# Estimating the Impact of Scalable Pointer Analysis on Optimization

Manuvir Das[1], Ben Liblit[2], Manuel Fähndrich[1], and Jakob Rehof[1]
1: Microsoft Research 2: EECS Department, UC Berkeley
{manuvir,maf,rehof}@microsoft.com, liblit@eecs.berkeley.edu

**Abstract.** This paper addresses the following question: Do scalable control-flow-insensitive pointer analyses provide the level of precision required to make them useful in compiler optimizations?
We first describe alias frequency, a metric that measures the ability of a pointer analysis to determine that pairs of memory accesses in C programs cannot be aliases. We believe that this kind of information is useful for a variety of optimizations, while remaining independent of a particular optimization. We show that control-flow and context insensitive analyses provide the same answer as the best possible pointer analysis on at least 95% of all statically generated alias queries. In order to understand the potential run-time impact of the remaining 5% queries, we weight the alias queries by dynamic execution counts obtained from profile data. Flow-insensitive pointer analyses are accurate on at least 95% of the weighted alias queries as well.
We then examine whether scalable pointer analyses are inaccurate on the remaining 5% alias queries because they are context-insensitive. To this end, we have developed a new context-sensitive pointer analysis that also serves as a general engine for tracing the flow of values in C programs. To our knowledge, it is the first technique for performing context-sensitive analysis with subtyping that scales to millions of lines of code. We find that the new algorithm does not identify fewer aliases than the context-insensitive analysis.

## 1 Introduction

Programs written in C typically make widespread use of pointer variables. In order to analyze a program that uses pointers, it is necessary to perform a pointer analysis that computes, at every dereference point in a program, a superset of the set of memory locations that may be accessed by the dereference. These "points-to" sets can be used to perform alias analysis in an optimizing compiler: two memory accesses whose points-to sets do not intersect cannot be aliases. Alias information can be utilized by a variety of optimizations, including but not limited to code scheduling, register allocation, loop unrolling and constant propagation.

Over the years a wide variety of algorithms for pointer analysis have been proposed (including [LR92,And94,EGH94,WL95,Ste96,LH99]). All of these algorithms either do not scale to large programs, or are believed to produce poor

alias information. This is one reason why most optimizing compilers do not perform global pointer analysis, and are therefore forced to make conservative assumptions about potential aliases. In this paper, we argue that scalable pointer analyses do produce precise alias information.

We are interested in determining whether scalable pointer analyses can impact a variety of optimizations. Therefore, we avoid evaluating pointer analyses in the context of a specific optimization and a specific compiler. Instead, we develop a new metric, "alias frequency", that measures the frequency with which a pointer analysis is forced to assert that a pair of statically generated memory accesses in a C program may be aliases. Our experiments show that the alias frequency of scalable pointer analyses (in particular, Das's algorithm [Das00]) is within 5% of the alias frequency of the best possible pointer analysis.

Although this result is extremely encouraging, we must also consider whether the 5% alias queries on which scalable pointer analyses are imprecise may be the very queries that have the greatest impact on a given optimization. If this is so, the code associated with these queries must dominate the run-time of the programs. Then, if we weight the responses to alias queries by dynamic execution counts from profile data, we should expect a large gap in alias frequency between Das's algorithm and the best possible pointer analysis. However, our experiments show that Das's algorithm is within 5% of the best possible pointer analysis in terms of weighted alias frequency as well.

One possible source of the remaining inaccuracy in Das's algorithm is its lack of context-sensitivity. To understand the impact of this limitation, we have developed a new algorithm that is a context-sensitive version of Das's algorithm. Our generalized one level flow (GOLF) algorithm uses the one level flow idea from [Das00] to achieve a limited form of context-sensitivity in addition to subtyping. Our results show no appreciable decrease in alias frequency from context-sensitivity.

GOLF is a general engine for tracing the flow of values in programs with pointers and indirect function calls in a context-sensitive manner. It can be used for applications such as program slicing [Tip95] or escape analysis [Ruf00]. GOLF is the first context-sensitive analysis with subtyping that scales to millions of lines of code. Even though GOLF does not improve alias frequency, it can provide much more precise results than a context-insensitive algorithm if the client of the analysis is itself context-sensitive (see Section 5.2).

In summary, we make the following contributions:

- We present "alias frequency", a new metric for measuring the impact of pointer analysis on optimization.
- We demonstrate that scalable pointer analyses are able to produce precise responses to at least 95% of all alias queries on all of our test programs.
- We show that the addition of context-sensitivity does not improve the alias frequency of scalable pointer analyses.
- We present GOLF, a new flow-insensitive pointer analysis that utilizes a limited amount of context-sensitivity and subtyping. It produces a points-to graph that is linear in the size of the program, in almost linear time.

All points-to sets in the program can be extracted from the graph using CFL-Reachability, in worst-case cubic time.
– We show that on all of our test programs, GOLF is linear in time and space requirements. We also show that the limited forms of context-sensitivity and subtyping used in GOLF provide the same precision as algorithms with full polymorphism and subtyping. We therefore claim that GOLF is likely to provide the same precision as an algorithm with full polymorphic subtyping.

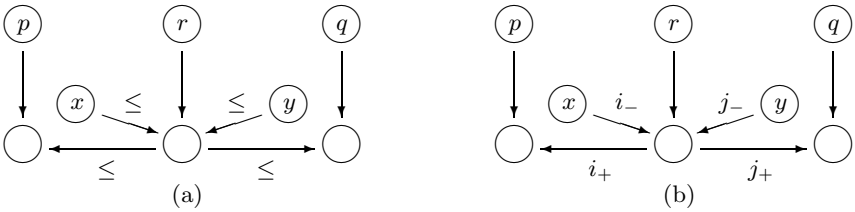
The rest of the paper is organized as follows: in Section 2, we motivate GOLF through an example. We describe GOLF in Section 3. In Section 4, we present alias frequency. In Section 5, we present our empirical results. We discuss related work in Section 6, and conclude in Section 7.

## 2 Example

Consider the fragment of a C program with function calls shown below.

```
id(r) { return r; }
p = id_i(&x);
q = id_j(&y);
*p = 3;
```

The goal of a context-sensitive pointer analysis is to avoid confusing the addresses returned from the function *id* to the variables $p$ and $q$ at the two calls to *id*.




The points-to information computed by Das's algorithm is shown in (a) above. The points-to graph shown contains nodes representing memory locations and edges representing pointer relationships. Every node contains a single pointer edge. Thus, the target of the node for $p$ represents the location $*p$. The points-to graph includes special "flow" edges (labeled $\leq$) between nodes. Flow edges are introduced at assignments, one level below (in the points-to graph) the expressions involved in the assignment. In the example program, the implicit assignment from $\&x$ to parameter $r$ induced by the function call introduces a flow edge from the node for $x$ to the pointer target node of $r$, indicating that the set of symbols represented by $*r$ must include $x$. The return statement in *id* induces implicit assignments from $r$ to $p$ and from $r$ to $q$. As a result, the set of symbols represented by $*p$ includes both $x$ and $y$, even though there is no execution of the program in which the address of $y$ flows to $p$. As has been pointed out by

several authors in previous work [RHS95,RF01], the problem arises because a value flowing in to *id* from one call site ($i$) is allowed to flow out to a *different* call site ($j$).

The points-to graph produced by GOLF is shown in (b) above. We label flow edges arising from function calls with identifiers. All edges from a given call site have the same identifier. Edges also have a polarity, indicating whether a value is flowing in to ($-$) or out from ($+$) the called function. From this graph, we can see that $x$ need not be included in the set of symbols at $*q$, because the only path from $x$ to $*q$ has an edge labeled $i_-$ followed by an edge labeled $j_+$. In a "valid" flow path, calls and returns are "matched": an edge labeled $i_-$ may be matched only by an edge labeled $i_+$.

A valid path in the GOLF graph is one whose sequence of labels forms a string in a context-free language of matched $l_-$ and $l_+$ labels. It is well known that the presence of valid paths between a pair of nodes can be determined in worst-case cubic time using CFL-Reachability queries [RHS95].

Both Das's algorithm and GOLF achieve scaling partly by limiting the use of flow edges to one level in the points-to graph, while using unification (or, type equality rules) to merge nodes at lower levels. Our experiments show that the restriction of context-sensitivity to one level does not lead to loss of precision compared to full context-sensitivity.

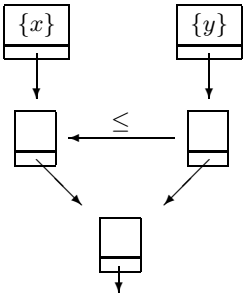
## 3 GOLF: Generalized One Level Flow

A pointer analysis can be thought of as an abstract computation that models memory locations. Every location $\tau$ is associated with an id or set of symbols $\varphi$, and holds some contents $\alpha$ (an abstract pointer value) (Figure 1 (b)). A location "points-to" another if the contents of the former is a pointer to the latter. Information about locations can be encoded as a points-to graph, in which nodes represent locations and edges represent points-to relationships.

In Steensgaard's unification-based algorithm [Ste96], the effect of an assignment from $y$ to $x$ is to equate the contents of the locations associated with $y$ and $x$. This is achieved by unifying (*i.e.*, equating their ids and contents) the locations pointed-to by $y$ and $x$ into one representative location. Das's algorithm extends Steensgaard's algorithm by pushing the effect of assignment processing one level down the chains in the points-to graph (Figure 1). The effect of an assignment from $y$ to $x$ is to introduce a special "flow" edge from the pointed-to location of $y$ to the pointed-to location of $x$, and to equate *only the contents* of the two pointed-to locations (Figure 1 (a)). Flow edges relate ids of locations: all of the symbols in the id of the source of a flow edge must be included in the id of the target of the edge. Assignment processing is represented declaratively in Figure 1 (b): the type rule says that the program is correctly typed iff the pointed-to locations of $y$ and $x$ have the same contents, and if the id of the pointed-to location of $y$ is a subset of the id of the pointed-to location of $x$.

GOLF extends Das's algorithm by treating implicit assignments induced by function calls in a special manner, so as to obtain context-sensitive information.

(a)




(b)

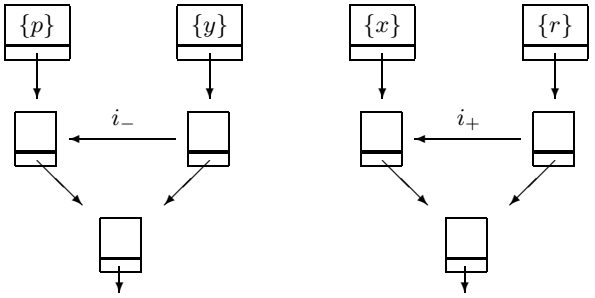
$s \in \mathit{Symbols}$
$\tau \in \mathit{Locations} \ ::= \ (\varphi, \alpha)$
$\varphi \in \mathit{Ids} \ ::= \ \{s_1, \ldots, s_n\}$
$\alpha \in \mathit{Values} \ ::= \ \bot \mid \mathit{ptr}(\tau)$

$\bot \leq \alpha$
$\mathit{ptr}(\varphi, \alpha) \leq \mathit{ptr}(\varphi', \alpha) \quad \Leftrightarrow \quad \varphi \subseteq \varphi'$

$$\frac{A \vdash x : (\varphi, \alpha) \qquad A \vdash y : (\varphi', \alpha') \qquad \alpha' \leq \alpha}{A \vdash \mathit{welltyped}(x = y)}$$

**Fig. 1.** Assignment processing in Das's algorithm. Figure (a) above shows the points-to graph after processing $x = y$. The domains and type rule in figure (b) above provide a declarative specification of assignment processing.

(a)




(b) $\mathit{ptr}(\varphi, \alpha) \leq^i_p \mathit{ptr}(\varphi', \alpha) \quad \Leftrightarrow \quad \varphi \leq^i_p \varphi' \qquad p \in \{+, -\}$

$$\frac{A \vdash p : (\varphi, \alpha) \quad A \vdash r : (\varphi', \alpha') \qquad A \vdash f : \alpha \to \alpha' \quad \forall s \in S^* : A \vdash \mathit{welltyped}(s)}{A \vdash \mathit{welltyped}(f = \text{fun } (p) \to (r)\ S^*)}$$

$$\frac{A \vdash x : (\varphi_x, \alpha_x) \quad A \vdash y : (\varphi_y, \alpha_y) \qquad A \vdash f : \alpha \to \alpha' \quad \alpha_y \leq^i_- \alpha \quad \alpha' \leq^i_+ \alpha_x}{A \vdash \mathit{welltyped}(x = f_i(y))}$$

**Fig. 2.** Function call processing in GOLF. The graph fragments in (a) above represent points-to information after processing $x = f_i(y)$, a call to function $f$ with argument $y$ at call site $i$. For ease of exposition, we assume that functions are normalized: the statement $f = \text{fun } (p) \to (r)\ S^*$ defines a function $f$ that has a single formal parameter $p$, an out parameter $r$ that holds the return value, and a statement body $S^*$. The labeled constraints $\leq^i_-$ and $\leq^i_+$ generated at function calls are similar to instantiation constraints used in polymorphic type inference [Hen93], except that the direction of constraints with negative $(-)$ polarity is reversed to match the direction of flow of values (see [RF01]).

The effect of function calls in GOLF is shown in Figure 2. Parameter passing induces flow edges that are labeled by a call site identifier and a polarity (Figure 2 (a)). The polarity indicates the direction of flow of values, either into the called function through a formal parameter ($-$), or out of the function through a return statement ($+$). Function call processing is represented declaratively through the type rules for function definitions and function calls in Figure 2 (b). Value flow in and out of a called function generates special labeled constraints between the ids of pointed-to locations, while the contents of pointed-to locations are equated. These labeled constraints are similar to subset constraints, except that the labels are used to restrict the ways in which constraints can be composed transitively. As explained in Section 2, we wish to rule out the invalid flow of values that arises when an edge labeled $i_-$, representing the flow of values into a function at call site $i$, is followed by an edge labeled $j_+$, representing the flow of values back to a different call site $j$.

**Valid flow paths.** The set of valid flow paths is characterized precisely by the grammar shown below, and taken from [RF01]. The sequence of labels encountered along a path of flow edges forms a string. A path is a "valid path" iff its sequence of labels forms a string in the context-free language recognized by non-terminal $S$:

$$\begin{array}{lll} S & ::= & P\ N \\ P & ::= & M\ P \mid i_+\ P \mid \epsilon \end{array} \qquad \begin{array}{lll} N & ::= & M\ N \mid i_-\ N \mid \epsilon \\ M & ::= & i_-\ M\ i_+ \mid M\ M \mid \leq \mid \epsilon \end{array}$$

### 3.1 Declarative specification

The GOLF algorithm can be viewed as a set of non-standard type inference rules over a simple language of pointer related assignments. The set of rules includes the rules from Figure 2, and the rules from Das's algorithm for handling various kinds of explicit assignment statements, shown below:

$$\frac{A \vdash x : (\varphi, \alpha) \qquad A \vdash y : (\varphi', \alpha') \qquad \alpha' \leq \alpha}{A \vdash welltyped(x = y)} \qquad \frac{A \vdash x : (\varphi, \alpha) \qquad A \vdash y : \tau \qquad ptr(\tau) \leq \alpha}{A \vdash welltyped(x = \&y)}$$

$$\frac{A \vdash x : (\varphi, \alpha) \quad A \vdash y : (\varphi', ptr(\tau)) \quad \tau = (\varphi'', \alpha'') \quad \alpha'' \leq \alpha}{A \vdash welltyped(x = *y)} \qquad \frac{A \vdash x : (\varphi', ptr(\tau)) \quad A \vdash y : (\varphi, \alpha) \quad \tau = (\varphi'', \alpha'') \quad \alpha \leq \alpha''}{A \vdash welltyped(*x = y)}$$

### 3.2 Correctness

We claim that the type rules in Figure 2 and above provide a specification of a correct flow-insensitive but context-sensitive pointer analysis. This follows

from the observation that our type rules can be viewed as a restriction of the type system presented by Rehof and Fähndrich in [RF01]. Their type system, which has been shown to be correct, defines an algorithm with full subtyping and polymorphism. The GOLF type rules define an algorithm with one level of subtyping and one level of polymorphism.

There is a formal connection between constraint satisfaction in our type inference rules and valid paths in the GOLF points-to graph. The connection is provided in [RF01].

**Global storage.** The goal of GOLF is to identify all valid flow induced by function calls. In C programs, this may include flow because of uses of global variables within functions. It is possible that some flow may occur because of a global variable even though no labeled flow edge is produced in the points-to graph. Reps *et al* have suggested treating globals as extra parameters, but this may lead to a large increase in the size of the points-to graph. Instead, we identify nodes associated with globals (we call these "global storage" nodes) and add self loops on these nodes, labeled with every possible call site and polarity. This conservative approximation ensures that we cannot omit any flow of values through global variables.

A similar problem occurs with indirect accesses through pointer valued parameters. One solution would be to modify our function call rule to add self loops with the given call site label on all nodes below the nodes related by labeled flow edges. Instead, we use the conservative approximation of treating these nodes as global storage nodes.

### 3.3 Operational algorithm

Every symbol referenced in the program is associated with a unique location on demand. The program is processed one assignment at a time, including implicit assignments generated by function calls. At every assignment, locations are unified as necessary to satisfy the type equality requirements imposed by the non-standard type inference rules in Figure 2 and Section 3.1. Processing of simple subset constraints and labeled constraints is delayed by introducing flow edges between locations, as shown in Figures 1 and 2. When two locations are unified, flow edges between the locations turn into self loops. Unlabeled ($\leq$) self loops are discarded, but labeled ($\leq_p^i$) self loops are retained in order to capture all valid flow. The GOLF graph could therefore contain more edges than Das's points-to graph. In practice, the increase in edge count is very low.

Once processing of the entire program is complete, points-to sets are produced from the points-to graph. Symbol $x$ must be included in the points-to set at a dereference $*p$ iff there is a valid path from the node associated with $x$ to the node associated with $*p$. Reps *et al* have observed that the presence of such a path can be determined using CFL-Reachability queries [RHS95]. We use single-source queries, one for each symbol in the program, to populate all points-to sets.

**Global storage.** We identify and mark global storage nodes in a linear scan of the points-to graph. Instead of adding a linear number of self edges at each

global storage node, we account for the effect of the self edges implicitly: if there is a valid path from node $u$ to node $v$, where $v$ is a global storage node, and there is a valid path from node $v$ to node $w$, then there must be a valid path from node $u$ to node $w$. This is because any unmatched labeled edges in the path from $u$ to $w$ through $v$ can be matched by following an appropriate set of self edges at $v$. In other words, the effect of a global storage node is to introduce transitivity in CFL-Reachability queries. This leads to a modified reachability procedure: node $v$ is reachable from node $u$ iff it is possible to reach from $u$ to $v$ by "stepping" on global storage nodes and using valid paths to "hop" from one global storage node to the next.

### 3.4 Complexity

The algorithm has two steps: an assignment processing step, which produces a points-to graph with flow edges, and a flow propagation step. The first step has the same complexity as Steensgaard's algorithm. It uses linear space, and has almost linear running-time (in the size of the program). Every implicit assignment causes the addition of a single labeled edge. The number of implicit assignments is linear even in the presence of indirect calls, because there is a single signature for all possible target functions at a call site [Ste96].

The flow step involves a CFL-Reachability query on the graph for each symbol in the program. The worst-case cost of an all-pairs CFL-Reachability query over a graph is cubic in the number of graph nodes [RHS95]. Therefore, the complexity of GOLF is cubic in the size of the program.

### 3.5 Efficient CFL-Reachability

In this subsection we explain three insights that allow us to efficiently compute points-to sets for large programs.

**Memoization.** Our first insight is that a simple memoization, borrowed from [Das00], can allow us to amortize the cost of multiple queries by avoiding repeated work. Our experiments show that in every points-to graph, there is a single node (the "blob") that has a large number of outgoing flow edges. In every graph, the blob has an order of magnitude more outgoing edges than any other node. Now consider the set of symbols that have valid paths to the blob. For each such symbol, we would repeat a scan of the subgraph originating from the blob. Instead, we would like to perform the scan from the blob exactly once, cache the result, and share this across all symbols that reach the blob.

We pre-compute the set of nodes reachable from the blob ("frontier nodes"), and the set of symbols that reach the blob. For every symbol that does not reach the blob, we perform a forward scan to dereference nodes, as usual. For a symbol that reaches the blob, we perform a forward scan, but we stop at frontier nodes. Once we have processed all symbols, we append the symbols that reach the blob to the points-to set at every frontier node.

Consider a symbol for which we compose the scan from the blob with the scan from the symbol as described above. Because CFL-Reachability is not transitive,

we may be treating more nodes as reachable from the symbol than necessary. However, if the blob is a global storage node, we can compose without loss of precision. The only programs for which the blob is not a global storage node are extremely small, and therefore do not require memoization. On the other hand, there may be some frontier nodes at which the scan from a symbol arrives with less matching requirements for a valid path than the scan from the blob. If we stop the scan at these nodes and compose with the scan from the blob, we may fail to visit some nodes. Therefore, we identify such cases during the scan from the symbol, and continue the scan through the frontier node.

This simple memoization results in dramatic speedup. Our empirical evidence shows that there are many scans that involve the blob, for which we amortize the scan cost. All remaining scans cover very small regions of the graph.

We believe that the existence of the blob is not a coincidence. Rather, it reflects the presence of global variables that are referenced throughout a program. The blob node is an accumulator for large points-to sets. These sets are poor targets for improvement via more precise pointer analysis, because they are unlikely to shrink to very small sets, and because a precise analysis is likely to spend considerable resources tracking global variables. Points-to sets outside the reach of the blob are better targets for more precise analysis.

**Global storage.** Our second insight is that we can use the transitive behaviour of global storage nodes to make a single scan more efficient. Global storage nodes serve as points where we can use a divide and conquer strategy to form longer valid paths from shorter valid paths without enforcing matching requirements.

**Summary edges.** Our algorithm for a single CFL-Reachability query is based on a demand algorithm outlined by Horwitz *et al* in [HRS95], which improves the efficiency of queries by adding special summary edges to the graph. We have adapted their algorithm to handle nodes that are shared across functions because of unification, and to handle global storage.

## 4 Alias frequency

We are interested in estimating the impact of pointer analysis on compiler optimizations, in a manner that is independent of a particular optimizing compiler or optimization. However, previously defined measures of precision for pointer analysis that are independent of a particular optimization, such as average points-to set size and number of singleton points-to sets, provide little indication of the ability of a pointer analysis to enable optimizations by identifying memory accesses as not aliased.

Therefore, we propose "alias frequency", a new metric that estimates the precision of alias information produced by a given pointer analysis.

### 4.1 Simple alias frequency

For a given program, we define *queries* to be a set of alias queries. Each query $(e_1, e_2)$ involves a pair of memory access expressions occuring statically in the

program. The alias frequency of a given pointer analysis is the percentage of queries for which the analysis says that $e_1$ and $e_2$ may refer to the same memory location in some execution of the program:

$$\text{simple alias frequency} = \frac{\sum_{(e_1,e_2)\in queries} a(e_1,e_2)}{\sum_{(e_1,e_2)\in queries} 1} \times 100$$

$$a(e_1,e_2) = \begin{cases} 1 & \text{if } e_1, e_2 \text{ may be aliases} \\ 0 & \text{otherwise} \end{cases}$$

**Alias queries.** An extreme approach to generating alias queries would be to consider all pairs of memory accesses encountered anywhere in the program. This would result in a large number of pairs of accesses from different functions. Most of these pairs are uninteresting, because a typical optimizer will not optimize code across function boundaries. Therefore, we consider only alias queries where both memory access expressions occur in the body of the same function (there may be duplicate pairs). We believe these queries represent most intra-procedural optimizations performed in commonly used C compilers.[1]

Some expressions contain multiple dereference operators. In order to limit the number of queries, we consider only top-level memory accesses from assignment expressions, conditional expressions, and function arguments[2]. We have experimented with different criteria for selecting queries (such as including subexpressions of nested dereferences, and ignoring function arguments), and have found that our results remain consistent.

**Categorizing queries.** We categorize memory accesses based on whether they require pointer information to resolve. We define a "symbol-access" recursively: a symbol-access is a variable, a field access operation on a symbol-access, or an array index operation on a symbol-access of array type. Every remaining memory access, including a dereference, an arrow operation, or an array index operation on an object of pointer type, is a "pointer-access". Every alias query relating two symbol-accesses can be answered without pointer analysis. If the two symbol-accesses refer to the same variable, we say they may be aliases:

$$a(s_1,s_2) = \begin{cases} 1 & \text{if } var(s_1) = var(s_2) \\ 0 & \text{otherwise} \end{cases}$$

**Measuring a given pointer analysis.** Given a pointer analysis that produces a points-to set *pts(e)* for every expression $e$ (we set $pts(e) = \{var(e)\}$ for a symbol-access $e$), we can answer queries involving pointer-accesses. Two accesses may be aliases if and only if their points-to sets overlap:

$$a(e_1,e_2) = \begin{cases} 1 & \text{if } pts(e_1) \cap pts(e_2) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

[1] Notice that these queries would include aliases between globals and locals referenced in the body of the same function.

[2] Given $**p = x$, we consider $**p$ and $x$, but not $*p$.

**Measuring the best and worst possible pointer analysis.** We are especially interested in understanding the gap in alias precision between scalable pointer analyses and more precise algorithms. Therefore, we create an artificial lower bound analysis that under-estimates the alias frequency of the best possible safe pointer analysis, by treating every query involving at least one pointer-access as not aliased. The only exception is when GOLF determines that a pair of accesses refer to the same *single* stack or global symbol. The lower bound analysis treats these pairs as aliases:[3]

$$a(e_1, e_2) = \begin{cases} 1 & \text{if } \ pts(e_1) = pts(e_2) = \{v\} \\ 0 & \text{otherwise} \end{cases}$$

The lower bound analysis has the property that it is at least as precise as the best possible pointer analysis on every alias query. Therefore, if a given pointer analysis is close in alias frequency to the lower bound analysis, it must be at least as close to any more precise safe pointer analysis.

We also create an artificial upper bound analysis, by treating every query involving at least one pointer-access as aliases. The upper bound analysis indicates whether any form of pointer analysis is necessary for a given program.

Our metric over-estimates the alias frequency of the lower bound analysis, because a pair of accesses of the same variable may not be aliases if the accesses refer to different structure fields. However, we are concerned with the *difference* between a given analysis and the lower bound. Consider pairs of accesses where at least one access is a pointer-access. The lower bound analysis treats such pairs as not aliased, whereas any of our pointer analyses could potentially improve its response to these queries using field distinction. For pairs of symbol-accesses, all of the analyses, including the lower bound analysis, suffer equally. Therefore, our lack of field distinction leads us to conservatively *over-estimate* the precision gap between a given pointer analysis and the lower bound analysis.[4]

### 4.2 Weighted alias frequency

As mentioned in the introduction, we would like to estimate the potential impact on run-time of the alias queries on which a pointer analysis produces a possibly inaccurate response. Therefore, we weight the response $a(e_1, e_2)$ of any analysis to every query by the sum of the dynamic execution counts ($num(e_1) + num(e_2)$, gathered from profile data) of the accesses in the query:

$$\text{weighted alias frequency} = \frac{\sum_{(e_1,e_2)\in queries} a(e_1,e_2)\times(num(e_1)+num(e_2))}{\sum_{(e_1,e_2)\in queries} num(e_1)+num(e_2)} \times 100$$

A small difference between the weighted alias frequency of a given pointer analysis and the lower bound analysis means that a more precise pointer analysis is unlikely to enable additional optimizations that improve run-time significantly.

[3] We create dummy symbols, one at each dynamic allocation site, to represent heap storage. The lower bound analysis does not treat accesses of the same heap symbol as aliases, whereas our pointer analyses do.

[4] The same argument applies to accesses of static arrays.

| Program | LOC | AST nodes | Time (s) |
|---|---|---|---|
| compress | 1,904 | 2,234 | 0.05 |
| li | 7,602 | 23,379 | 0.50 |
| m88ksim | 19,412 | 65,967 | 0.88 |
| ijpeg | 31,215 | 79,486 | 1.13 |
| go | 29,919 | 109,134 | 0.98 |
| perl | 26,871 | 116,490 | 1.53 |
| vortex | 67,211 | 200,107 | 4.09 |
| gcc | 205,406 | 604,100 | 7.17 |
| Word97 | 2,150,793 | 5,961,129 | 133.66 |

**Table 1.** Benchmark data. For each program, the table above shows the lines of code, the AST node count, and the running-time (in seconds) of GOLF.

## 5 Experiments

We have produced a modular implementation of GOLF using the AST Toolkit, which is itself an extension of the Microsoft Visual C++ compiler. Our implementation handles all of the features of C. Details may be found in [Das00]. We implemented GOLF by modifying the rules for parameter passing and return statements in our implementation of Das's algorithm, and by adding a CFL-Reachability engine. Our implementation of Das's algorithm has been tested extensively. Apart from all of the usual testing, we verified the correctness of our implementation of GOLF in two ways. First, we performed reachability queries forward and backward, with and without memoization, and verified that we get the same results in every case. Second, we tested our implementation of CFL-Reachability by treating labeled edges as unlabeled and verifying that we obtain the same points-to sets as with Das's algorithm.

**Benchmark programs.** Table 1 shows our benchmark programs, consisting of the integer benchmarks from SPEC95, and a version of Microsoft Word. For each benchmark, we list the total lines of source code (including comments and blank lines), as well as the number of AST nodes (a more accurate measure of program size), and the analysis time (in seconds) for GOLF, averaged over 5 runs. Analysis time includes time to analyze each compilation unit (excluding parse time), time to write out object files, time to read in all of the object files, perform unifications, and compute points-to sets exhaustively at all static dereference points in the program using CFL-Reachability. All of our experiments were conducted on a Dell 610 desktop PC running Windows 2000, with 512MB RAM and a single 800Mhz Intel Pentium III processor.

### 5.1 Alias precision of pointer analysis

Table 2 shows the simple and weighted alias frequencies of various pointer analyses. We obtained execution counts for computation of weighted alias frequency by instrumenting the benchmarks and running them on their SPEC reference

| Program | Simple Alias frequency | | | | | | Weighted Alias frequency | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lower | GOLF | Das00 | Ste96 | Upper | Diff | Lower | GOLF | Das00 | Ste96 | Upper | Diff |
| compress | 13.8 | 14.02 | 14.02 | 14.13 | 32.38 | 0.22 | 9.93 | 9.93 | 9.93 | 9.93 | 28.82 | 0.0 |
| li | 10.17 | 18.84 | 18.84 | 19.53 | 42.27 | 8.67 | 13.10 | 22.98 | 22.98 | 22.99 | 62.12 | 9.88 |
| m88ksim | 14.97 | 17.0 | 17.0 | 20.44 | 40.5 | 2.03 | 11.53 | 13.77 | 13.77 | 18.67 | 37.77 | 2.24 |
| ijpeg | 5.93 | 17.9 | 17.9 | 19.14 | 61.49 | 11.97 | 5.55 | 16.31 | 16.31 | 16.31 | 57.42 | 10.76 |
| go | 7.85 | 7.87 | 7.87 | 7.87 | 8.35 | 0.02 | 9.5 | 9.73 | 9.73 | 9.73 | 15.53 | 0.23 |
| perl | 9.54 | 14.45 | 14.45 | 14.53 | 45.17 | 4.91 | 3.45 | 12.56 | 12.56 | 12.56 | 53.87 | 9.11 |
| vortex | 6.12 | 10.81 | 10.81 | 15.71 | 42.69 | 4.69 | 3.7 | 7.18 | 7.18 | 14.51 | 50.20 | 3.48 |
| gcc | 5.49 | 11.98 | 11.98 | 14.64 | 50.36 | 6.49 | 4.62 | 9.36 | 9.36 | 10.72 | 51.66 | 4.74 |
| Word97 | 6.63 | 14.45 | 15.07 | 20.37 | 44.21 | 8.44 | - | - | - | - | - | - |
| Average | 8.94 | 14.15 | 14.22 | 16.26 | 40.82 | 5.27 | 7.67 | 12.73 | 12.73 | 14.43 | 44.67 | 5.06 |

**Table 2.** Precision of various pointer analyses. For each benchmark program, the table above shows the simple alias frequency of the lower bound analysis (Lower), GOLF, Das's algorithm (Das00), Steensgaard's algorithm (Ste96), and the upper bound analysis (Upper), and the difference in simple alias frequency between Das00 and Lower (Diff). The same data is also shown for weighted alias frequency. We were not able to obtain dynamic execution counts for Word97.

inputs.[5] The data shows that all of the scalable pointer analyses are surprisingly close to the lower bound analysis. Das's algorithm does as well as the lower bound analysis on all but 5.2% of the alias queries, on our benchmark programs.

To better understand the loss in precision from scalable pointer analysis, we manually examined a fraction of the queries on which Das's algorithm differs from the lower bound analysis. We found that in almost every case, either the lower bound analysis is unsound, or we could have used straightforward field distinction to resolve the query as not aliased. Therefore, we believe that the gap in alias frequency between scalable pointer analyses and the best possible pointer analysis is in fact *much less* than 5%.

The data in Table 2 also shows that the difference in weighted alias frequency between Das's algorithm and the lower bound analysis is very similar to the difference in simple alias frequency, for every benchmark. We therefore claim that the queries on which Das's algorithm is inaccurate are not likely to provide significant additional opportunity for optimization.

On all of our benchmarks, the differences in weighted alias frequencies between various analyses are very similar to the differences in simple alias frequency between the same analyses. Hence, we argue that simple alias frequency is a useful indicator of precision for implementors of pointer analysis who do not have access to either profile data or optimizing compilers that can consume alias information produced by their analyses.

[5] The reference input for gcc consists of 56 C source files. We ran gcc on the five largest source files and averaged the execution counts.

**Das00 vs GOLF.** Table 2 shows that the alias frequency of Das's algorithm is not improved by the addition of context-sensitivity for any benchmark other than Word97. This data shows that in practice, scalable pointer analyses do not sacrifice optimization opportunity because of a lack of context-sensitivity.

**Ste96.** The data also shows that Steensgaard's algorithm is surprisingly close to the lower bound analysis, given the relatively poor precision of the algorithm in terms of points-to set size (see Table 3). We believe that this is largely because the pollution of points-to sets that occurs in Steensgaard's algorithm leads to accumulation of variables across functions, but this pollution does not result in conservative alias relationships between pointer variables from the same function. Also, smaller points-to sets do not imply lower alias frequency, if the smaller sets contain the same subset of common symbols. Finally, points-to set sizes are often artificially inflated by the inclusion of symbols that are out of scope. Table 2 and Table 3 clearly show that traditional measures of precision for pointer analysis do not reflect the ability of the analysis to produce good alias information.

**Das00 vs Andersen.** Previous work [Das00] has shown that Das's algorithm and Andersen's algorithm [And94] produce almost identical points-to sets. Therefore, their alias frequencies can be expected to be almost identical as well.

**Limitations.** First, although we measure alias frequency, we are really evaluating pointer analysis. It may be possible to significantly improve the alias frequency of any analysis, including the lower bound analysis, by adding a structure field analysis and/or an array index analysis. Second, our results apply only to C programs; whether they apply to programs written in C++ or Java is an open question. Third, aggressive inter-procedural optimizers may be able to utilize alias information inter-procedurally [Ruf00]. These opportunities are not reflected in our selection of alias queries.

One potential concern with our results may be that scalable pointer analyses appear close to the lower bound analysis because we have swamped the query set with pairs of symbol accesses. We find that on the average, 30% of the queries require some form of pointer information. This large percentage indicates that there is a need for at least some form of pointer analysis in compilers.

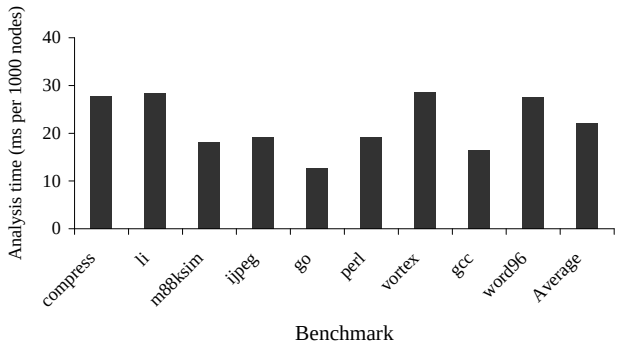
As might be expected, we generate a large number of alias queries (10 million queries for Word97). Each query may require reachability on the points-to graph. By using the amortization technique described in Section 3.5, we are able to answer alias queries extremely efficiently. We can answer all queries for Word97 in less than seven minutes.

There may be regions of a program where a more accurate analysis may eliminate aliases. For instance, consider a linked list traversal using previous and current pointers. Our results show that a useful approach may be to first run a scalable pointer analysis, and then apply a more precise shape analysis locally, on a few functions. These functions can be identified using alias frequency.

### 5.2 Performance and precision of GOLF

**Performance.** In the figure below, we chart the running times for GOLF from Table 1. We use the ratio of running-time to program size. The chart shows that

this ratio is fairly steady as program size grows, indicating that the analysis scales linearly with program size. GOLF requires roughly twice as much time and as much memory as Das's algorithm. We do not present detailed data on space consumption, which is very low. GOLF requires 20MB for Word97:




**Precision vs other scalable pointer analyses.** Table 3 shows the precision of GOLF measured using traditional metrics. The table shows the average size of points-to sets at dereference points and the number of singleton points-to sets for each benchmark. Following previous work, the size of the points-to set at a dereference point is the number of program symbols, including dummy symbols produced at dynamic allocation sites, in the points-to set of the dereference expression. All three analyses were run with the same settings, using the same implementation. We omit data for points-to sets at indirect call sites, although GOLF can also be used to improve points-to sets for function pointers as well.

Points-to sets with single elements represent opportunites for replacing conditional updates with strong updates. Smaller points-to sets may lead to greater efficiency in subsequent analyses [SH97], as well as less run-time overhead in systems that instrument code [MCE00]. The data shows that GOLF produces more singleton sets than Das's algorithm for several benchmarks. On the whole, our results appear to be consistent with the results of Foster *et al* ([FFA00]), who found little improvement in precision from the addition of polymorphism to a pointer analysis with subtyping. Our benchmark programs are much larger than in [FFA00], and we do see greater improvement on larger programs.

**Precision vs full polymorphic subtyping.** GOLF approximates a full polymorphic subtyping algorithm by restricting both subtyping and polymorphism to one level in the type structure. Das has already shown that the one level restriction of subtyping does not cause loss in precision [Das00]. The data for the FRD00 and OneLev columns in Table 3 shows that the one level restriction of polymorphism does not cause precision loss either.

It is therefore likely that GOLF extracts most of the precision of an analysis with full polymorphism and subtyping. However, it is still possible that the combination of full polymorphism and full subtyping may eliminate more spurious flow of values than the combination of limited polymorphism and limited sub-

| Program | Average thru-deref size | | | | | Singleton sets | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Ste96 | Das00 | Golf | FRD00 | OneLev | Ste96 | Das00 | Golf | FRD00 | OneLev |
| compress | 2.1 | 1.22 | 1.22 | 2.9 | 2.9 | 36 | 47 | 47 | 30 | 30 |
| li | 287.7 | 185.62 | 185.62 | 189.63 | 194.80 | 15 | 39 | 39 | 15 | 15 |
| m88ksim | 86.3 | 3.29 | 3.27 | 14.13 | 15.16 | 116 | 638 | 641 | 256 | 251 |
| ijpeg | 17.0 | 13.14 | 11.78 | 13.01 | 14.30 | 1,671 | 3,287 | 3,287 | 1,802 | 1,777 |
| go | 45.2 | 14.79 | 14.79 | 16.06 | 16.06 | 28 | 28 | 28 | 23 | 23 |
| perl | 36.1 | 22.24 | 21.90 | 23.89 | 23.91 | 240 | 1,023 | 1,155 | 307 | 306 |
| vortex | 1,064.5 | 59.86 | 59.30 | 57.42 | 65.70 | 808 | 4,855 | 4,855 | 4,764 | 4,764 |
| gcc | 245.8 | 7.96 | 7.71 | 90.62 | 97.17 | 1,323 | 6,830 | 6,896 | 2,637 | 2,598 |
| Word97 | 27,176.3 | 11,219.5 | 7,756.6 | - | - | 11,577 | 41,904 | 43,142 | - | - |

**Table 3.** Precision of various pointer analyses. For each benchmark program, the table above shows the average size of points-to sets at static dereference points for Ste96, Das00, GOLF, a polymorphic version of Steensgaard's algorithm (FRD00), and a one level restriction of FRD00 (OneLev). The table also shows the number of dereference points with singleton points-to sets found using each of these algorithms. FRD00 and OneLev cannot be compared directly with the other analyses, because they are based on Rehof's implementation of a polymorphic version of Steensgaard's algorithm [FRD00]. We were not able to use this implementation to analyze Word97.

typing used in GOLF. We were unable to perform a direct comparison, because it has not been possible to scale polymorphic subtyping to large programs.

**Context-sensitive clients.** In order to populate points-to sets, we accumulate all flow of pointer values into a function from its callers. In the example below, the points-to set of `l` is the same using Das's algorithm or GOLF:

```
void Read(Obj o1, Obj o2) { LockWrap(&o1.lock); LockWrap(&o2.lock); }
void LockWrap(Lock *l) { AcquireLock(l); }
```

However, the labeled edges in GOLF can be used to produce distinct summaries of function behaviour at different call sites. These summaries can be leveraged by a client of GOLF, as long as the client is context-sensitive. For instance, a context-sensitive analysis that tracks lockable objects can use the summaries of `LockWrap` produced by GOLF to conclude that `o1` *must* be locked by the first call to `LockWrap`. Das's algorithm can only say that `o1` *may* be locked by *either* call. We believe that this is the real value of GOLF.

## 6 Related work

**GOLF.** As we mentioned in the introduction, our work on GOLF follows a long line of research on context-sensitive pointer analysis. The most precise algorithms are control-flow-sensitive and context-sensitive [LR92,WL95,EGH94,CRL99]. It is not clear whether any of these algorithms will scale beyond 50,000 lines of

code. Previous algorithms for control-flow-insensitive context-sensitive pointer analysis include [LH99,CH00,FRD00]. The first two algorithms follow every edge in the call graph, whether the call graph is pre-computed or constructed on the fly. This may limit their applicability to large programs, which have very large (quadratic sized) call graphs due to indirect calls. On the other hand, [FRD00] appears to scale well, but it does not provide any degree of subtyping, which is important for larger programs.

GOLF is a context-sensitive algorithm with subtyping that scales to large programs. It is an extension of Das's algorithm [Das00]. We apply his one level flow idea to restrict polymorphism without losing precision, and we borrow his caching technique to speed up our flow computation. GOLF can also be viewed as a restriction of Rehof and Fähndrich's general polymorphic subtyping framework in [RF01]. With some modifications to account for unification, globals, and pointers, GOLF can be viewed as a variant of Reps, Horwitz and Sagiv's framework in [RHS95]. GOLF is a scalable instantiation of these two frameworks.

Liang and Harrold have described a mechanism for extracting some context-sensitivity from context-insensitive pointer analyses [LH00]. We believe that their approach could be used to add context-sensitivity to Das's algorithm. It is not clear how GOLF would compare with the resulting analysis.

Ruf [Ruf95] and Foster *et al* [FFA00] have reported empirical investigations of the added precision provided by context-sensitive pointer analysis. Both argue that there is little gain in precision from context-sensitivity. Our results are consistent with theirs, and extend their conclusions to much larger programs. However, we believe that the real value of GOLF is as a context-sensitive value flow analysis that produces polymorphic summaries of function behaviour.

**Impact of flow-insensitive pointer analysis.** The issue we have addressed in this paper is the usefulness of control-flow-insensitive pointer analyses in compiler optimizations. Although conventional wisdom says that the lack of flow-sensitivity and structure-field distinction can severely limit the usefulness of scalable pointer analyses, there is no empirical evidence to support this belief. In fact, several studies have produced results that contradict this idea [DMM98,CH00,HP98]. Cheng and Hwu have shown that a context-sensitive pointer analysis with subtyping can enable many optimizations in a compiler [CH00]. Their result inspired us to develop a scalable context-sensitive pointer analysis with subtyping. Hind and Pioli have shown that flow-sensitivity has little impact on the precision of pointer analysis [HP98]. Diwan *et al* have shown that for a particular Java optimization, a flow-insensitive pointer analysis provides all of the precision that can be exploited by an optimizer [DMM98]. Our results are consistent with all of these studies.

We know of no previous work that uses alias frequency to estimate the impact of pointer analysis on compiler optimizations. Diwan *et al* have studied the effect of pointer analysis on a particular Java optimization at several levels, including static points-to information, optimization opportunities enabled, and run-time improvement [DMM98]. Ideally, we would like to repeat their study for every

conceivable optimization and every pointer analysis. We propose weighted alias frequency as a practical replacement for such a large set of experimental studies.

One avenue for further improvement in precision that is suggested by our results is to run a scalable analysis globally, and apply more precise analysis locally. Rountev *et al* have proposed this idea in [RRL99]. Our results provide evidence that supports their approach. They use Steensgaard's algorithm as the scalable global analysis. We believe that using GOLF as the global analysis would lead to greater precision. Also, our alias frequency measure can be used in their framework, to identify target functions for more precise analysis.

## 7 Conclusions

In this paper, we have provided experimental evidence to support the claim that scalable pointer analyses provide precise alias information for C programs. We believe this is a strong argument for the routine use of scalable pointer analysis in optimizing compilers. We have also developed a framework for measuring the impact of pointer analysis on compiler optimizations in a manner that is independent of a particular optimization or optimizing compiler. Finally, we have presented GOLF, the first algorithm that can trace the flow of values in very large C programs, while providing a degree of subtyping and context-sensitivity. We believe that the most useful method for analysis of large programs may be to use a scalable global analysis in conjunction with an expensive local analysis.

## Acknowledgements

We would like to thank Tom Reps and Rakesh Ghiya for helpful discussions, and Jim Larus and the anonymous referees for suggestions on the paper.

## References

And94. L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

CH00. B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

CRL99. R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *26th ACM SIGPLAN Symposium on Principles of Programming Languages*, 1999.

Das00. M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

DMM98. A. Diwan, K. S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation*, 1998.

EGH94. M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, 1994.

FFA00. J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the 7'th International Static Analysis Symposium*, 2000.

FRD00. M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.

Hen93. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

HP98. M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Fifth International Static Analysis Symposium, Pisa, Italy*, number 1503 in LNCS, pages 57–81. Springer-Verlag, 1998.

HRS95. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering, SIGSOFT Software Engineering Notes 20, 4*, 1995.

LH99. D. Liang and M. Harrold. Efficient points-to analysis for whole program analysis. In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999.

LH00. D. Liang and M. Harrold. Light-weight context recovery for efficient and accurate program analyses. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

LR92. W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 1992.

MCE00. M. Mock, C. Chambers, and S. J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *In 33rd Annual International Symposium on Microarchitecture, December 2000, (Micro-33), Monterrey, California*, December 2000.

RF01. J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-Reachability. In *Proceedings of the 28th Annual ACM Symposium on Principles of Programming Languages*, January 2001.

RHS95. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1995.

RRL99. A. Rountev, B. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999.

Ruf95. E. Ruf. Context-sensitive alias analysis reconsidered. *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, 1995.

Ruf00. E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

SH97. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *LNCS 1302, 4th International Symposium on Static Analysis*. Springer-Verlag, 1997.

Ste96. B. Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages*, 1996.

Tip95. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

WL95. R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN 95 Conference on Programming Language Design and Implementation*, 1995.