exhaustive framework, and conclude that $\vdash_{ex} x \hookrightarrow y$. Other cases are similar. $\square$

Lemma 2.
$$\begin{cases} \textit{If } \vdash_{ex} x \hookrightarrow y \textit{ then } x \hookrightarrow \cdot \vdash_d x \hookrightarrow y \\ \textit{If } \vdash_{ex} x \hookrightarrow y \textit{ then } \cdot \hookrightarrow y \vdash_d x \hookrightarrow y \end{cases}$$

Proof: We proceed by induction on the proof of $\vdash_{ex} x \hookrightarrow y$. Depending on the last rule involved in this proof, we split the problem in four cases. In each case we have to prove both parts of the lemma. We show one of these proofs here. The complete proof can be found in Appendix A.

If the last rule used is $\frac{z \hookrightarrow x \quad u \hookrightarrow y}{x \hookrightarrow y}$`*z=u`, let's prove that $x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$.

```
int x, *p, *q, *r, **w;
q = &x;
w = &p;
w = &r;
*w = q;
```

Query: $p \hookrightarrow \cdot$
Derived: $p \hookrightarrow \cdot, w \hookrightarrow \cdot, w \hookrightarrow p, w \hookrightarrow r, q \hookrightarrow \cdot, q \hookrightarrow x, p \hookrightarrow x$
Points-to relations derived: $w \hookrightarrow p, w \hookrightarrow r, q \hookrightarrow x, p \hookrightarrow x$

Optimal demand-driven analysis: $w \hookrightarrow p, q \hookrightarrow x, p \hookrightarrow x$

Figure 5: Inefficiency of the partial demand-driven framework.

$$(+) \qquad \frac{x \hookrightarrow \cdot}{\cdot \hookrightarrow x} \qquad \frac{x \hookrightarrow y \quad \cdot \hookrightarrow y}{\cdot \hookrightarrow x}$$

$$\left.\begin{array}{c} \dfrac{x \hookrightarrow \cdot}{x \hookrightarrow y} \\[2ex] \dfrac{\cdot \hookrightarrow y}{x \hookrightarrow y} \end{array}\right\} \text{if } \texttt{x=\&y} \text{ in } P.$$

$$\left.\begin{array}{c} \dfrac{x \hookrightarrow \cdot}{y \hookrightarrow \cdot} \\[2ex] \dfrac{y \hookrightarrow z \quad x \hookrightarrow \cdot}{x \hookrightarrow z} \\[2ex] \dfrac{y \hookrightarrow z \quad \cdot \hookrightarrow z}{x \hookrightarrow z} \end{array}\right\} \text{if } \texttt{x=y} \text{ in } P.$$

$$\left.\begin{array}{c} \dfrac{x \hookrightarrow \cdot}{y \hookrightarrow \cdot} \\[2ex] \dfrac{y \hookrightarrow z \quad x \hookrightarrow \cdot}{z \hookrightarrow \cdot} \\[2ex] \dfrac{y \hookrightarrow z \quad z \hookrightarrow u \quad x \hookrightarrow \cdot}{x \hookrightarrow u} \\[2ex] \dfrac{y \hookrightarrow z \quad z \hookrightarrow u \quad \cdot \hookrightarrow u}{x \hookrightarrow u} \end{array}\right\} \text{if } \texttt{x=*y} \text{ in } P.$$

$$\left.\begin{array}{c} \dfrac{x \hookrightarrow z \quad z \hookrightarrow \cdot}{y \hookrightarrow \cdot} \\[2ex] \dfrac{x \hookrightarrow z \quad y \hookrightarrow u \quad z \hookrightarrow \cdot}{z \hookrightarrow u} \\[2ex] \dfrac{y \hookrightarrow u \quad \cdot \hookrightarrow u}{x \hookrightarrow \cdot} \\[2ex] \dfrac{x \hookrightarrow z \quad y \hookrightarrow u \quad \cdot \hookrightarrow u}{z \hookrightarrow u} \end{array}\right\} \text{if } \texttt{*x=y} \text{ in } P.$$

Figure 6: Deduction rules for the demand-driven pointer analysis.

```
int x, *p, *q, *r, **w;
q = &x;
w = &p;
w = &r;
*w = q;
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p \hookrightarrow \cdot}{\cdot \hookrightarrow p}}{w \hookrightarrow p}\texttt{w=\&p} \quad p \hookrightarrow \cdot}{q \hookrightarrow \cdot}\texttt{*w=q}}{q \hookrightarrow x}\texttt{q=\&x} \quad \dfrac{\dfrac{p \hookrightarrow \cdot}{\cdot \hookrightarrow p}}{w \hookrightarrow p}\texttt{w=\&p} \quad p \hookrightarrow \cdot}{p \hookrightarrow x}\texttt{*w=q}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot \hookrightarrow x}{q \hookrightarrow x}\texttt{q=\&x} \quad \cdot \hookrightarrow x}{w \hookrightarrow \cdot}\texttt{*w=q}}{w \hookrightarrow r}\texttt{w=\&r} \quad \dfrac{\cdot \hookrightarrow x}{q \hookrightarrow x}\texttt{q=\&x} \quad \cdot \hookrightarrow x}{r \hookrightarrow x}\texttt{*w=q}$$

Figure 7: Two derivations.

$$\frac{x \hookrightarrow \cdot}{x \hookrightarrow y}\texttt{x=\&y} \longrightarrow \frac{}{x \hookrightarrow y}\texttt{x=\&y} \qquad \frac{z \hookrightarrow u \quad u \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}\texttt{x=*z} \longrightarrow \frac{z \hookrightarrow u \quad u \hookrightarrow y}{x \hookrightarrow y}\texttt{x=*z}$$

$$\frac{\cdot \hookrightarrow y}{x \hookrightarrow y}\texttt{x=\&y} \longrightarrow \frac{}{x \hookrightarrow y}\texttt{x=\&y} \qquad \frac{z \hookrightarrow u \quad u \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}\texttt{x=*z} \longrightarrow \frac{z \hookrightarrow u \quad u \hookrightarrow y}{x \hookrightarrow y}\texttt{x=*z}$$

$$\frac{z \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}\texttt{x=z} \longrightarrow \frac{z \hookrightarrow y}{x \hookrightarrow y}\texttt{x=z} \qquad \frac{z \hookrightarrow x \quad u \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}\texttt{*z=u} \longrightarrow \frac{z \hookrightarrow x \quad u \hookrightarrow y}{x \hookrightarrow y}\texttt{*z=u}$$

$$\frac{z \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}\texttt{x=z} \longrightarrow \frac{z \hookrightarrow y}{x \hookrightarrow y}\texttt{x=z} \qquad \frac{z \hookrightarrow x \quad u \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}\texttt{*z=u} \longrightarrow \frac{z \hookrightarrow x \quad u \hookrightarrow y}{x \hookrightarrow y}\texttt{*z=u}$$

Figure 8: Reduction of the rules of the demand-driven analysis to rules of the exhaustive analysis.

$x \hookrightarrow \cdot \vdash_d \cdot \hookrightarrow x$ is derived using rule $\frac{x \hookrightarrow \cdot}{\cdot \hookrightarrow x}$.
$\cdot \hookrightarrow x \vdash_d z \hookrightarrow x$ by induction since $\vdash_{ex} z \hookrightarrow x$.
$x \hookrightarrow \cdot \vdash_d z \hookrightarrow x$.
$x \hookrightarrow \cdot \vdash_d u \hookrightarrow \cdot$ using rule $\frac{z \hookrightarrow x \quad x \hookrightarrow \cdot}{u \hookrightarrow \cdot}$ `*z=u`.
$u \hookrightarrow \cdot \vdash_d u \hookrightarrow y$ by induction since $\vdash_{ex} u \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d u \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$ using rule $\frac{z \hookrightarrow x \quad u \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}$ `*z=u`. $\square$

Combining Lemmas 1 and 2 proves:

THEOREM 1. $\begin{cases} \vdash_{ex} x \hookrightarrow y \text{ iff } x \hookrightarrow \cdot \vdash_d x \hookrightarrow y \\ \vdash_{ex} x \hookrightarrow y \text{ iff } \cdot \hookrightarrow y \vdash_d x \hookrightarrow y \end{cases}$

5. OPTIMALITY

We first show optimality w.r.t. answers: the demand-driven analysis computes a points-to relation only if that relation is an answer to some query (initial or derived).

LEMMA 3. $\begin{cases} \text{If } a \hookrightarrow \cdot \vdash_d x \hookrightarrow y \text{ then either} \\ \quad a \hookrightarrow \cdot \vdash_d x \hookrightarrow \cdot \quad \text{or} \quad a \hookrightarrow \cdot \vdash_d \cdot \hookrightarrow y \\ \text{If } \cdot \hookrightarrow a \vdash_d x \hookrightarrow y \text{ then either} \\ \quad \cdot \hookrightarrow a \vdash_d x \hookrightarrow \cdot \quad \text{or} \quad \cdot \hookrightarrow a \vdash_d \cdot \hookrightarrow y \end{cases}$

This lemma is immediate from Figure 8, which shows that for each of the eight rules with conclusion $x \hookrightarrow y$, either $x \hookrightarrow \cdot$ or $\cdot \hookrightarrow y$ appears as a hypothesis. Intuitively, it proves that the demand-driven analysis computes *at most* the answers to all initial and derived queries. This is dual to Theorem 1, which states that the analysis computes *at least* the answers to all initial and intermediate queries; hence, the analysis computes *exactly* the answers to all queries.

Lemma 3 effectively proves that the demand-driven analysis is optimal *provided that the set of queries generated during the analysis is optimal.* But what does it mean for the set of derived queries to be optimal? Intuitively, we want to show is that answers to the derived queries are needed to correctly answer the initial query.

We first discuss queries of the form $q \hookrightarrow \cdot$. Consider Figure 9 (for brevity, we omit $\cdot \hookrightarrow p$ and $\cdot \hookrightarrow q$ in the derived queries). In the first example, the derived query $q \hookrightarrow \cdot$ is clearly necessary to answer $p \hookrightarrow \cdot$, since we need to look at assignments to q in order to discover that $q \hookrightarrow x$ and therefore that $p \hookrightarrow x$. However, in the second example the usefulness of $q \hookrightarrow \cdot$ is less clear, since the points-to set for q is empty, and so the query $q \hookrightarrow \cdot$ does not in fact contribute to answers for $p \hookrightarrow \cdot$. Of course, there is no way to know this in advanace. To correctly answer $p \hookrightarrow \cdot$, we must look for answers to $q \hookrightarrow \cdot$ i.e. we have to inspect those parts of the program that are relevant to $q \hookrightarrow \cdot$, such as assignments `q = &x`. More generally, the set of derived queries can be viewed as a slice (a subset) of a program. This slice represents the parts of P that the demand-driven algorithm inspects as it tries to answer Q. As we have illustrated, we sometimes inspect a part of the program only to find that in the end it does not contribute to the initial query. Intuitively, if we could not have predicted this outcome until we actually inspected that part of the program, then such an inspection was necessary to obtain correct results.

In the third example in Figure 9, the usefulness of the query $q \hookrightarrow \cdot$ is again unclear, since the answers to $q \hookrightarrow \cdot$ do not lead to new answers for $p \hookrightarrow \cdot$. However, suppose that assignments `q=&w` and `w=&x` were added to the program. These additions generate $p \hookrightarrow x$, a new answer to $p \hookrightarrow \cdot$. Hence $q \hookrightarrow \cdot$ is indeed needed, because it is this query that prompts us to look for potential assignments to q, and without looking for such assignments, we might miss answers for $p \hookrightarrow \cdot$.

We formalize these intuitions using ideas borrowed from program slicing. In program slicing, the value of a program variable is said to *depend* on some program component if changing that component could change the variable's value. Similarly, we shall define that a query Q is relevant to query I for program P if extending the program at Q (i.e. adding assignments that change Q) can change the answers to P.[1] It remains to define what we mean by "extending the program at Q".

As the third example in Figure 9 shows, for a query like $q \hookrightarrow \cdot$, we might need to add more than one assignment to affect a change to the answers for I. Clearly, these assignments cannot be arbitrary: they must at least change the answers to the intermediate query. However, even more restrictions are needed. For example, let P be the program $\{p = \&x\}$, let I be $p \hookrightarrow \cdot$ and Q be $q \hookrightarrow \cdot$. Now, consider extending the program with assignments `q=&p, p=&r`. This changes the answers to both Q and I, and yet Q is not needed to correctly answer I. The main problem in this example is the assignment `p=&r`, which directly modifies the points-to set of p, effectively bypassing the query $q \hookrightarrow \cdot$. To avoid this problem, we require that program extensions must use new variables (in addition to the variable in Q).

Recall that a points-to relation $x \hookrightarrow y$ answers $x \hookrightarrow \cdot$ and $\cdot \hookrightarrow y$. Now, given a program P, initial query I and a derived query Q of the form $q \hookrightarrow \cdot$, define that Q is *relevant*[2] to I if there exists a program extension P_Q such that

(i) P_Q contains only the variable q and new variables,
(ii) q only appears in assignments of form $q = \cdots$, and
(iii) there is a points-to relation that answers I and is derivable from $P \cup P_Q$ but not from P alone.

Condition (ii) is key: it says that the only direct change we can make to P using P_Q is to change what q can point to; we cannot directly change any other component/variable of P.

Next, consider queries of the form $\cdot \hookrightarrow q$. Intuitively, such a query is relevant to initial query I if we can extend the program in such a way that we change the answers to $\cdot \hookrightarrow q$, and in the process alter the answers to I. Again, we may need to add more than one assignment. For example, consider the program `p=*x`, initial query $p \hookrightarrow \cdot$ and derived query $\cdot \hookrightarrow x$. To show that $\cdot \hookrightarrow x$ is relevant to $p \hookrightarrow \cdot$, we need to construct a program like `r=&x, *r=y, y=&z`. As before,

[1] More generally, we could consider arbitrary changes to P at Q, instead of just additions. This change would not alter the notion of optimality obtained; it would however add complexity to the following definitions.

[2] In the definition of relevant, we could add the condition "there is a points-to relation that answers Q and is derivable from $P \cup P_Q$ but not from P alone"; this turns out to be redundant.

1:
```
int x, *p, *q;
q = &x;
p = q;
```
Query: $p \hookrightarrow \cdot$

Derived: $\begin{cases} p \hookrightarrow \cdot, q \hookrightarrow \cdot, \\ q \hookrightarrow x, p \hookrightarrow x \end{cases}$

2:
```
int x, *p, *q;
/* q = &x; */
p = q;
```
Query: $p \hookrightarrow \cdot$

Derived: $p \hookrightarrow \cdot, \overbrace{q \hookrightarrow \cdot}^{Why?}$

3:
```
   int *p, **q;
   p = *q;
/* q = &w; */
/* w = &x; */
```
Query: $p \hookrightarrow \cdot$

Derived: $\begin{cases} p \hookrightarrow \cdot \\ q \hookrightarrow \cdot \end{cases}$

4:
```
int x, y, *r, *q, *p;
r = &y;
q = &x;
p = q;
```
Query: $p \hookrightarrow \cdot$

Derived: $\begin{cases} p \hookrightarrow \cdot, q \hookrightarrow \cdot, \\ q \hookrightarrow x, p \hookrightarrow x \end{cases}$

Figure 9: Ambiguity of the notion of optimality for queries.

we must constrain program extensions to use only new variables (in addition to the variable in the query). Given a program P, initial query I and a derived query Q of the form $\cdot \hookrightarrow q$, define that Q is *relevant* to I if there exists a program extension P_Q such that

(i) P_Q contains only the variable q and new variables,
(ii) q only appears in assignments of form $\cdots = \&q$, and
(iii) there is a points-to relation that answers I and is derivable from $P \cup P_Q$ but not from P along.

Again condition (ii) is key: it says that the only direct change we can make to P using P_Q is to change what variables point to q; we cannot directly change any other component/variable of P. We now define that a demand-driven analysis is *optimal* if it only generates relevant queries i.e. for all programs P, all queries derivable from an initial query I are relevant to I.

THEOREM 2. $\vdash_{d(P)}$ *is optimal: for all initial queries* I *and programs* P, *if* $I \vdash_{d(P)} Q$ *then* Q *is relevant to* I.

The proof of this result appears in Appendix B. The basic strategy is as follows: for each program P, initial query I and intermediate query Q generated, we inductively construct a program extension P_Q that extends P at Q. We note that the proof in Appendix B in fact uses a strengthened induction hypothesis c.f. the statement of Theorem 2.

Returning to the programs in Figure 9, when we apply the construction to program two with the query $q \hookrightarrow \cdot$, we would build an extension of the form `q=&x`. Applying the construction to program three with the query $q \hookrightarrow \cdot$ would yield `q=&w, w=&x`. Finally, consider the fourth program. Clearly the query $r \hookrightarrow \cdot$ is not needed to compute $p \hookrightarrow \cdot$, and the demand-driven algorithm ignores the assignment to r. It is easy to check that $r \hookrightarrow \cdot$ is not relevant to $p \hookrightarrow \cdot$ according to the above definitions. However, suppose that we were to weaken the notion of program extension used to define optimality, and admit an extension that adds the assignment `p=r` to the fourth program. Then, according to this modified definition, the query $r \hookrightarrow \cdot$ would in fact be relevant to the initial query $p \hookrightarrow \cdot$. This shows why the restrictions on program extension are important. In particular, if we attempt to simplify the definition of optimality and admit *arbitrary* program extensions, then, in essence, all queries become relevant and the definition of optimality becomes meaningless. We remark that an earlier draft of this paper attempted to simplify the definition of optimality in exactly this (erroneous) way.

6. APPLICATION

We apply our demand-driven analysis to the problem of computing the call graph of a C program: our implementation computes a safe approximation of the possible functions that may be called from each call-site in a program. In the absence of function pointers, this is trivial to compute. However with function pointers we must perform may-alias analysis of the program in order to obtain a safe approximation of the program's runtime behavior. One approach is to perform a full pointer analysis of the program, but this can be expensive. Alternatively, we may try to handle the common cases such as the use of callback functions as parameters of function calls. Although they are cheaper to run, such custom analyses are easy to break. For example, what if the callback function is a member of a structure? A "one-level" pointer analysis would crash on the following piece of code (excerpted from *povray*):

```
int Iteration_HCompl (VECTOR, FRACTAL *);
...
void SetUp_Fractal(FRACTAL *Fractal) { ...
Fractal->Iteration_Method = Iteration_HCompl; ...
}
```

Our approach is to use demand-driven pointer analysis. The rationale for this approach is that in a typical program there are relatively few function pointers, compared to many multi-purpose pointers. Thus the call graph problem appears likely to benefit from a demand-driven approach.

6.1 Implementation

The starting point of our implementation is an already existing system described in [15, 16]. This system implements an exhaustive pointer analysis for C programs. It consists of three components. The first component is a compiler which translates each module of a preprocessed C code into an "object file", which is a highly indexed database of statements of the kind we defined in Section 2. It is written in SML using the CKit frontend [13]. The second component is a linker that combines the databases into a single object file. The third component reads this object file and performs pointer analysis. Both the linker and analyzer are implemented in C. This architecture, called CLA, as well as the core of the analyzer are described in more details in [15]. This baseline implementation deals with full C including structs, unions, arrays and function call/return (including indirect calls). Support for many of these features is based on simple syntactic transformations in the compiler part of the system. Structs/unions are handled in a field-based manner (we generate a new variable for each field of a struct definition, and then map each access of that field to the variable), arrays are handled in an index-independent manner (we essentially ignore the index component of sub expressions), functions are handled by introducing standardized naming conventions for function arguments and returns, which are linked together by the linker, and indirect function calls are handled by using the same naming convention, but the linking happens during analysis as function pointers are discovered – see [15, 16] for further details on the treatment of structs and functions.

In order to implement our call graph analysis we reuse the compiler, the linker and many modules of the analyzer, especially the set implementation used to build the sets of points-to relations during the analysis. The main modification to the base infrastructure is that we add some additional information about call-sites and function pointers to the object files. In summary, the object files contain information about variables (we consider functions as variables), assignments (`x=&y`, `*x=y`...), function pointers, call-sites (A calls B where A is a function and B a function or a function pointer), as well as various indexes to navigate through these object files. The call graph analyzer proceeds as follows: it computes the demand-driven pointer analysis of the input program where the queries are all call-sites that are not known functions. In essence, if we have a call $*x(e_1, \dots, e_n)$, then we will add the initial query $x \hookrightarrow \cdot$.

6.2 Exhaustive algorithm

Figure 10 describes a basic algorithm for our baseline exhaustive analysis implementation [16]. We use X to denote the points-to set for the program variable x (and similarly Y for y etc.). The exhaustive algorithm has two phases. In the first phase, the *static phase*, we initialize the points-to graph by loading all primitive points-to relations (i.e. relations such as $x \hookrightarrow y$ corresponding to a program assignment `x=&y`). In the second phase, called the *dynamic phase*, we iteratively build the closure of this graph under the derivation rules associated with the assignments of the program. This central component of this system is a very efficient implementation of sets. In addition, the CLA architecture is used to load the object file on demand and reduce space utilization.

6.3 Demand-driven algorithm

Given a program and a collection of queries, our aim is to find an efficient way to compute the queries and points-to relations that are derivable from the rules presented in Figure 6. To this end, we observe that the rules $\frac{x \hookrightarrow \cdot}{\cdot \hookrightarrow x}$ and $\frac{x \hookrightarrow y \quad \cdot \hookrightarrow y}{\cdot \hookrightarrow x}$ are independent of any particular assignment. Thus, we can merge them into the other rules. As a result, we obtain the modified rules in Figure 11. These modified rules are equivalent to the rules in Figure 6 if we assume the following property about the initial queries: if $x \hookrightarrow \cdot$ is an initial query, then so is $\cdot \hookrightarrow x$. In other words, the input queries must be closed under the (+) rule.

During the demand-driven algorithm, a program variable v can be in one of three states:

(1) We have queries $v \hookrightarrow \cdot$ and $\cdot \hookrightarrow v$.
(2) We just have the query $\cdot \hookrightarrow v$.
(3) There is currently no query about v.

	LOC (preproc)	Program Variables	Primitive Assignments	Non-triv Call-sites
nethack	44.1K	3856	10402	11
burlap	74.6K	6859	19022	15
vortex	170.3K	11395	34126	15
emacs	93.5K	12587	36603	103
povray	175.5K	12570	40280	108
gcc	199.8K	18749	69715	138
gimp	7486.7K	131552	344156	5964

Table 1: Benchmarks.

We use X' to denote $\{v : v \in X \text{ and } \cdot \hookrightarrow v\}$ i.e. the "tracked" variables in X. Using this notation, the algorithm for demand-driven pointer analysis is described on Figure 12. We remark that the property $X' \subset X$ is implicit in the algorithm. We refer to this algorithm as the *two-set* algorithm, since it keeps two sets X and X' for a program variable x.

We also define an algorithm that is potentially more efficient than the two-set algorithm, but sacrifices optimality. This algorithm maintains just one set per variable, and so we call it the *one-set* algorithm. In other words, whereas the two-set algorithms maintain two sets X and X' corresponding to variable x, the one-set algorithm only maintains X. Thus, during execution, we may have to rebuild X' from X, which in principle is an expensive operation. The one-set algorithm reduces this cost by using X in place of X' whenever using X' would require nested iterations over sets. The one-set algorithm is strictly less demand-driven that the two-set algorithm. Let E, D_2 and D_1 be respectively the points-to relation (i.e. a set of elements of the form $x \hookrightarrow y$) computed by the exhaustive algorithm, the two-set algorithm and one-set algorithm respectively. These sets are related as follows: $D_2 \subset D_1 \subset E$. In other words, D_1 provides an exact answer to the query but manipulates a potentially larger set of points-to relations than D_2. We shall see that in most cases D_1 and D_2 differ little in size.

7. RESULTS

We compare the three algorithms – the exhaustive algorithm, the two-set algorithm and the one-set algorithm – on the problem of computing the possible functions that may be called from each call-site. The benchmarks we use are described in Table 1. The first six benchmarks were obtained from the authors of [4], the last one by the authors of [5]. The columns report the number of lines of preprocessed code, the number of program variables (excluding temporary variables introduced when mapping C code into primitive assignments), the number of primitive assignments, and the number of non-trivial call-sites respectively. A call-site is non-trivial if it doesn't involve a call to a known function. Introducing fresh variables as necessary, we transform each non-trivial call-site into an expression of form $*x(e_1, \dots, e_n)$, and generate the initial query $x \hookrightarrow \cdot$. Each benchmark is run with the entire set of initial queries for that benchmark i.e. we run the analysis on all initial queries at once.

Table 2 summarizes our results, obtained using a single processor of an 8 processor 250MHz MIPS R10000 machine with 4GB of memory. For each algorithm and each benchmark,

For each assignment of the form `x=&y` insert y in X (we note X for the points-to set of x).
Repeat:
 for each assignment of the form `x=y` insert the elements of Y in X.
 for each assignment of the form `*x=y`, for each variable v in X, insert the elements of Y in V.
 for each assignment of the form `x=*y`, for each variable v in Y, insert the elements of V in X.
Until a no change.

Figure 10: Algorithm for the exhaustive pointer analysis.

$$\left.\begin{array}{c}\dfrac{x\hookrightarrow\cdot}{x\hookrightarrow y}\\[2ex]\dfrac{\cdot\hookrightarrow y}{x\hookrightarrow y\quad \cdot\hookrightarrow x}\end{array}\right\}\text{ if } \texttt{x=\&y} \text{ in } P. \qquad \left.\begin{array}{c}\dfrac{x\hookrightarrow\cdot}{y\hookrightarrow\cdot\quad \cdot\hookrightarrow y}\\[2ex]\dfrac{y\hookrightarrow z\quad x\hookrightarrow\cdot}{x\hookrightarrow z}\\[2ex]\dfrac{y\hookrightarrow z\quad \cdot\hookrightarrow z}{x\hookrightarrow z\quad \cdot\hookrightarrow x}\end{array}\right\}\text{ if } \texttt{x=y} \text{ in } P.$$

$$\left.\begin{array}{c}\dfrac{x\hookrightarrow\cdot}{y\hookrightarrow\cdot\quad \cdot\hookrightarrow y}\\[2ex]\dfrac{y\hookrightarrow z\quad x\hookrightarrow\cdot}{z\hookrightarrow\cdot\quad \cdot\hookrightarrow z}\\[2ex]\dfrac{y\hookrightarrow z\quad z\hookrightarrow u\quad x\hookrightarrow\cdot}{x\hookrightarrow u}\\[2ex]\dfrac{y\hookrightarrow z\quad z\hookrightarrow u\quad \cdot\hookrightarrow u}{x\hookrightarrow u\quad \cdot\hookrightarrow x}\end{array}\right\}\text{ if } \texttt{x=*y} \text{ in } P. \qquad \left.\begin{array}{c}\dfrac{x\hookrightarrow z\quad z\hookrightarrow\cdot}{y\hookrightarrow\cdot\quad \cdot\hookrightarrow y}\\[2ex]\dfrac{x\hookrightarrow z\quad y\hookrightarrow u\quad z\hookrightarrow\cdot}{z\hookrightarrow u}\\[2ex]\dfrac{y\hookrightarrow u\quad \cdot\hookrightarrow u}{x\hookrightarrow\cdot\quad \cdot\hookrightarrow x}\\[2ex]\dfrac{x\hookrightarrow z\quad y\hookrightarrow u\quad \cdot\hookrightarrow u}{z\hookrightarrow u\quad \cdot\hookrightarrow z}\end{array}\right\}\text{ if } \texttt{*x=y} \text{ in } P.$$

Figure 11: Deduction rules for the demand-driven pointer analysis after inlining.

For each query of the form $x\hookrightarrow\cdot$, add the query $\cdot\hookrightarrow x$.
Repeat:
 for each `x=&y`: if $x\hookrightarrow\cdot$ then insert y in X.
  if $\cdot\hookrightarrow y$ then insert y in X' and add $\cdot\hookrightarrow x$.
 for each `x=y`: if $x\hookrightarrow\cdot$ then add $y\hookrightarrow\cdot$, $\cdot\hookrightarrow y$.
  if $x\hookrightarrow\cdot$ then insert all elements of Y in X.
  if Y' not empty then insert all elements of Y' in X' and add $\cdot\hookrightarrow x$.
 for each `x=*y`: if $x\hookrightarrow\cdot$ then add $y\hookrightarrow\cdot$ and $\cdot\hookrightarrow y$.
  if $x\hookrightarrow\cdot$ then for each z in Y, add $z\hookrightarrow\cdot$ and $\cdot\hookrightarrow z$.
  if $x\hookrightarrow\cdot$ then for each z in Y, insert all elements of Z in X.
  for each z in Y', if Z' not empty then insert all elements of Z' in X' and add $\cdot\hookrightarrow x$.
 for each `*x=y`: for each z in X', if $z\hookrightarrow\cdot$ then $y\hookrightarrow\cdot$ and $\cdot\hookrightarrow y$.
  for each z in X', if $z\hookrightarrow\cdot$ then insert all elements of Y in Z.
  if Y' not empty then add $x\hookrightarrow\cdot$ and $\cdot\hookrightarrow x$.
  if Y' not empty then for each z in X, insert all elements of Y' in Z' and add $\cdot\hookrightarrow z$.
Until no change.

Figure 12: Algorithm for the two-set demand-driven pointer analysis.

	Exhaustive analysis			One-set algorithm			Two-set algorithm		
	$\hookrightarrow$ count	utime	cycles	$\hookrightarrow$ count	utime	cycles	$\hookrightarrow$ count	utime	cycles
nethack	7248	0.01s	4.4M	26	0.00s	1.9M	26	0.00s	1.9M
burlap	201240	0.10s	29.5M	195567	0.60s	231.1M	195297	1.94s	684.6M
vortex	392785	0.24s	74.4M	23	0.01s	5.8M	23	0.01s	5.8M
emacs	11232360	1.25s	396.4M	106	0.01s	5.4M	106	0.01s	5.4M
povray	141541	0.12s	35.9M	61	0.01s	4.5M	61	0.01s	4.5M
gcc	123826	0.31s	118.1M	44	0.00s	3.7M	44	0.01s	3.7M
gimp	15298727	1.83s	485.6M	14715288	23.18s	5038.5M	14707976	69.77s	10591.6M

Table 2: Results.

we report three numbers. The first is the sum of the sizes of the points-to sets for all program objects in the original program (this excludes the points-to sets of temporary variables introduced during the analysis; we do not include the count of queries generated by the algorithm). The second is the user times as reported by `ssusage` when our system is compiled using `cc -Ofast`. Since these numbers are very small we report a third number which is the total computed cycles as reported by `pixie` when our system is compiled by `cc -O3` (we cannot run `pixie` with `cc -Ofast` compilation).

For *burlap* and *gimp*, the demand-driven analysis turns out to essentially require an analysis of the entire program. Thus the demand-driven algorithms are much slower than our optimized algorithm and they require more memory (this slow down represents the administrative overheads inherent in the demand-driven approach, and the fact that the demand-driven analysis is not tuned to perform an exhaustive analysis). In the case of *gimp*, the exhaustive analysis requires about 12MB, the one-set demand-driven analysis about 20MB and the two-set implementation 23MB. In the remaining five cases, we obtain very significant speedups ranging from 8 to 60, with the exception of the *nethack* benchmark which is too small to show any major improvement. These five benchmarks consume less that 5MB and almost all of this is statically allocated. We note that the two-set algorithm, while more expensive, does not provide a significant improvement in the size of the set of answers computed. This has been confirmed by other experiments not reported here.

8. DISCUSSION

We have developed a demand-driven technique for context-insensitive, flow-insensitive Andersen-style points-to analysis, established its correctness, and shown that it is optimal in the sense that we take maximum advantage of the initial queries to reduce the work performed by the analysis. We have also implemented this algorithm to understand the administrative overheads of the demand-driven approach, and to study variations that trade optimality for simpler demand-driven mechanisms.

Our results indicate that the administrative overheads of the demand-driven algorithm are substantial – sometimes up to a factor of 30 or more, although they can be greatly reduced by trading optimality for overhead. In practice, for the call-graph problem investigated in the paper, we see two kinds of behavior: either the demand-driven approach leads to very substantial reduction in the points-to relations that must be computed ($< 1\%$), or else the demand-driven approach does not help, and we essentially need to compute the entire points-to graph ($> 95\%$). In the former case, we see significant performance advantages; in the latter case, since we are essentially computing the entire points-to graph, we see large performance degradation. We conjecture that for many compiler optimization problems, if the demand-driven analysis takes a long time, then we will probably finish up computing most of the the points-to graph, and further, the call-site sets will probably be large and not very useful. This suggests using a strategy of running the demand-driven approach with a time-out mechanism, and if the timeout is reached, assuming worst case information about call-sites (which we argue is what would probably be returned if we'd allowed the analysis to complete).

As discussed in Section 1, we see the greatest benefit of demand-driven techniques in the development of context-sensitive points-to analysis. In addition, we expect that context-sensitivity will significantly reduce the likelihood of the degenerate behavior of demand-driven analysis we have just observed in some benchmarks.

Acknowledgements: Thanks to Andreas Podelski and the anonymous referees for their comments, and to Satish Chandra and Jeff Foster for access to their respective systems and benchmarks.

9. REFERENCES

[1] A. Deutsch, "Interprocedural May-Alias Analysis for Pointers: Beyond-k limiting", *PLDI* 1994.

[2] M. Emami, R. Ghiya and L. Hendren, "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers", *PLDI* 1994.

[3] M. Fahndrich, J. Foster, Z. Su and A. Aiken, "Partial Online Cycle Elimination in Inclusion Constraint Graphs", *PLDI* 1998.

[4] A. Rountev and S. Chandra, "Off-line Variable Substitution for Scaling Points-to Analysis", *PLDI* 2000.

[5] Z. Su, M. Fahndrich and A. Aiken, "Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs", *POPL* 2000.

[6] L. Andersen, "Program Analysis and Specialization for the C Programming Language", PhD. thesis, DIKU, University of Copenhagen, May 1994, DIKU report 94/19.

[7] B. Steensgaard, "Points-to Analysis in Almost Linear Time", *POPL* 1996.

[8] M. Shapiro and S. Horwitz, "Fast and Accurate Flow-Insensitive Points-To Analysis", *POPL* 1997.

[9] M. Das, "Unification-Based Pointer Analysis with Directional Assignments", *PLDI* 2000.

[10] E. Ruf, "Context-insensitive alias analysis reconsidered", *PLDI* 1995.

[11] J. Foster, M. Fahndrich and A. Aiken, "Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C", *SAS* 2000.

[12] E. Duesterwald, R. Gupta and M. Soffa, "Demand-driven Computation of Interprocedural Data flow", *POPL* 1995.

[13] S. Chandra, N. Heintze, D. MacQueen, D. Oliva and M. Siff, "ckit: an extensible C frontend in ML", available as an SML/NJ library.

[14] "Programming Languages - C", *ISO/IEC* 9899:1990, Internation Standard, 1990.

[15] N. Heintze, "Analysis of Large Code Bases: The Compile-Link-Analyze Model", unpublished report, November 1999.

[16] N. Heintze, O. Tardieu, "Ultra-fast Pointer Analysis using CLA: A Million Lines of C Code in a Second", PLDI 2001 (this volume).

Appendix A: Equivalence Proof

We prove Lemma 2: If $\vdash_{ex} x \hookrightarrow y$ then $x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$
If $\vdash_{ex} x \hookrightarrow y$ then $\cdot \hookrightarrow y \vdash_d x \hookrightarrow y$

The proof is by induction on the derivation $\vdash_{ex} x \hookrightarrow y$. We consider four cases, based on the last rule used in the derivation.

Rule $\dfrac{}{x \hookrightarrow y}\mathtt{x=\&y}$.

$x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$ is derived using rule $\frac{x \hookrightarrow \cdot}{x \hookrightarrow y}\mathtt{x=\&y}$.
$\cdot \hookrightarrow y \vdash_d x \hookrightarrow y$ is derived using rule $\frac{\cdot \hookrightarrow y}{x \hookrightarrow y}\mathtt{x=\&y}$.

Rule $\dfrac{z \hookrightarrow y}{x \hookrightarrow y}\mathtt{x=z}$.

$x \hookrightarrow \cdot \vdash_d z \hookrightarrow \cdot$ is derived using rule $\frac{x \hookrightarrow \cdot}{z \hookrightarrow \cdot}\mathtt{x=z}$.
$z \hookrightarrow \cdot \vdash_d z \hookrightarrow y$ by induction since $\vdash_{ex} z \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d z \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$ using the rule $\frac{z \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}\mathtt{x=z}$.

$\cdot \hookrightarrow y \vdash_d z \hookrightarrow y$ is obtained by induction since $\vdash_{ex} z \hookrightarrow y$.
$\cdot \hookrightarrow y \vdash_d x \hookrightarrow y$ using the rule $\frac{z \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}\mathtt{x=z}$.

Rule $\dfrac{z \hookrightarrow u \quad u \hookrightarrow y}{x \hookrightarrow y}\mathtt{x=*z}$.

$x \hookrightarrow \cdot \vdash_d z \hookrightarrow \cdot$ is derived using rule $\frac{x \hookrightarrow \cdot}{z \hookrightarrow \cdot}\mathtt{x=*z}$.
$z \hookrightarrow \cdot \vdash_d z \hookrightarrow u$ by induction since $\vdash_{ex} z \hookrightarrow u$.
$x \hookrightarrow \cdot \vdash_d z \hookrightarrow u$.
$x \hookrightarrow \cdot \vdash_d u \hookrightarrow \cdot$ using rule $\frac{z \hookrightarrow u \quad x \hookrightarrow \cdot}{u \hookrightarrow \cdot}\mathtt{x=*z}$.
$u \hookrightarrow \cdot \vdash_d u \hookrightarrow y$ by induction since $\vdash_{ex} u \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d u \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$ using the rule $\frac{z \hookrightarrow u \quad u \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}\mathtt{x=*z}$.

$\cdot \hookrightarrow y \vdash_d u \hookrightarrow y$ is obtained by induction since $\vdash_{ex} u \hookrightarrow y$.
$\cdot \hookrightarrow y \vdash_d \cdot \hookrightarrow u$ using rule $\frac{u \hookrightarrow y \quad \cdot \hookrightarrow y}{\cdot \hookrightarrow u}$.
$\cdot \hookrightarrow u \vdash_d z \hookrightarrow u$ by induction since $\vdash_{ex} z \hookrightarrow u$.
$\cdot \hookrightarrow y \vdash_d z \hookrightarrow u$.
$\cdot \hookrightarrow y \vdash_d x \hookrightarrow y$ using the rule $\frac{z \hookrightarrow u \quad u \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}\mathtt{x=*z}$.

Rule $\dfrac{z \hookrightarrow x \quad u \hookrightarrow y}{x \hookrightarrow y}\mathtt{*z=u}$.

$x \hookrightarrow \cdot \vdash_d \cdot \hookrightarrow x$ is derived using rule $\frac{x \hookrightarrow \cdot}{\cdot \hookrightarrow x}$.
$\cdot \hookrightarrow x \vdash_d z \hookrightarrow x$ by induction since $\vdash_{ex} z \hookrightarrow x$.
$x \hookrightarrow \cdot \vdash_d z \hookrightarrow x$.
$x \hookrightarrow \cdot \vdash_d u \hookrightarrow \cdot$ using rule $\frac{z \hookrightarrow x \quad x \hookrightarrow \cdot}{u \hookrightarrow \cdot}\mathtt{*z=u}$.
$u \hookrightarrow \cdot \vdash_d u \hookrightarrow y$ by induction since $\vdash_{ex} u \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d u \hookrightarrow y$.
$x \hookrightarrow \cdot \vdash_d x \hookrightarrow y$ using the rule $\frac{z \hookrightarrow x \quad u \hookrightarrow y \quad x \hookrightarrow \cdot}{x \hookrightarrow y}\mathtt{*z=u}$.

$\cdot \hookrightarrow y \vdash_d u \hookrightarrow y$ is obtained by induction since $\vdash_{ex} u \hookrightarrow y$.
$\cdot \hookrightarrow y \vdash_d z \hookrightarrow \cdot$ using rule $\frac{u \hookrightarrow y \quad \cdot \hookrightarrow y}{z \hookrightarrow \cdot}\mathtt{*z=u}$.
$z \hookrightarrow \cdot \vdash_d z \hookrightarrow x$ by induction since $\vdash_{ex} z \hookrightarrow x$.
$\cdot \hookrightarrow y \vdash_d z \hookrightarrow x$.
$\cdot \hookrightarrow y \vdash_d x \hookrightarrow y$ using the rule $\frac{z \hookrightarrow x \quad u \hookrightarrow y \quad \cdot \hookrightarrow y}{x \hookrightarrow y}\mathtt{*z=u}$. ∎

Appendix B: Proof of Optimality (Theorem 2)

Let P be a program, V be the variables in P, x a variable of P and z a new variable ($x \in V$, $z \notin V$). We prove the following induction hypothesis (IH), for all $y \in V$:

(i) If $x \hookrightarrow \cdot \vdash_d y \hookrightarrow \cdot$ then $\exists u \notin V$ and a program P_u^y s.t. $x \hookrightarrow \cdot \vdash_{d(P \cup P_u^y \cup \{y=\&u\})} x \hookrightarrow z$ and $V \cap var(P_u^y) = \emptyset$.
(ii) If $x \hookrightarrow \cdot \vdash_d \cdot \hookrightarrow y$ then $\exists u \notin V$ and a program Q_u^y s.t. $x \hookrightarrow \cdot \vdash_{d(P \cup Q_u^y \cup \{u=\&y\})} x \hookrightarrow z$ and $V \cap var(Q_u^y) = \emptyset$.

We proceed by structural induction on the proof of $x \hookrightarrow \cdot \vdash_d y \hookrightarrow \cdot$ or $x \hookrightarrow \cdot \vdash_d \cdot \hookrightarrow y$. There are eight cases:

1. $x \equiv y$: In this case, $x \hookrightarrow \cdot \vdash_{d(P \cup \{x=\&z\})} x \hookrightarrow z$, and so $u \equiv z$ and $P_z^x \equiv \emptyset$ proves IH.
2. The last rule is $\frac{y \hookrightarrow \cdot}{\cdot \hookrightarrow y}$. By IH, there exist $v \notin V$ and P_v^y such that $x \hookrightarrow \cdot \vdash_{d(P \cup P_v^y \cup \{y=\&v\})} x \hookrightarrow z$ and $V \cap var(P_v^y) = \emptyset$. Let $Q_u^y \equiv P_v^y \cup \{*u = w, w = \&v\}$ where w is a new variable.
   $x \hookrightarrow \cdot \vdash_{d(P \cup P_v^y \cup \{y=\&v\})} x \hookrightarrow z$ by IH.
   $\vdash_{ex(P \cup P_v^y \cup \{y=\&v\})} x \hookrightarrow z$. by Thm 1.
   $y \hookrightarrow v \vdash_{ex(P \cup P_v^y)} x \hookrightarrow z$.
   since the only rule for $y = \&v$ is $\frac{}{y \hookrightarrow v}$.
   $\vdash_{ex(\{*u=w, w=\&v, u=\&y\})} y \hookrightarrow v$.
   $\vdash_{ex(P \cup P_v^y \cup \{*u=w, w=\&v, u=\&y\})} x \hookrightarrow z$ by last 2 lines.
   $\vdash_{ex(P \cup Q_u^y \cup \{u=\&y\})} x \hookrightarrow z$.
   $x \hookrightarrow \cdot \vdash_{d(P \cup Q_u^y \cup \{u=\&y\})} x \hookrightarrow z$ by Thm 1
   Consequently, there exist $u \notin V$ and Q_u^y such that $x \hookrightarrow \cdot \vdash_{d(P \cup Q_u^y \cup \{u=\&y\})} x \hookrightarrow z$ and $V \cap var(Q_u^y) = \emptyset$. The arguments for the remaining cases are similar: we outline the main construction in each case.
3. The last rule is $\frac{y \hookrightarrow a \quad \cdot \hookrightarrow a}{\cdot \hookrightarrow y}$. Apply the induction hypothesis to $\cdot \hookrightarrow a$ and let Q_v^a be the program obtained, where v is a new variable. Now, define Q_u^y to be $Q_v^a \cup \{v = *u\}$, and by using $y \hookrightarrow a$ we can prove IH for $\cdot \hookrightarrow y$.
4. Rule $\frac{a \hookrightarrow \cdot}{y \hookrightarrow \cdot}\mathtt{a=y}$. Define P_u^y to be P_u^a (the result of applying IH to $a \hookrightarrow \cdot$).
5. Rule $\frac{a \hookrightarrow \cdot}{y \hookrightarrow \cdot}\mathtt{a=*y}$. Define P_u^y to be $P_v^a \cup \{u = \&v\}$, where P_v^a is the result of applying IH to $a \hookrightarrow \cdot$.
6. Rule $\frac{b \hookrightarrow y \quad a \hookrightarrow \cdot}{y \hookrightarrow \cdot}\mathtt{a=*b}$. Define P_u^y to be P_u^a (the result of applying IH to $a \hookrightarrow \cdot$).
7. Rule $\frac{a \hookrightarrow b \quad \cdot \hookrightarrow b}{y \hookrightarrow \cdot}\mathtt{*y=a}$. Define P_u^y to be Q_u^b (the result of applying IH to $\cdot \hookrightarrow b$).
8. Rule $\frac{a \hookrightarrow b \quad b \hookrightarrow \cdot}{y \hookrightarrow \cdot}\mathtt{*a=y}$. Define P_u^y to be P_u^b (the result of applying IH to $b \hookrightarrow \cdot$).

For each program P, each initial query of the form "what may x point to?" and each intermediate query that can be derived from the initial query, we have show how to build an extension P' such that in $P \cup P'$ we have "x may point to z", where z being a new variable. Moreover, in this extension, any new points-to relation that can be derived about x, has to be derived through the intermediate query.

We want the same result for initial queries of the form "which variable may point to x?". Let P be a program, V the set of variables of P, x a variable of P and z a new variable ($x \in V$, $z \notin V$). Again using induction, we can prove, $\forall y \in V$:

(i) If $\cdot \hookrightarrow x \vdash_d y \hookrightarrow \cdot$ then $\exists u \notin V$ and a program P_u^y s.t. $\cdot \hookrightarrow x \vdash_{d(P \cup P_u^y \cup \{y=\&u\})} z \hookrightarrow x$ and $V \cap var(P_u^y) = \emptyset$
(ii) If $\cdot \hookrightarrow x \vdash_d \cdot \hookrightarrow y$ then $\exists u \notin V$ and a program Q_u^y s.t. $\cdot \hookrightarrow x \vdash_{d(P \cup Q_u^y \cup \{u=\&y\})} z \hookrightarrow x$ and $V \cap var(Q_u^y) = \emptyset$

If $x \equiv y$ then $u \equiv z$ and $Q_z^x \equiv \emptyset$ are solutions. Otherwise we apply the previous construction. ∎