Fast and Accurate Flow-Insensitive Points-To Analysis*

Marc Shapiro and Susan Horwitz
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
Electronic mail: {mds, horwitz}@cs.wisc.edu

Abstract

In order to analyze a program that involves pointers, it is necessary to have (safe) information about what each pointer points to. There are many different approaches to computing points-to information. This paper addresses techniques for flow- and context-insensitive interprocedural analysis of stack-based storage.

The paper makes two contributions to work in this area:

- The first contribution is a set of experiments that explore the trade-offs between techniques previously defined by Lars Andersen and Bjarne Steensgaard. The former has a cubic worst-case running time, while the latter is essentially linear. However, the former may be much more precise than the latter. We have found that in practice, Andersen's algorithm is consistently more precise than Steensgaard's. For small programs, there is very little difference in the times required by the two approaches; however, for larger programs, Andersen's algorithm can be much slower than Steensgaard's.

- The second contribution is the definition of two new algorithms. The first algorithm can be "tuned" so that its worst-case time and space requirements, as well as its accuracy range from those of Steensgaard to those of Andersen. We have experimented with several versions of this algorithm; one version provided a significant increase in accuracy over Steensgaard's algorithm, while keeping the running time within a factor of two.

  The second algorithm uses the first as a subroutine. Its worst-case time and space requirements are a factor of $\log N$ (where N is the number of variables in the program) worse than those of Steensgaard's algorithm. In practice, it appears to run about ten times slower than Steensgaard's algorithm; however it is significantly more accurate than Steensgaard's algorithm, and significantly faster than Andersen's algorithm on large programs.

1 Introduction

In order to analyze a program that involves pointers, it is necessary to have (safe) information about what each pointer points to. In general, the more precise the points-to information, the more precise the analysis. For example, consider the following code segment:

```
[1] x = 0;
[2] *p = 1;
[3] write(x);
```

Dataflow-analysis problems like constant propagation, reaching definitions, and live variables all rely on knowing which variables are defined at each statement. In the example above, we need to know where variable p might point in order to determine the effect of executing statement 2. For example, if precise information has been computed, and it is known that p points to x, then it can be determined that x has the constant value 1 at statement 3. Similarly, if it is known that p does not point to x, then it can be determined that x has the constant value 0 at statement 3. By contrast, if no points-to information is available, then it must be assumed that p might or might not point to x, and the value of x at statement 3 cannot be determined.

There has been a great deal of work on techniques for computing points-to information. Some have addressed tracking heap-allocated storage (e.g.: [JM81] [HPR89] [Hen90] [Deu90] [CWZ90] [Deu94] [GH96] [SRW96]), while others have concentrated on stack-based storage. The latter can be further classified as flow-sensitive or flow-insensitive, and as context-sensitive or context-insensitive. Flow-sensitive analysis (e.g.: [LR92] [CBC93] [EGH94]) takes into account the order in which statements are executed, while flow-insensitive analysis (e.g.: [Wei80] [MCCH94] [And94] [Ste96b] [ZRL96]), assumes that statements can be executed in any order. Similarly, context-sensitive analysis takes into account the fact that a function must return to the site of the most recent call, while context-insensitive analysis propagates information from a call site, through the

* This work was supported in part by the National Science Foundation under grant CCR-8958530, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

called function, and back to *all* call sites. (An interesting pair of papers that address this issue are [WL95] and [Ruf95].) Flow and context sensitivity generally provide more precise results, but can also be more costly in terms of time and/or space.

This paper addresses techniques for flow- and context-insensitive analysis of stack-based storage. Even within this somewhat limited context there is a range of approaches that trade precision for speed. At one end is the algorithm defined by Andersen [And94],[2] which may require $O(n^3)$ time, where n is the size of the program. At the other end is the algorithm defined by Steensgaard[Ste96b], which runs in almost linear time, but which may produce less precise results than Andersen's analysis.

The first contribution of this paper is a set of experiments that explore the actual trade-offs between Andersen's and Steensgaard's approaches. We have implemented both algorithms and used them to analyze 61 C programs, ranging in size from 300 to 24,300 lines. The results of this study are reported in Section 3.2. We have found that Andersen's algorithm is consistently more precise than Steensgaard's: the average size of a points-to set computed by Steensgaard's algorithm was more than twice the size computed by Andersen's algorithm in 38 of the 61 cases. For small programs (up to about 3,000 lines), there is very little difference in the times required by the two approaches; however, for larger programs, Andersen's algorithm can be much slower than Steensgaard's (as much as 150 times as slow in one case).

One can think of both Andersen's and Steensgaard's approaches as building a graph (sometimes called a *storage shape graph*[CWZ90] or an *alias graph*[MCCH94]) that represents the points-to relationships among the program's variables. The important difference between the two approaches has to do with the out-degree of the graph; Andersen allows each node to have an arbitrary number of out-edges, while Steensgaard allows only one out-edge. Because of this restriction, a node in Steensgaard's graph may represent more than one variable, while in Andersen's graph, each node represents exactly one variable. The coarser granularity of Steensgaard's graphs leads to both the fast runtime of his algorithm and its loss of precision.

Example: Figure 1 shows the graphs that would be built by the two approaches for an example program, and the points-to information that each graph represents. Note that Steensgaard's approach erroneously determines that b might point to e, and that d might point to c. □

The second contribution of this paper is the definition of two new algorithms for flow-insensitive pointer analysis. The first algorithm takes two parameters that make it tunable: setting the parameters to one extreme causes it to produce the same results (in the same worst-case time) as Steensgaard's algorithm; setting the parameters to the other extreme causes it to produce the same results (in the same worst-case time) as Andersen's algorithm; intermediate values produce intermediate results. Experimental results (reported in Section 3.3) show that with one version of this algorithm we can achieve a significant increase in accuracy over Steensgaard's algorithm, while keeping the running time within a factor of two.

Our second algorithm, which uses the first as a subroutine, is also parameterized: by the out-degree of the nodes of the points-to graph. In the worst case, for out-degree k, this algorithm uses $O(k^2\ N)$ space and $O(k^2\, n\, \alpha(k^2\, n,\, k^2\, n)\, \log_k N)$ time, where α is the (very slowly growing) inverse Ackermann's function that arises in the context of fast union/find data structures[Tar83], N is the number of variables, and n is the size of the program.[3] Thus, for a fixed value of k, its asymptotic complexity is slightly worse (by a factor of $\log N$) than that of Steensgaard's algorithm. In theory, its accuracy may be no better than that of Steensgaard's algorithm; however, experimental evidence (reported in Section 3.4) indicates that in practice this algorithm is more accurate than Steensgaard's (and more accurate than our first algorithm with the same k) – it can be almost as precise as Andersen's. Although on small programs it is slower than Andersen's algorithm, on the 6 examples for which Andersen's algorithm took the most time, our algorithm was considerably faster. Therefore, at least when analyzing large programs, it may be the algorithm of choice for fast and accurate points-to analysis.

2 New Algorithms for Flow-Insensitive Pointer Analysis

In this section we describe our new algorithms for flow-insensitive pointer analysis. The first algorithm can be viewed as an extension of Steensgaard's algorithm in which a variable's points-to set is partitioned into multiple categories. While this algorithm is interesting in its own right, it also forms the basis for the second algorithm, which calls it as a subroutine. It is the second algorithm that seems to hold the most promise for fast and precise pointer analysis.

Due to space constraints, we are not able to give a complete review of Steensgaard's algorithm, or a complete definition of our own first algorithm. Instead of including those details, we have taken a high-level approach: we concentrate on explaining how our algorithm avoids some of the imprecision of Steensgaard's algorithm, and we use examples to demonstrate how our algorithm works. For those familiar with Steensgaard's explanation of his algorithm in terms of a non-standard type-inference system, we have included an appendix that defines our approach as an extension to that system.

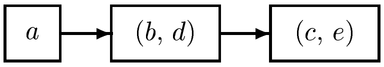
[2] Andersen also defines an algorithm that uses function in-lining to achieve some context sensitivity. Whenever we refer to Andersen's algorithm in this paper we mean his context-insensitive version.

[3] These are the space and time requirements for building a data structure that represents the points-to sets. Additional space, $O(N^2)$ in the worst case, and additional time, $O(N^2\ \log_k N)$ in the worst case, is needed to extract the actual sets. Alternatively, the data structure can be used to answer queries of the form "might x point to y?", with no additional space requirements in $O(\log_k N)$ time for each query. Similarly, the bounds given for Steensgaard's algorithm, $O(n)$ space and $O(n\, \alpha(n, n))$ time, only account for building the data structure. Using his data structure, the actual points-to sets (of size $O(N^2)$ in the worst case) can be extracted in worst-case time $O(N^2)$, while a query "might x point to y?", can be answered in constant time.

Program

$a = \&b$
$b = \&c$
$a = \&d$
$d = \&e$

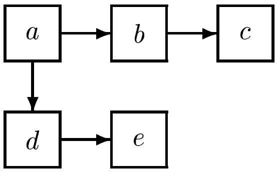
Steensgaard's Graph



Steensgaard's Points-To Set

$a \to b$
$a \to d$
$b \to c$
$b \to e$
$d \to c$
$d \to e$

Andersen's Graph



Andersen's Points-To Set

$a \to b$
$a \to d$
$b \to c$
$d \to e$

Figure 1: Example of points-to analysis using Steensgaard's and Andersen's algorithms

2.1 Algorithm 1: Multiple Categories, One Run

2.1.1 Motivation

As discussed in the Introduction, one source of imprecision in Steensgaard's analysis is that nodes in the points-to graph are limited to out-degree at most one. In particular, whenever two variables v_1 and v_2 are both in the points-to set of some variable x, the nodes that represent v_1 and v_2 are merged (using fast union-find data structures[AHU74]). This can lead to two kinds of imprecision:

1. All points-to sets that contain v_1 will also contain v_2 (and vice versa).
2. Everything in the points-to set of v_1 will also be in the points-to set of v_2 (and vice versa).

Example: Figure 2 (which gives an example program, the graph that would be built by Steensgaard's algorithm, and the points-to information represented by the graph) illustrates the first kind of imprecision: the nodes that represent v_1 and v_2 are merged because they are both in the points-to set of x; this leads to having v_2 as well as v_1 in the points-to set of y.

Figure 1 illustrates the second kind of imprecision: the nodes that represent b and d are merged because they are both in the points-to set of a; this in turn forces the nodes that represent c and e to be merged, which leads to having e in the points-to set of b, and c in the points-to set of d. □

2.1.2 Algorithm description

Our first pointer-analysis algorithm permits a more precise analysis by allowing each node of the graph to have out-degree k (where k is one of the algorithm's parameters). Each program variable is assigned to one of k *categories*; if variables v_1 and v_2 are both in the points-to set of some variable x, the nodes that represent v_1 and v_2 are merged only if v_1 and v_2 are in the same category. The assignment of variables to categories is the algorithm's second parameter. Different partitionings of variables into categories will, in general, lead to different results. Assigning all variables to the same category yields Steensgaard's algorithm, while assigning each variable to a unique category yields an algorithm that is equivalent to Andersen's.

Example: Figure 3 gives a program, four different assignments of variables to categories, the graphs that would be built by our algorithm for each such assignment, and the "extra" (imprecise) points-to information represented by the graphs. □

Figure 3 only shows the results of the analysis; it does not provide insight into the details of the algorithm. However, the details are not really new; the basic approach is the same as Steensgaard's:

- We use fast union-find data structures to represent the nodes of the graph.
- We use the "cjoin" (conditional-join) operation and "pending" sets to permit assignment statements to be treated asymmetrically, without having to iterate over the program statements.

2.1.3 Algorithm complexity

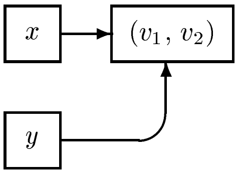
Like Steensgaard's, our algorithm processes each program statement once. The most expensive statements are those that involve indirection; for example: $x = *y$, or $*y = x$.[4] Handling $*y$ requires processing all of the nodes "two steps" away from y (i.e., all of the nodes that represent the points-to sets of the variables that

[4] We assume that multiple levels of indirection have been transformed to single levels. For example, $x = **y$ would be transformed to $t1 = *y$; $x = *t1$ (and a different temporary would be used for each such transformation). These transformations will, of course, increase the number of statements in the program. However, since each new statement is of constant size, and the number of new statements is proportional to the number of indirections that appear in the program, the increase in overall program size is linear.

Program

$x = \&v_1$
$x = \&v_2$
$y = \&v_1$

Steensgaard's Graph



Corresponding Points-To Set

$x \to v_1$
$x \to v_2$
$y \to v_1$
$y \to v_2$ /* imprecise! */

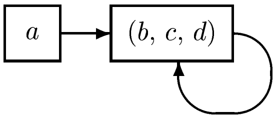
Figure 2: Example illustrating one kind of imprecision in Steensgaard's analysis.

Program

$a = \&b$
$a = \&c$
$a = \&d$
$c = \&d$

1 Category (equivalent to Steensgaard)

graph

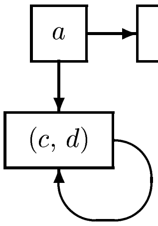


extra points-to pairs

$b \to b$ $\quad$ $c \to c$
$b \to c$ $\quad$ $d \to b$
$b \to d$ $\quad$ $d \to c$
$c \to b$ $\quad$ $d \to d$

2 Categories: $\{a, b\}$ and $\{c, d\}$

graph

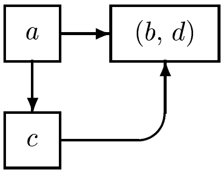


extra points-to pairs

$c \to c$
$d \to c$
$d \to d$

2 Categories: $\{a, c\}$ and $\{b, d\}$

graph

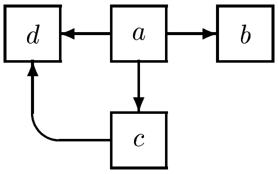


extra points-to pairs

$c \to b$

3 Categories: $\{a, b\}$, $\{c\}$, and $\{d\}$

graph



extra points-to pairs

$--$ *none* $--$

Figure 3: Examples of analysis using multiple categories.

are themselves in the points-to sets of y). Given k categories, there are k^2 such nodes. Thus, at most $k^2\,n$ nodes are processed, where n is the size of the program. Processing a node involves some constant-time work; in addition, in the course of processing the $k^2\,n$ nodes, at most $k^2\,n$ *Find* and N *Union* operations are performed, where N is the number of variables (and thus the number of nodes in the graph before any *Union* operations are performed). The average cost of $k^2\,n$ *Find* operations is $O(k^2\,n\,\alpha(k^2\,n,\,k^2\,n))$, where α is the inverse Ackermann's function. This subsumes the cost of the N *Union* operations (which each take constant time), as well as the constant-time costs of processing the nodes. The total cost of the algorithm is therefore $O(k^2\,n\,\alpha(k^2\,n,\,k^2\,n))$.

2.1.4 Motivation for the second algorithm

As illustrated by the examples in Figure 3, the precision of this analysis depends both on the number of categories and on the particular assignment of variables to categories. There is no obvious way to assign variables to categories so as to maximize the precision of the analysis. However, we can observe that different assignments of variables to categories may lead to the computation of different points-to sets. If a variable y is in the points-to set of variable x only as the result of some but not all such assignments, then we can be sure that x *cannot* in fact point to y. This observation leads to our second pointer-analysis algorithm, described in the following section.

2.2 Algorithm 2: Multiple Categories, Multiple Runs

2.2.1 Algorithm description

The idea behind our second algorithm is to use multiple runs of the first algorithm, each with a different assignment of variables to categories. The final points-to set for each variable x is computed by intersecting the points-to sets computed for x by each run. There are many interesting ways one might choose the number of categories and the assignment of variables to categories for each run, as well as the number of runs. The particular strategy we have chosen has the property that for every pair of variables $(x,\,y)$, there is at least one run in which x and y are assigned to different categories; the number of runs is as small as it can be given this property.

The number of categories, k (which must be in the range 2 to N, where N is the number of variables), is specified as a parameter to the algorithm. The number of runs is then set to $R = \lceil \log_k N \rceil$. Each variable is assigned a unique number in the range 0 to N-1, and that number is written in base k (padded with leading zeros if necessary to make the number R digits long). On the n^{th} run, the n^{th} digit (counting from right to left) is used to determine the variable's category. Since every variable has a unique number, every pair of variables must differ on at least one digit, and thus there must be some run on which they are assigned to different categories.

Example: The program shown in Figure 3 has four variables (a, b, c, and d). If k is 2, then R will also be 2, and the variable numbers in base k will be:

a	00
b	01
c	10
d	11

Thus, for the first run, the categories will be: $\{a,\,c\}$ and $\{b,\,d\}$, and for the second run, they will be: $\{a,\,b\}$ and $\{c,\,d\}$. The graphs built by the first algorithm using those categories have been shown in Figure 3; they represent the following points-to sets:

Run 1	Run 2
$a \to b$	$a \to b$
$a \to c$	$a \to c$
$a \to d$	$a \to d$
$c \to b$	$c \to c$
$c \to d$	$c \to d$
$d \to d$	$d \to c$

The intersection of these sets yields:

$a \to b$
$a \to c$
$a \to d$
$c \to d$

which is the same as the result of running the algorithm with 4 categories (i.e., is as precise as Andersen's algorithm). □

2.2.2 Algorithm complexity

As argued in the previous section, each individual run requires $O(k^2\,n\,\alpha(k^2\,n,\,k^2\,n))$ time in the worst case. Therefore, the time required for all of the runs is $O(k^2 n\alpha(k^2 n, k^2 n)\log_k N)$, which is slower than Steensgaard's algorithm by a factor of $k^2\,\log_k N$. Actually extracting the points-to set for each variable x requires intersecting the $\log_k N$ sets computed for x by the individual runs. For a single variable x, this requires worst-case time $O(N\,\log_k N)$, because the sizes of x's points-to sets might be $O(N)$ (e.g., when x points to all N variables). Therefore, the worst-case time to extract all points-to sets is $O(N^2\,\log_k N)$. It is worth noting that the points-to sets computed by each run are guaranteed to be no larger than the points-to sets computed by Steensgaard's algorithm (and in fact we would expect them to be smaller, since we are using $k > 1$ categories). Thus, the time required by our algorithm to extract points-to sets is no more than a factor of $\log_k N$ larger than the time required by Steensgaard's.

Although in theory the results of our "multiple-category multiple-runs" analysis can be the same as that of Steensgaard's algorithm, we have found that in practice they are significantly better. Experimental results are given in the next section.

3 Experimental Results

3.1 Background to the Experiments

We have implemented the two algorithms described in Section 2, as well as Andersen's algorithm, in order to compare the running times and accuracies of Steensgaard's algorithm, Andersen's algorithm, and the new algorithms proposed in this paper. (Using our first algorithm with a separate category for each variable produces the same results as Andersen's algorithm; however, the approach is different, and although the two approaches have the same asymptotic time requirements, in practice, our algorithm is slower than Andersen's. Therefore, it would not have been fair to use an N-category version of our algorithm to compare the running times of Andersen's algorithm with those of the other algorithms. We do, however, use the 1-category version of our algorithm as Steensgaard's algorithm. Because our implementation is more general, the 1-category version may actually be slightly slower than a direct implementation of Steensgaard's algorithm.)

The three algorithms were implemented in C and used with a front end that analyzes a C program and generates the corresponding control-flow graph (reported running times do not include the time for the front-end since this was the same for all of the algorithms). The algorithms handle function calls via pointers to functions as part of the analysis (i.e., they track the set of functions that might be pointed to by the pointer variable and essentially include additional calls as new elements are added to the set). Each "malloc" site is treated as returning the address of a distinct variable. Calls to library routines other than malloc are treated like calls to functions with empty bodies[5]. Structures, unions, and arrays are all treated as single objects; i.e., an assignment to any field (any array element) is treated as an assignment to the object. (Recent work by Steensgaard[Ste96a] has addressed extensions to distinguish among the components of structured objects.)

Tests were carried out on a Sparc 20/71 with 256 MB of RAM. The study used 61 C programs including Gnu Unix utilities, Spec benchmarks, and programs used for benchmarking by Landi[LRZ93] and by Austin[ABS94]. The tables in Figures 4 and 5 give the number of lines of preprocessed source code (with blank lines removed) for each test program. The programs are divided into the two tables according to the running times of Steensgaard's and Andersen's algorithms: "small" programs, those for which both algorithms took less than one second are included in Figure 4; the rest of the programs are included in Figure 5. Experimental results for all programs are reported below in our comparison of Andersen's and Steensgaard's algorithms. However, in the interest of brevity, for the remainder of this section results are reported only for the 25 programs for which at least one of those algorithms required at least one second.

[5]While this is not safe in general, it suffices to compare the different pointer analyses.

3.2 Comparison of Andersen's and Steensgaard's Algorithms

The purpose of our first set of experiments was to compare the accuracies and running times of Andersen's and Steensgaard's analyses. The results of these experiments are shown in Figures 4, 5, 6, and 7. Figures 4 and 5 list the test programs sorted by size; for each program and for each of the two analyses, the figure includes the total size of the points-to sets (i.e., the sum of the sizes of all variables' points-to sets), the average size of a variable's points-to set, and the time required for the analysis.[6] Figures 6 and 7 are graphs comparing the relative precision of Steensgaard's and Andersen's analyses. Each point is the ratio (Steensgaard / Andersen) of the total size of the points-to sets computed for one test program. The data is sorted along the x-axis by test program size (points on the x-axis are labeled with the first 4 characters of the test-program names – see Figures 4 and 5). There is one data point (for the program *patch*) for which the ratio is actually about 70-to-1. Letting the y-axis run from 0 to 70 would make the graph unreadable; instead, the "outlier" data point is represented by placing it at the top of the graph with an up-arrow to indicate that it is off-scale. Similar data points can be found in Figures 9, 11, and 13.

We found that for small programs (up to about 3,000 lines), both Andersen's and Steensgaard's analyses are very fast (usually less than one second). For larger programs, Andersen's algorithm occasionally takes a very long time, while Steensgaard's is more consistent. For example, there are six test programs for which Andersen's algorithm takes more than ten times as long as Steengaard's; for two of them (*espresso* and *li*), it takes more than one hundred times as long.

In many cases, Andersen's algorithm yields more precise results, often by a large margin. For example, for 37 out of our 61 programs, the total (and average) sizes of the points-to sets computed by Andersen are less than half of those computed by Steensgaard. Considering only the "large" programs, the sizes of the points-to sets computed by Andersen are less than half of those computed by Steensgaard in 21 out of 25 cases; they are less than one-fifth of those computed by Steensgaard in 11 out of 25 cases.

3.3 Multiple Categories, One Run

The purpose of our next set of experiments was to assess the accuracy and time requirements of the algorithm described in Section 2.1, relative to Steensgaard's and Andersen's algorithms. We tried four different versions: using 2, 3, 5, and 9 categories. In each case, variables were partitioned into categories by numbering the variables from 0 to $N-1$ and then using the number modulo the number of categories. The results of these experiments are shown in Figures 8 and 9.

Figure 8 shows the precision of the four versions relative to those of Steensgaard and Andersen. In this

[6]The average sizes are computed using the variables identified by Steensgaard's algorithm as having non-empty points-to sets. In general, this is a superset of the variables so identified by Andersen's algorithm. For one test program, *ul*, this disparity causes the average size reported for Andersen's algorithm to be less than one.

Test Program	Lines	Total Size		Average Size		Time	
		And.	Steens.	And.	Steens.	And.	Steens.
diff.diffh	303	38	167	2.38	10.44	0.07	0.08
genetic	336	24	82	1.6	5.47	0.07	0.12
anagram	344	32	50	1.19	1.85	0.07	0.08
allroots	427	11	14	1.57	2.0	0.03	0.05
ul	451	8	9	0.89	1.0	0.10	0.18
ks	574	87	92	1.74	1.84	0.10	0.15
compress	657	14	1	1.0	1.0	0.13	0.20
stanford	665	34	4	1.26	1.52	0.17	0.15
clinpack	695	27	4	1.5	2.44	0.15	0.15
travel	725	22	39	2.75	4.88	0.05	0.10
lex315	747	36	36	4.5	4.5	0.17	0.15
sim	748	145	282	1.04	2.03	0.48	0.23
mway	806	33	43	1.14	1.48	0.17	0.27
pokerd	1099	112	729	2.0	13.02	0.27	0.28
ft	1185	140	197	1.92	2.70	0.15	0.18
ansitape	1222	109	389	2.95	10.51	0.22	0.28
loader	1255	212	1544	1.91	13.91	0.18	0.25
gcc.main	1285	386	2330	4.02	24.27	0.38	0.35
voronoi	1394	129	13	1.04	1.05	0.17	0.28
ratfor	1531	523	6006	4.89	56.13	0.50	0.42
livc	1674	112	156	37.33	52.0	0.63	0.83
struct.beauty	1701	653	5286	7.02	56.84	0.48	0.43
diff.diff	1761	127	144	1.59	1.8	0.60	0.60
xmodem	1809	70	82	2.59	3.04	0.32	0.40
compiler	1908	94	106	3.62	4.08	0.35	0.40
learn.learn	1954	7	259	1.60	5.51	0.37	0.50
gnugo	1963	37	42	1.19	1.35	0.33	0.60
ML-parse	2019	94	102	1.81	1.96	0.12	0.12
dixie	2439	156	577	1.97	7.30	0.38	0.43
eqntott	2470	313	617	1.87	3.69	0.82	0.70
twig	2555	1428	3141	7.80	17.16	0.93	0.80
arc	2574	204	228	1.25	14.0	0.63	0.77
cdecl	2577	280	2030	4.06	29.42	0.42	0.53
patch	2746	292	20701	1.86	131.85	0.58	0.83
assembler	2994	281	294	1.32	13.84	0.52	0.68
unzip	3261	108	316	1.61	4.72	0.33	0.43

Figure 4: Information about the "small" test programs: number of lines of preprocessed code, plus total sizes of points-to sets, average sizes of points-to sets, and running times for Andersen's and Steensgaard's algorithms.

graph, the value 0 represents the total size of the points-to sets computed by Andersen's analysis, and the value 1 represents the total size of the points-to sets computed by Steensgaard's analysis. Values in between are scaled linearly. Thus, for example, the value 0.5 means that the total size of the points-to sets computed by our algorithm was halfway between that of Andersen and that of Steensgaard. Figure 9 shows the ratios of the running times of Andersen's algorithm and of our four versions, to that of Steensgaard's algorithm.

As expected, increasing the number of categories increases both the accuracy and the running time of our algorithm. It is worth noting that using 3 categories rarely takes more than twice as long as using one category (i.e., Steensgaard's algorithm), and the precision can be significantly better: for 5 of the 25 cases, the sizes of the points-to sets computed by the 3-category algorithm are at least 25% smaller than those computed by Steensgaard's algorithm, and for 17 cases the sizes are at least 5% smaller; however, the precision is not as good as Andersen's.

3.4 Multiple Categories, Several Runs

Next, we ran several sets of experiments to evaluate the efficacy of using more than one run of the algorithm from Section 2.1. We compared four versions: each used 3 categories, and the number of runs ranged from 1 to 4. Variables were partitioned into categories by numbering them in base 3 and using the n^{th} digit on the n^{th} run, as described in Section 2.2 (except, of course, that for these experiments, not all digits were used, since the number of runs was less than $\log_3 N$).

Results are shown in Figures 10 and 11. As for the previous experiments, we show the relative precision of the results compared to Steensgaard and Andersen, as well as the ratios of the running times.

The precision of the results usually improves signifi-

Test Program	Lines	Total Size		Average Size		Time	
		And.	Steens.	And.	Steens.	And.	Steens.
triangle	1968	862	4716	4.01	21.93	2.9	0.8
football	2075	120	207	1.82	3.14	1.1	1.7
lex	2645	628	1928	3.02	9.2	1.1	1.0
gcc.cpp	4061	5090	16386	13.47	43.35	4.9	1.3
simulator	4239	604	1299	2.50	5.37	1.1	1.0
struct.structure	4457	619	4485	2.11	15.31	3.1	1.5
gzip	4584	1000	8507	2.96	25.17	1.7	1.1
agrep	4906	600	868	3.77	5.46	1.0	1.3
ML-typecheck	4997	5679	8870	30.86	48.21	4.0	0.5
ptx	5001	1295	6638	3.94	20.18	2.5	1.6
li	6054	163614	437744	171.14	457.89	738.5	4.7
bc	6745	7241	32586	18.57	83.55	5.5	1.6
ispell.freq	6830	449	3290	2.25	16.45	0.9	1.2
bison	7271	78	9375	1.72	20.51	3.1	3.2
sc	7378	190	13333	3.23	22.56	3.1	2.6
grep	7433	3771	13506	8.16	29.23	5.8	2.5
ispell.ispell	7700	476	3921	2.14	17.66	3.8	2.0
sed	8022	17997	40345	23.25	52.13	18.1	2.4
find	8824	16510	44699	31.33	84.82	40.0	3.4
flex	9488	3573	15298	8.91	38.15	353.9	3.0
less	12152	308	27666	7.11	63.75	1.9	1.5
make	15564	62528	346541	74.70	414.03	250.8	6.1
tar	18585	11980	36964	17.41	53.7	23.2	3.6
espresso	22050	210847	276218	109.53	143.4	1373.6	10.2
screen	24300	118643	724673	106.89	652.8	514.5	10.1

Figure 5: Information about the "large" test programs: number of lines of preprocessed code, plus total sizes of points-to sets, average sizes of points-to sets, and running times for Andersen's and Steensgaard's algorithms.

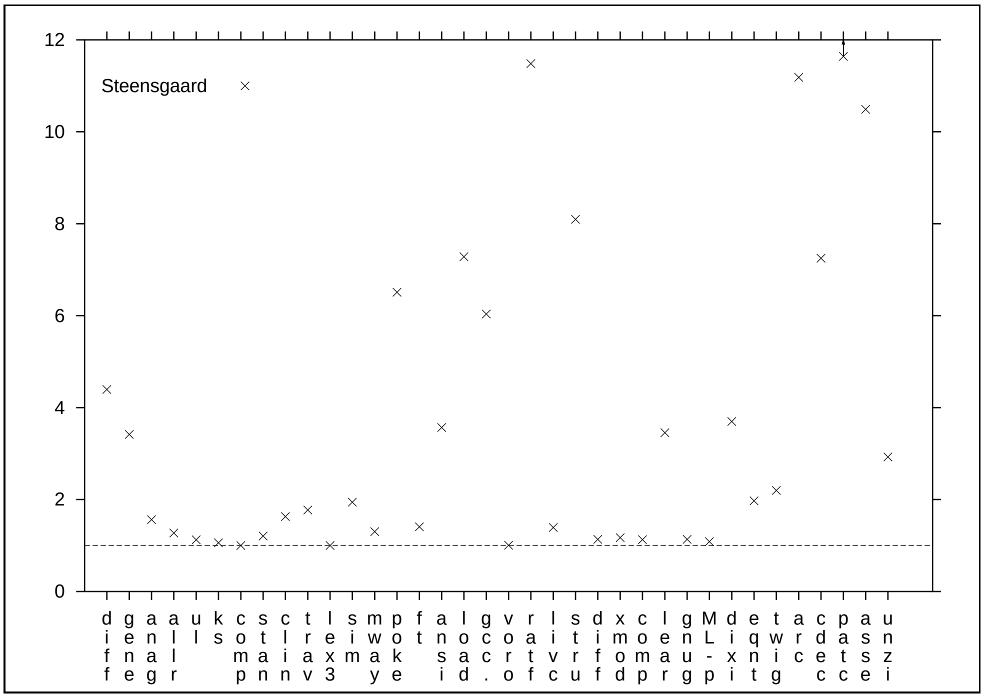


Figure 6: Ratios of sizes of points-to sets computed using Steensgaard's algorithm to those computed using Andersen's algorithm ("small" programs).

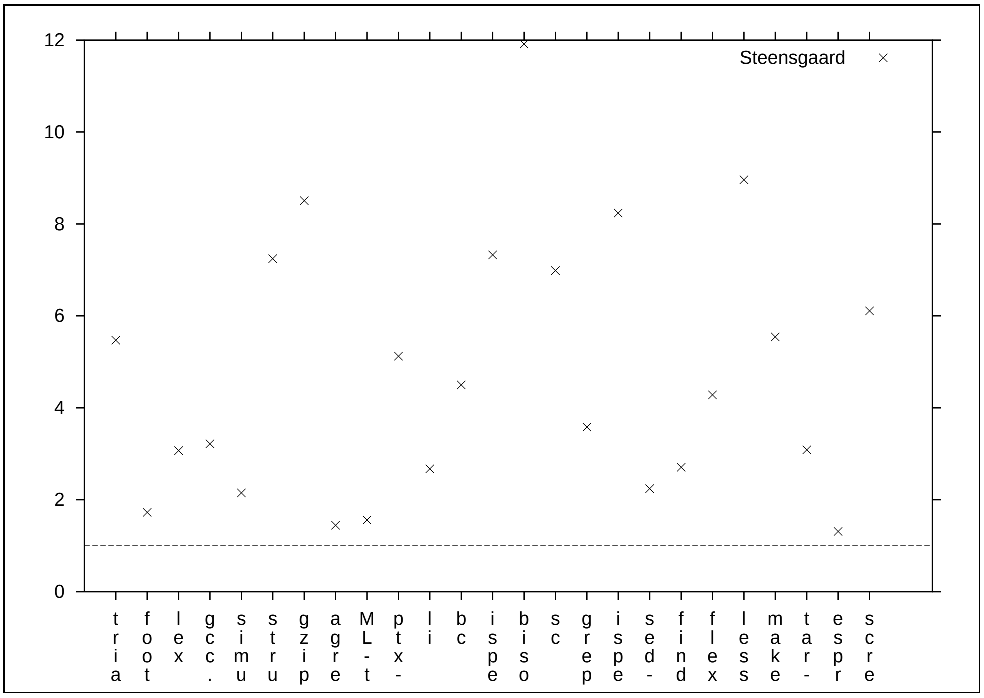


Figure 7: Ratios of sizes of points-to sets computed using Steensgaard's algorithm to those computed using Andersen's algorithm ("large" programs).

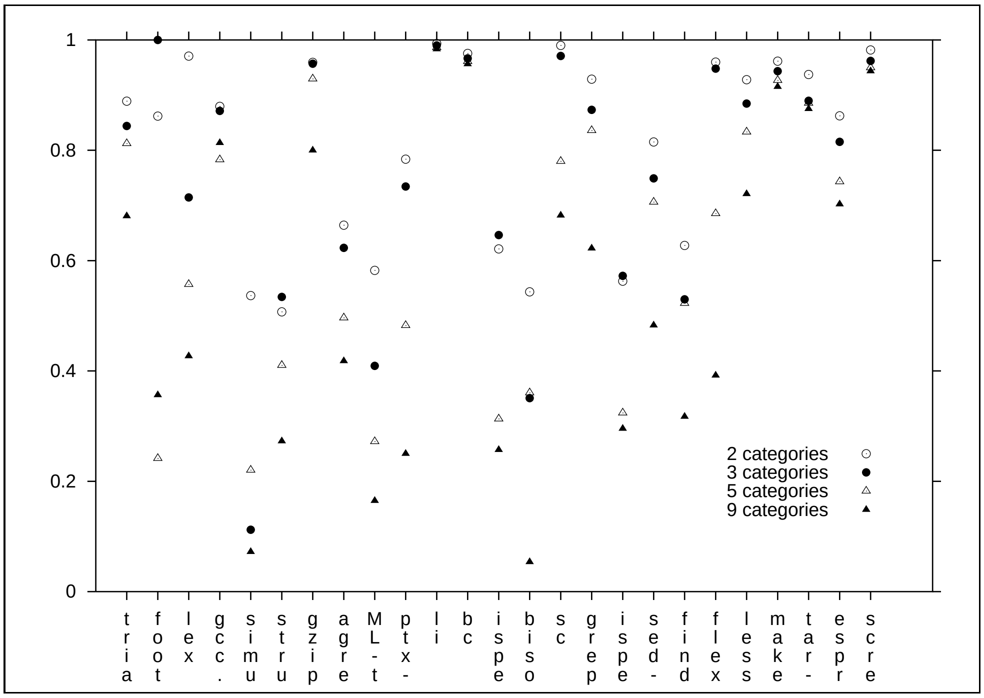


Figure 8: Relative precision of points-to sets computed using multiple categories. 1=Steensgaard 0=Andersen; intermediate values are scaled linearly.

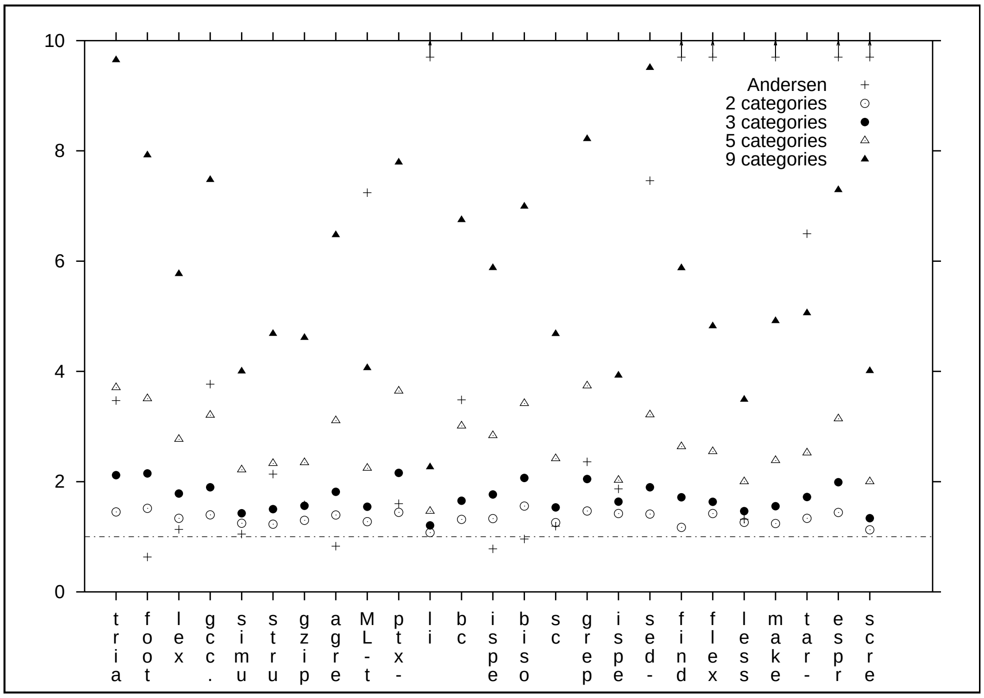


Figure 9: Ratios of the times required by Andersen's algorithm and by our multiple-category algorithms to compute points-to sets, to the times required by Steensgaard's algorithm.

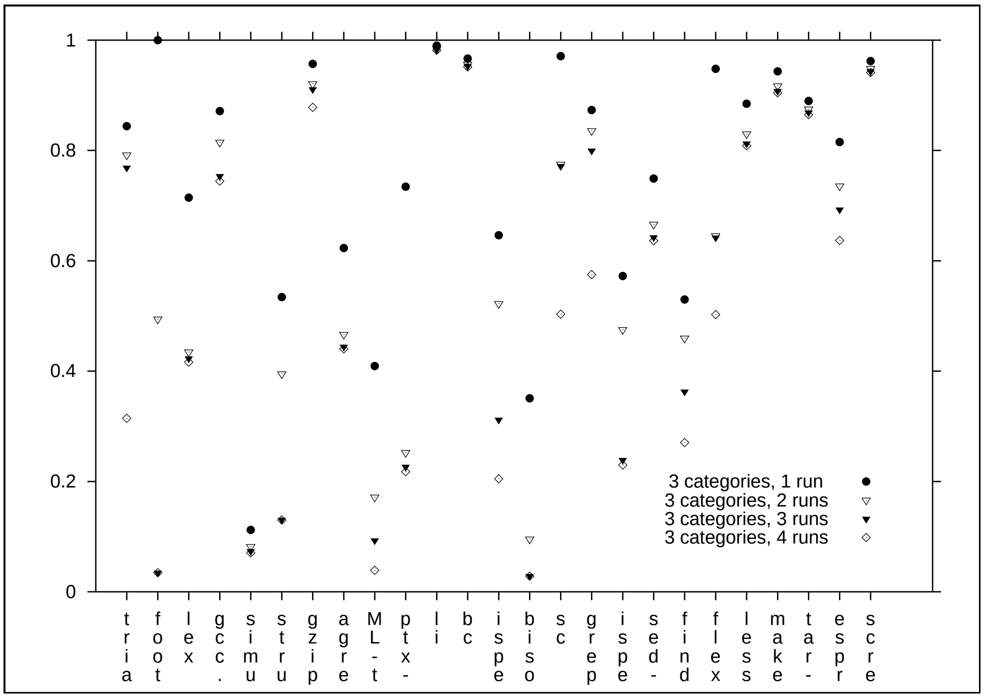


Figure 10: Relative precision of points-to sets computed using 3 categories, varying the number of runs. 1=Steensgaard 0=Andersen; intermediate values are scaled linearly.

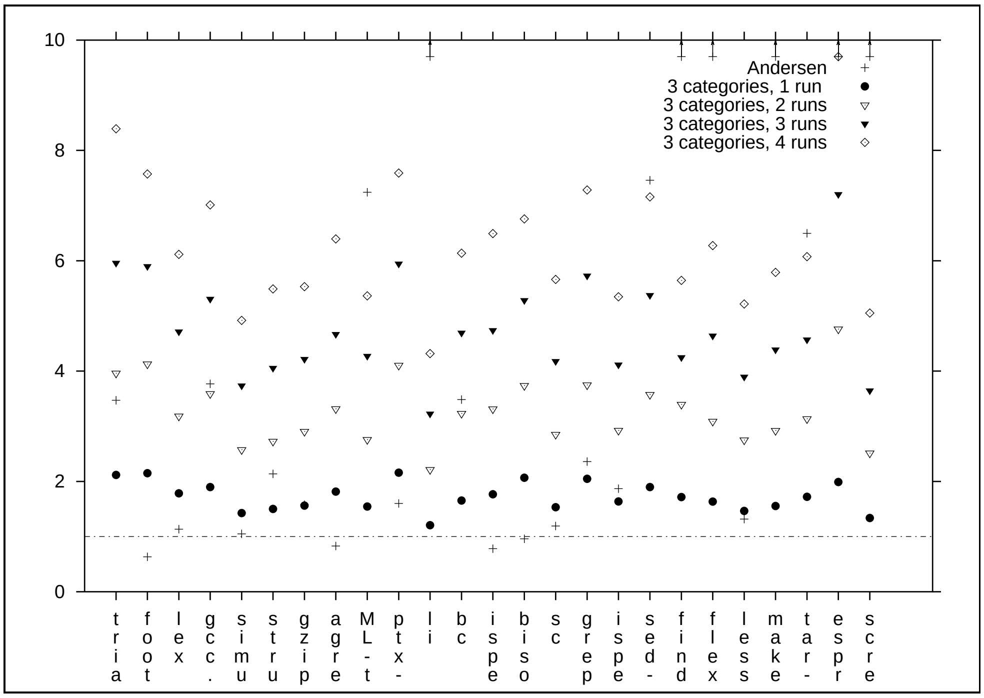


Figure 11: Ratios of the times required by Andersen's algorithm and by different versions of our 3-category algorithm (varying the number of runs) to compute points-to sets, to the times required by Steensgaard's algorithm.

cantly each time an extra run is added: Averaging over the 25 tests, the sizes of the points-to sets computed using three categories and two runs is 12.1% smaller than the sizes computed using 3 categories and 1 run; using 3 categories and 3 runs is 8.7% smaller than 3 categories and 2 runs, and using 3 categories and 4 runs is 6.3% smaller than 3 categories and 3 runs.

Our current implementation starts each run of the algorithm from scratch; thus, doubling the number of runs essentially doubles the running time. An area for future work is to see how information from previous runs could be used to speed up subsequent ones; for example, a significant amount of extra work can sometimes be avoided if it is known that a variable's points-to set is empty.

3.5 Multiple Categories, $\log N$ Runs

Our final set of experiments compares the algorithm described in Section 2.2 with those of Andersen and Steensgaard. We tried three versions of our algorithm: using 2, 3, and 5 categories. Results are shown in Figures 12 and 13.

For these test programs, using three categories tends to take less overall time than other numbers of categories. However, increasing the number of categories only takes slightly longer, and tends to give more precise results. In many cases, the precision is much closer to that of Andersen's algorithm than Steensgaard's: Averaging over the 25 tests, the sizes of the points-to sets computed using three categories and $\log N$ runs is 2.67 times the sizes computed by Andersen's algorithm, while the sizes computed by Steensgaard's algorithm are 4.75 times the size of Andersen's.

The running time of the current implementation of our algorithm, in which each run starts from scratch, is not very impressive for small programs (it is, of course, always slower than Steensgaard's algorithm, and is often slower than Andersen's). However, it is worth noting that even this implementation is significantly faster than Andersen's on the six test programs for which Andersen's algorithm required the most time, without an unreasonable loss of precision. This can be seen in Figure 14.

Thus, this approach appears to hold a great deal of promise, and may prove to be the algorithm of choice for fast and accurate points-to analysis of large programs.

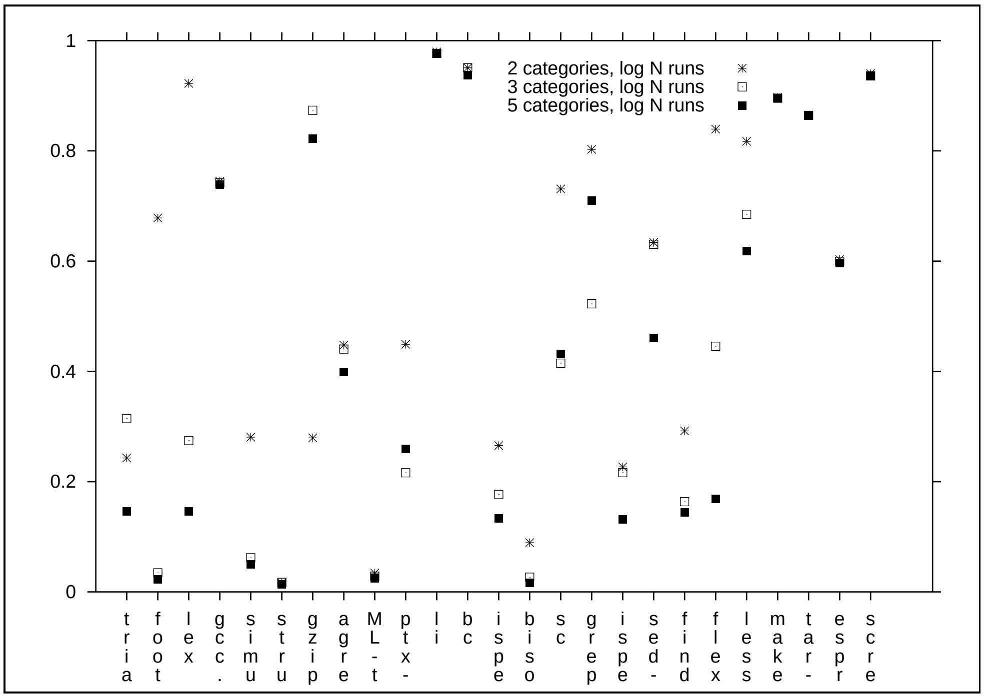


Figure 12: Relative precision of points-to sets computed using multiple categories, $\log N$ runs. 1=Steensgaard 0=Andersen; intermediate values are scaled linearly.

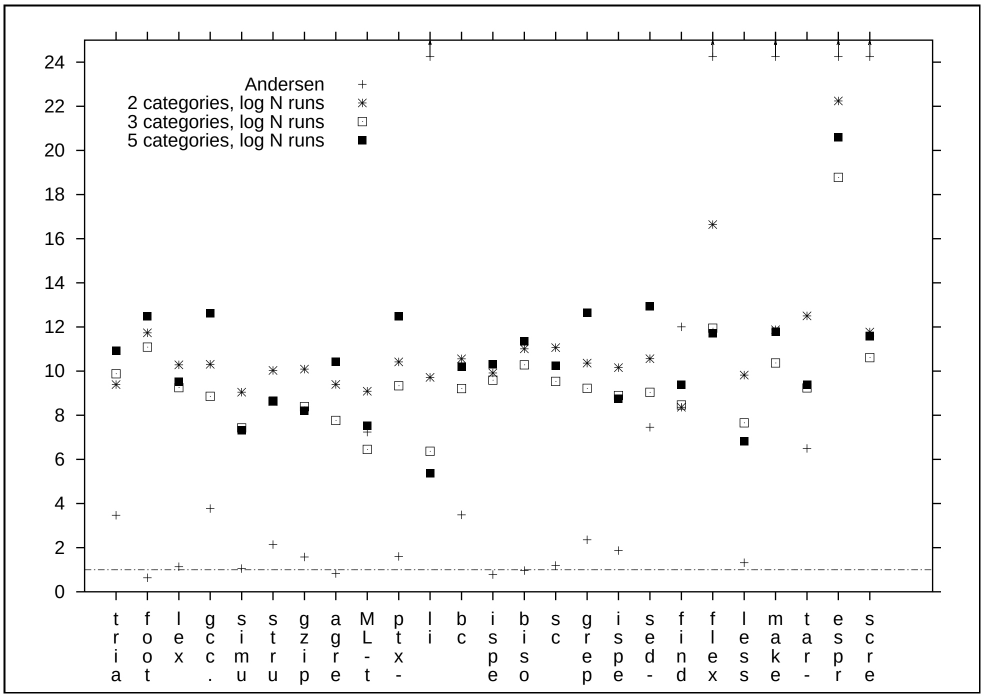


Figure 13: Ratios of the times required by Andersen's algorithm and by our multiple-categories, $\log N$ runs algorithm to compute points-to sets, to the times required by Steensgaard's algorithm.

Test Program	Total Points-To Andersen	Total Points-To 3 Cat, $\log N$ Runs	Size Ratio (3 Cat/ And)	Time And.	Time 3 Cat, $\log N$ Runs	Time Ratio (3 Cat/ And)
li	163614	431415	2.64	738.5	29.8	.04
find	16510	21121	1.28	41.0	28.9	.72
flex	3573	8796	2.46	353.9	35.4	.10
make	62528	316873	5.07	250.8	63.2	.24
espresso	210847	250028	1.19	1373.6	191.5	.14
screen	118643	686037	5.78	514.5	107.5	.21

Figure 14: Precision and run times of Andersen's algorithm, and our algorithm with 3 categories and $\log N$ runs, for those benchmarks on which Andersen's algorithm performed poorly.

A APPENDIX: Type-Inference Rules

In [Ste96b], Steensgaard explains his pointer-analysis algorithm via a non-standard type-inference system, in which types represent sets of locations (so the type inferred for a variable defines its points-to set).

The types are defined by the following grammar:

$$\begin{array}{lll} \alpha & ::= & \tau \times \lambda \\ \tau & ::= & \bot \mid ref(\alpha) \\ \lambda & ::= & \bot \mid lam(\alpha_1, \ldots, \alpha_n)(\alpha_{n+1}, \ldots, \alpha_{n+m}) \end{array}$$

The lambda types have to do with pointers to functions. Our algorithm extends the way pointers to functions are handled in the same way that it extends the way pointers to non-functions are handled; thus, we have not specifically discussed that aspect of the algorithm.

Steensgaard gives a set of typing rules, one for each kind of statement. The statements of the program to be analyzed induce typing constraints according to those rules, and the results of the points-to analysis can be defined as the minimal type environment consistent with those constraints.

Our algorithm can be viewed in terms of an extension to Steensgaard's type system in which the production given above for alpha is replaced by the following:

$$\begin{array}{lll} \alpha & ::= & \beta \times \gamma \\ \beta & ::= & \tau_1 \times \tau_2 \times \cdots \times \tau_k \\ \gamma & ::= & \lambda_1 \times \lambda_2 \times \cdots \times \lambda_{k'} \end{array}$$

where k is the number of categories for variables, and k' is the number of categories for functions.

The typing rules are extended in a straightforward way. Here are two typical examples (we follow Steensgaard in using an underscore as a "don't care" value):

$$\frac{\begin{array}{c} A \vdash x : \ ref((\tau_1 \times \tau_2 \times \cdots \times \tau_n \times \cdots \times \tau_k) \times \_) \\ A \vdash y : \ \tau_n \\ category(y) = n \end{array}}{welltyped(x = \&y)}$$

$$\frac{\begin{array}{c} A \vdash x : \ ref(\alpha) \\ A \vdash y : ref((ref(\alpha_1) \times ref(\alpha_2) \times \cdots \times ref(\alpha_k)) \times \_) \\ \forall j \in [1 \ldots k] : \alpha_j \lhd \alpha \end{array}}{welltyped(x = *y)}$$

References

[ABS94] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 290–301, June 1994.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 296–310, 1990.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 230–241, 1994.

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

[GH96] R. Ghiya and L.J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on Principles of Programming Languages.* ACM, New York, January 1996.

[Hen90] L. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, Jan 1990.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 28–40, 1989.

[JM81] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[LR92] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 235–248, June 1992.

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

[MCCH94] M.Burke, P. Carini, J.D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Galernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250, Ithaca, NY, August 1994. Springer-Verlag.

[Ruf95] E. Ruf. Context-sensitive alias analysis reconsidered. In *SIGPLAN Conference on Programming Languages Design and Implementation*, pages 13–22, June 1995.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *ACM Symposium on Principles of Programming Languages.* ACM, New York, January 1996.

[Ste96a] B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, April 1996.

[Ste96b] B. Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[Tar83] R. Tarjan. Data structures and network flow algorithms. volume CMBS44 of *Regional Conference Series in Applied Mathematics.* SIAM, 1983.

[Wei80] W.E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *ACM Symposium on Principles of Programming Languages*, pages 83–94, 1980.

[WL95] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[ZRL96] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer-induced aliasing analysis. Technical report, Rutgers University LCSR-TR-259, 1996.