

# Flow Insensitive Points-to Sets

Paul Anderson  
GrammaTech, Inc.  
317 N. Aurora St.  
Ithaca, NY 14850  
paul@grammatech.com

David Binkley<sup>†</sup>  
GrammaTech, Inc.  
317 N. Aurora St.  
Ithaca, NY 14850  
binkley@cs.loyola.edu

Genevieve Rosay  
GrammaTech, Inc.  
317 N. Aurora St.  
Ithaca, NY 14850  
rosay@execpc.com

Tim Teitelbaum  
GrammaTech, Inc.  
317 N. Aurora St.  
Ithaca, NY 14850  
tt@grammatech.com

## Abstract

*Pointer analysis is an important part of source code analysis. Many programs that manipulate source code take points-to sets as part of their input. Points-to related data collected from 27 mid-sized C programs (ranging in size from 1168 to 53,131 lines of code) is presented. The data shows the relative sizes and the complexities of computing points-to sets. Such data is useful in improving algorithms for the computation of points-to sets as well as algorithms that make use of this information in other operations. Several uses of the data are discussed.*

## 1. Introduction

Pointer analysis is becoming an increasingly important part of source code analysis. Many programs that manipulate source code require information about pointer variables. Understanding the use of pointers will be of increasing importance in the design of future tools for source code manipulation.

This paper presents data collected from an assortment of mid-sized C programs (ranging in size from 1168 to 53,131 lines of code). The data illustrates the relative complexities of computing points-to sets. It is also useful in developing future source code analysis algorithms.

Pointer analysis can be performed in a flow *sensitive* or a flow *insensitive* manner. Flow-sensitive analysis [9, 5, 6] considers the order in which statements are executed. For example, in the following code, flow-sensitive analysis correctly determines the `q` does not point to `b`.

```
p = &a  
q = p  
p = &b
```

In contrast, flow-insensitive analysis [16, 3, 12, 1] assumes that statements can be executed in any order. Thus, in the analysis of the above code fragment `q` may point to `a` and `b`. For interprocedural analyses, context-sensitivity can also be considered. A 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 called function, and back to all call sites [4, 14, 10]. This paper considers flows and context insensitive analysis.

Finally, structures and casts are particularly difficult to handle in the points-to analysis of C programs. Especially when fields are treated as separate entities and casts are used with overlapping C structures that share a common prefix [2]. A conservative approach “collapses” structure fields into a single “location.” Not surprising this leads to some imprecision. For example, given the code sequence

```
struct { int *x; int *y } s;  
int a, b;  
s.x = &a;  
s.y = &b;
```

The collapsed representation makes it appear that `s`, `s.x`, and `s.y` may all point to `a` and `b`.

---

Copyright © 2001 by GrammaTech Inc. All rights reserved.

<sup>†</sup>On sabbatical leave from Loyola College in Maryland.

Recent algorithms for pointer analysis have considered casting and structures more carefully. The implementation used to generate the data presented in Section 3 is based on the technique of Yong et. al. [15]. The data presented in Section 3 examines the importance of the collapse by considering some examples both with and without collapsing structure fields. One interesting outcome is that for some examples the expansion of fields actually improves the performance of the points-to computation.

## 2. Background

This section presents some background on points-to analysis and the algorithm used to collect the data presented in the next section. In general, flow insensitive points-to analysis is an NP-Hard problem [8]; consequently, existing practical algorithms are all approximations. The precision of these algorithms ranges from the  $O(n\alpha(n, n))$  (there  $\alpha$  is the inverse of Ackermann's function) algorithm of Steensgaard [12] to the  $O(n^3)$  algorithm of Anderson [1]. Shapiro and Horwitz [11] have presented a spectrum of point-to analysis algorithms that range in cost and precision from Steensgaard's to Anderson's. Figure 1, which is based on the figure from Shapiro and Horwitz, highlights the key differences between the two approaches.

The data presented in Section 3 was produced by an implementation of Anderson's algorithm. This algorithm reads in normalized statements that represent the pointer manipulations in the program. A graph is constructed from these statements and then a closure of this graph is computed. The final points-to sets are then extracted from the graph.

In more detail, the graph consists of nodes that represent memory locations and edges that represent relations between the nodes. Nodes are created for variables, addresses of variables, and dereferences of variables. There are four kinds of edges (named A, G, R, and W) that are defined as follows:

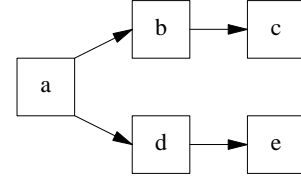
- (1)  $p \xrightarrow{A} q =_{df} q$  is in the points-to set of  $p$ .
- (2)  $p \xrightarrow{G} q =_{df} \text{points-to}(q) \subseteq \text{points-to}(p)$ .
- (3)  $p \xrightarrow{R} q =_{df} p$  represents the dereference of  $q$

---

Input (unordered):

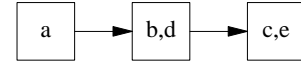
a = &b    b = &c  
a = &d    d = &e

Anderson



Final points to sets  
points-to(a) = {b, d}  
points-to(b) = {c}  
points-to(d) = {e}

Steensgaard



Final points to sets  
points-to(a) = {b, d}  
points-to(b) = {c, e}  
points-to(d) = {c, e}

---

**Figure 1. An example showing the the final points-to graphs produced by Anderson's and Steensgaard's algorithms (Only 'A' edges between variables are shown).**

- (4)  $p \xrightarrow{W} q =_{df} p$  represents the dereference of variable  $x$  and  $*x = q$ .

The algorithm operates on normalized assignment statements. The following are examples of normalized assignment statements and the initial graph fragments they produce.

$p = q:$      $p \xrightarrow{G} q$   
 $p = *q:$      $p \xrightarrow{G} *q \xrightarrow{R} q$   
 $p = \&q:$      $p \xrightarrow{G} \&q \xrightarrow{A} q$   
 $*p = q:$      $*p \xrightarrow{R} p$  and  $*p \xrightarrow{W} q$

After construction of the initial graph, the following rules are applied until a fixed point is reached.

Rule 1:  $a \xrightarrow{G} b \xrightarrow{A} c \Rightarrow a \xrightarrow{A} c$

Rule 2:  $a \xrightarrow{R} b \xrightarrow{A} c \Rightarrow a \xrightarrow{G} c$

Rule 3:  $a \xrightarrow{G} b$  and  $a \xrightarrow{W} c \Rightarrow b \xrightarrow{G} c$

For example, Rule 1 states that if  $points\text{-}to(b) \subseteq points\text{-}to(a)$  and  $c$  is a member  $points\text{-}to(b)$  then  $c$  is also a member of  $points\text{-}to(a)$  (i.e., add the edge  $a \xrightarrow{A} c$ ). The final points-to set for variable  $v$  is the set of all nodes reachable from  $v$  via an  $A$  edge.

In the absence of casting, structure fields can essentially be treated as separate variables. C casts significantly complicate the analysis. Yong et. al., handle casts through the use of three functions *normalize*, *lookup*, and *resolve* [15]. The *normalize* function is used to ensure that all sub-fields of a structure that have the same offset within the structure are mapped to the same canonical representative. The *lookup* function is used to identify the field that is referenced by a dereferenced pointer. When the declared type of the pointer does not match the type of an object to which it might point, *lookup* returns a safe approximation to the set of fields that are actually referenced. Finally, the *resolve* function is used to match each field of a structure of one type with the corresponding field(s) of a structure of another type.

### 3. Points-To Data

This section presents points-to set data generated from a collection of 27 mid-sized programs using an implementation of Anderson’s algorithm. For some of the programs, the data was also collected with structure fields expanded. This is not done for all program for one of two reasons: either, in the case of some of the larger programs, a lack of memory prevented the analysis from finishing, or for many smaller examples the output with structure fields expanded is quite similar to that with fields collapsed.

To begin with, Table 1 shows the data for all 27 programs with structure fields collapsed and this uninteresting. In the table, the first three columns are general information about the program. They include each program’s name, its size (in lines of code as reported by the unix utility *wc*), and the user time taken to compute the points-to sets (during each execution, the program received essentially 100% of the CPU cycles and the system time was negligible).

While the times are not the focus of this paper, one thing that stands out is that program size can be a very poor indicator of the processing time required. Several smaller examples (e.g., *tile-forth* and *ijpeg*) take considerable time, while the largest example (*a2ps-4.12*) takes almost no processing time at all. This is unfortunate as it leads to a lack of stability in the analysis and is an area for further research.

The remaining columns in Table 1 concern the points-to sets:

PV

The number of pointer nodes (variables) in the graph.

ADDRs

The number of addresses taken in the program.

FN ADDRs

The number of unique functions that have their address’ taken.

PT sets

The number of unique pointer sets (post analysis).

Indirect call sites

The number of call sites through function pointers.

Indirect call sets

The number of equivalent sets of indirect calls (post analysis).

Pointers to functions are separated out from other pointers because they have a unique impact on many source code analysis algorithms. For example, the construction of an interprocedural control-flow graph needs to include call edges to the control-flow graph for each procedure potentially called at an indirect call-site.

Perhaps the most important column for those working on pointer analysis algorithms is the PT sets column, which shows the number of unique points sets. Any algorithm that can maintain the precision of the final output while pre-identifying (some of) the pointers that can end up with the same points-to set, can save considerable computation time [13, 7].

Table 1 includes two versions of *flex*; thus, it is possible to see the effects of software evolution on points-to information. In this case the program grew about 20%, between the two versions, while the number of pointers

Name	LOC	time (sec)	PV	ADDRs	FN ADDRs	PT sets	Indirect call sites	sets
copia	1168	0.090	5	3	0	1	0	0
compress	1423	0.020	29	5	0	7	0	0
which	2052	0.030	76	10	0	12	2	0
barcode	3164	0.150	311	48	21	33	6	2
wdiff-0.5	3641	0.040	71	6	0	4	1	0
tile-forth-2.1	3717	278.030	774	313	258	23	2	1
gcc.cpp	4079	0.410	941	39	11	56	5	1
bc	4962	0.130	670	70	11	45	23	5
indent-1.10.0	6119	1.610	761	124	1	23	1	1
byacc	6337	0.250	846	56	0	66	0	0
li	6916	69.480	2412	323	190	35	4	2
gnuchess	8722	0.320	8722	190	78	6	62	3
ed-0.2	10172	1.030	1634	165	3	207	0	0
capd	11068	0.700	1254	73	57	52	101	4
oracolo2	11524	0.320	1070	249	0	301	0	0
prepro	11524	0.300	1053	251	0	298	0	0
diff	11755	1.000	1500	480	15	110	3	3
find	15057	2.210	1178	110	117	115	22	3
flex-2-4-7	15143	0.210	651	25	0	54	0	0
ctags 5.0	15564	2.560	1760	82	62	46	5	3
flex-2-5-4	18100	0.250	712	27	0	66	0	0
espresso	22050	31.220	4016	284	17	250	15	7
jpeg	24814	147.700	2714	66	132	58	622	1
go	28547	1.050	43	137	0	30	0	0
sendmail	37576	40.040	2701	154	60	213	31	4
ntpd	45635	6.080	2759	308	162	160	13	7
a2ps-4.12	53131	0.380	974	104	3	115	0	0

**Table 1: Points-to Data with Collapsed Fields**

grew just under 10%. This may indicate that there is little maintenance baggage in the code.

Finally, program `which` in Table 1 has no function whose addresses is taken and yet there are 2 indirect calls. This occurs with utilities, which include fragments similar to the following (`p` is a function pointer that is called only when it is defined and this provides a *hook* to another programmers).

```
if( p )
    p();
```

For selected programs, Table 2 shows the same data as Table 1, but this time with structure fields expanded. Many programs in this table illustrate the expected increase in pointer variables that should accompany the more precise analysis (see for example `tile-forth`). However, some interesting “anomalies” arise. For example, the number of pointers represented in the graph of `barcode` goes *down* from 311 to 288 when structure fields are expended. This counter intuitive result occurs because non-pointer fields of a structure must be treated as pointer fields in the collapsed approach. In `barcode` initialization and command line argument processing is performed using an array of structures describing

command line options. This array include pointers to specific initialization functions. Thus, it is possible for expansion of pointer fields to decrease the number of pointers by disambiguating the pointers from other fields in the structure.

As expected, there is a minimal increase in the the number of unique pointer sets (post analysis) when structure fields are expanded. Finally, note in Figure 2, the dramatic drop in the number of pointer variables for `gnuchess`. This occurs because other structure fields are no longer “lumped together” with pointers fields.

Tables 3 and 4 show the total edge counts before and after the closure. In Table 3, structure fields are collapsed, while in Table 4 they are expanded. Note the R and W edges are not added during closure; thus, their counts remain the same. For the most part, the trend is as expected: a greater number of initial edges produces a greater number of final edges. However, programs `go` and `li` are interesting exceptions to this rule. The closure adds very few A edges to `go`, which starts with quite a few. On the other hand, `li` starts with an average number of A edges, but ends up with the largest number of such edges.

An unexpected result that was observed while preparing these tables is illustrated by `espresso` (and to a lesser extent by `gcc.cpp`, `oracolo2`, and `prepro`). The final number of A edges, and thus, the final points-to sets are smaller when the more precise structure field expansion algorithm is used. This has a noticeable affect on the computation time. Especially for `espresso` whose processing time dropped form 31 seconds to just over 3 seconds.

The final table, Table 5, presents edge count histograms before and after closure for both the collapsed and expanded structure fields. Only A and G edges are shown. In each histogram, the number of outgoing A and G edges for each node are placed in one to six categories: those having 1, 2, 3, 4-10, 11-99, and 100 or more outgoing edges. For A edges this gives a break down of the size of the points-to set for the nodes. For G edges it gives a measure of the containment of pointer subsets.

One area of ongoing research on points-to algorithms deals with detecting, as early as possible, those variables that will have the same points-to set. This is because,

rediscovery of the same large points-to sets can account for a significant amount of the computation time. Dynamic detection of such equivalences is one way to reduce the over all computation cost. One tool that performs such detection is the BANE Pointer Analysis System [13].

The histograms for G edges indicate concentrations of G edges. Nodes with a large number of outgoing G edges represent *core* collections of addresses. Like those with large number of A edges, early discovery and improved representation for these nodes should save significant computation time.

Finally, Figures 2-5 present two different views with and with-out expended fields on the size of the pointer information. A log scale is used on both axes of all four figures to help separate out changes at smaller values and to stop larger programs from obscuring the remaining data. In all four figures, the vertical axis is the number of elements in a set. In Figures 2 and 4 each point on the horizontal axis represents a pointer (*i.e.*, a points-to set). These are sorted by the set size. Thus, long flat sections represent areas in which a collection of nodes have the same size points-to sets. In many cases these nodes have the same points-to sets and this indicates a place for potential improvement in the closure algorithm.

In Figures 3 and 5, each point on the horizontal axis represents a unique set of pointers in the final points-to output. In essence these figures show the goal for any points-to preprocessing algorithm. That is, if one wants to write an algorithm to preprocess the input by discovering a-priori pointers that will have the same points-to sets, Figures 3 and 5 represent to optimal preprocessing. Since no preprocessing was performed when generating the data shown in Figures 2 and 4, they show the actual initial graph constructed. Thus, the possibilities for preprocessing and cycle detention are highlighted by comparing corresponding programs from corresponding graphs.

It is also interesting to compare Figures 2 and 3. Take for example the program `tile-forth` (the tallest line in each figure). The length of the top horizontal bar in Figure 2 when compared to that of the similar bar in 3 illustrates that there is considerable room for improvement. The shorter bar represent the actual number of sets required, while the longer bar represent the number

Name	LOC	time	PV	ADDRs	FN ADDRs	PTS	Indirect call sites	sets
compress	1423	0.020	29	5	0	7	0	0
barcode	3164	0.090	288	48	21	37	6	3
tile-forth-2.1	3717	2275.810	4515	313	260	22	2	1
gcc.cpp	4079	0.270	1647	41	11	62	5	1
li	6916	107.640	7094	323	190	36	4	2
gnuchess	8722	0.370	190	69	6	62	3	1
oracolo2	11524	0.300	1025	261	0	301	0	0
prepro	11524	0.290	1025	263	0	298	0	0
ctags5.0	15564	12.130	5682	90	67	50	5	3
espresso	22050	3.570	5220	290	17	256	15	7
a2ps-4.12	53131	0.510	1253	111	3	116	0	0

**Table 2: Points-to Data for Expanded Fields**

name	Before Closure				After Closure			
	A	G	R	W	A	G	R	W
a2ps-4.12	826	3536	1094	410	3893	5957	1094	410
barcode	291	1000	239	70	8943	4501	239	70
bc	287	967	255	87	3118	2962	255	87
byacc	392	2282	736	178	8547	5900	736	178
capd	822	2447	811	214	39689	18783	811	214
compress	44	286	59	18	127	432	59	18
copia	264	133	28	4	304	791	28	4
ctags 5.0	1307	2987	967	333	165929	51899	967	333
diff	622	3744	865	322	78537	26685	865	322
ed-0.2	778	4066	754	274	70160	25308	754	274
espresso	1570	6781	2241	952	954048	320184	2241	952
find	662	2523	793	366	114764	50944	793	366
flex-2-4-7	415	2388	693	182	5458	5715	693	182
flex-2-5-4	465	2894	888	193	8582	7872	888	193
gcc.cpp	273	1896	588	207	38406	14133	588	207
gnuchess	1331	6032	1570	351	2559	8913	1570	351
go	8317	25509	8797	676	10771	40065	8797	676
ijpeg	730	7045	2506	1397	840821	448604	2506	1397
indent-1.10.0	407	1447	348	164	101271	23210	348	164
li	1104	1991	411	155	1152624	198971	411	155
ntpd	2022	5897	1571	991	201893	89788	1571	991
oracolo2	690	1052	459	401	13322	6899	459	401
prepro	694	1031	452	393	12896	6804	452	393
sendmail	2041	4531	1500	742	302328	139711	1500	742
tile-forth-2.1	1341	2253	648	107	759758	247418	648	107
wdiff-0.5	99	368	87	26	205	551	87	26
which	94	317	53	7	199	442	53	7

**Table 3: Total edge counts computed with collapsed structure fields**

name	Before Closure				After Closure			
	A	G	R	W	A	G	R	W
a2ps-4.12	1057	3857	1102	425	4700	11567	1102	425
barcode	291	1009	239	74	940	1708	239	74
compress	44	286	59	18	127	432	59	18
ctags 5.0	1455	3082	973	386	438112	227769	973	386
espresso	1571	6974	2268	1032	189554	69777	2268	1032
gcc.cpp	279	1974	589	226	18690	9268	589	226
gnuchess	1382	6076	1568	351	2586	9989	1568	351
li	1124	1995	412	160	1404316	1086305	412	160
oracolo2	729	1058	459	414	3429	7232	459	414
prepro	733	1037	452	405	3503	7099	452	405
tile-forth-2.1	1490	30928	659	114	4274768	4495493	659	114

**Table 4: Total edge counts computed with expanded structure fields**

		Edge Count Histograms											
program	edge kind	Collapsed Structure Fields						Expanded Structure Fields					
		1	2	3	4-10	11-99	100+	1	2	3	4-10	11-99	100+
a2ps-4.12	A	826	0	0	0	0	0	1057	0	0	0	0	0
	G	1221	862	32	26	10	1	1159	962	29	28	22	1
post closure	A	1993	675	123	34	0	0	1648	997	205	78	1	0
	G	2166	1224	135	71	19	1	2493	1619	160	402	112	1
ctags 5.0	A	1272	0	0	0	2	0	1420	0	0	0	2	0
	G	1131	665	34	30	9	0	1261	696	31	24	6	0
post closure	A	1947	91	0	0	2189	0	2007	93	1	0	5704	9
	G	3483	796	75	91	660	2	5231	928	84	123	817	1209
gcc.cpp	A	258	2	0	0	1	0	264	2	0	0	1	0
	G	838	445	10	22	1	0	859	466	10	23	3	0
post closure	A	552	34	3	1	1398	0	576	175	17	1329	468	0
	G	1349	496	26	42	403	0	1928	616	31	419	178	0
gnuchess	A	1323	1	0	1	0	0	1374	1	0	1	0	0
	G	1157	2150	22	57	7	0	1199	2169	29	58	6	0
post closure	A	2398	51	1	12	0	0	2407	60	1	12	0	0
	G	3297	2238	40	126	15	0	4276	2287	50	138	13	0
li	A	931	0	0	0	0	1	952	0	0	0	0	1
	G	1149	218	18	14	2	1	1148	219	18	15	2	1
post closure	A	977	8	1	11	3	2976	1178	14	1	40	3	7123
	G	2259	249	155	147	20	773	2424	299	175	218	62	4849
tile-forth-2.1	A	935	190	0	0	1	0	1468	0	0	0	1	0
	G	502	305	274	4	2	1	1344	290	1	3	142	14
post closure	A	1043	5	0	0	0	1350	1035	153	0	0	0	4907
	G	927	298	4	7	565	502	1772	314	4	9	146	4461

**Table 5: A and G edge-count histograms**

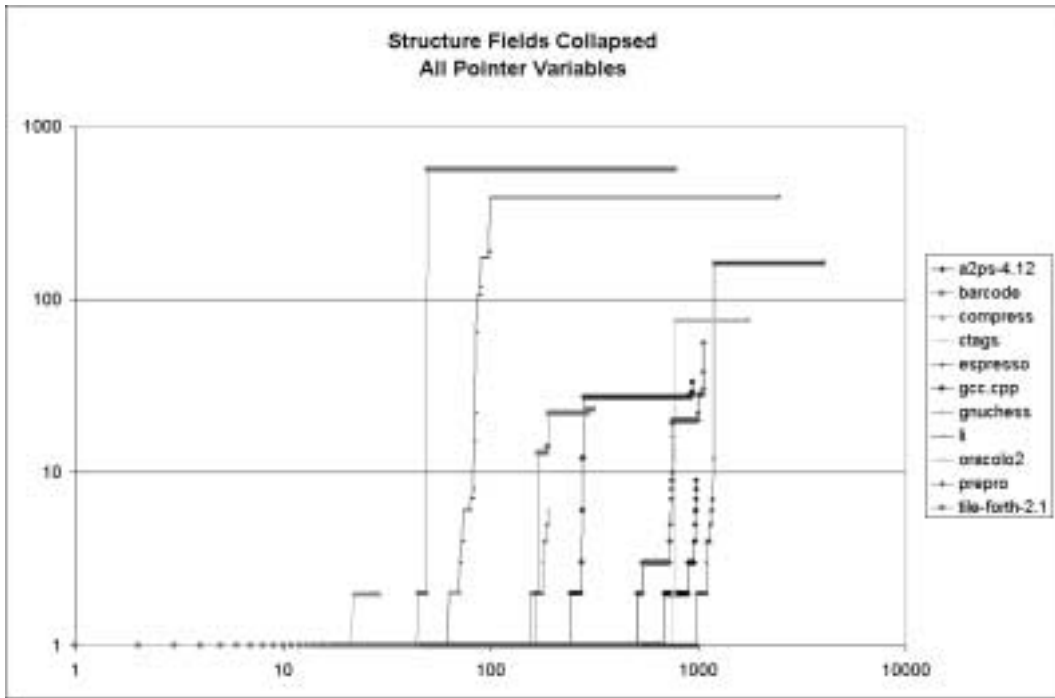


Figure 2. The size of the pointers sets for each pointer variable. Sorted by size.

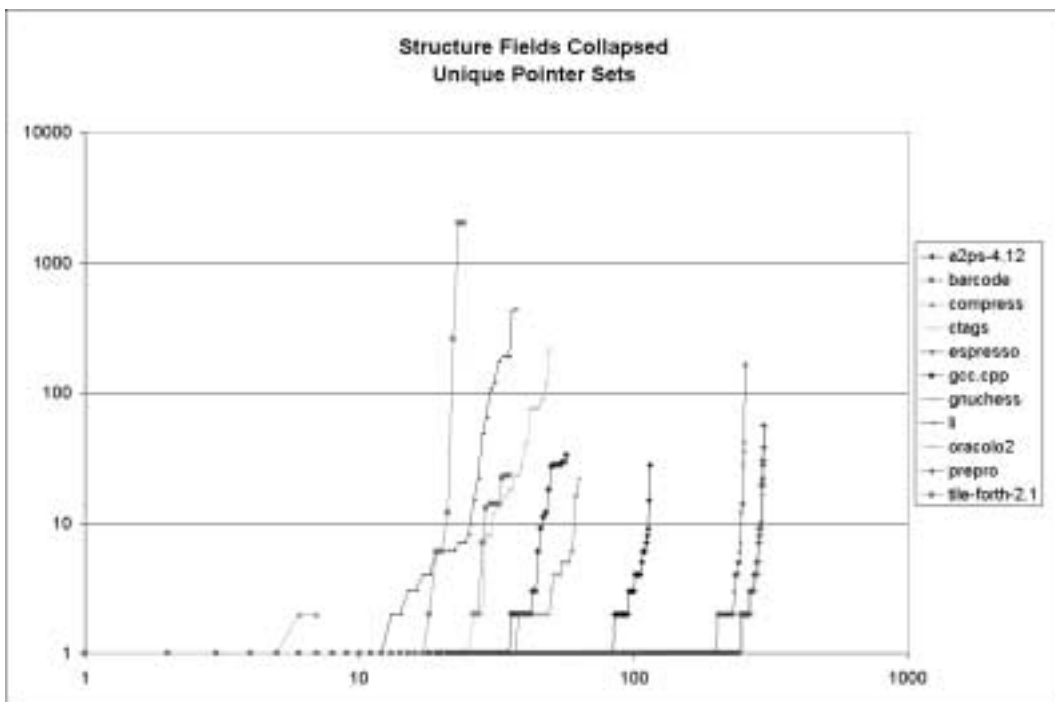


Figure 3. The size of the pointers sets for each unique pointer set. Sorted by size.

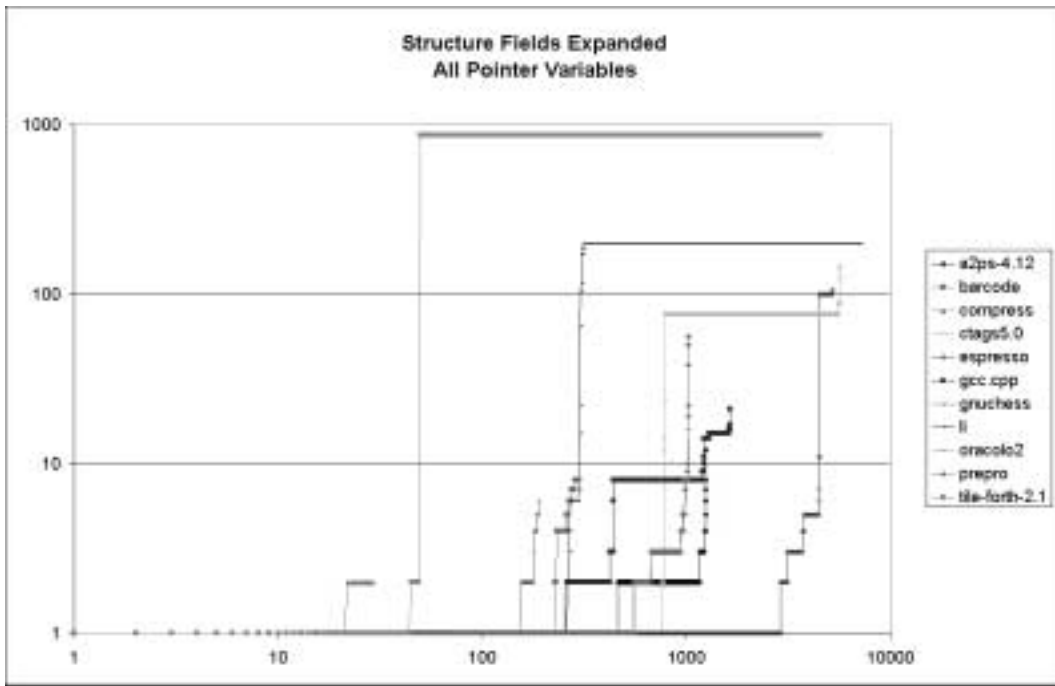


Figure 4. The size of the pointers sets for each pointer variable. Sorted by size.

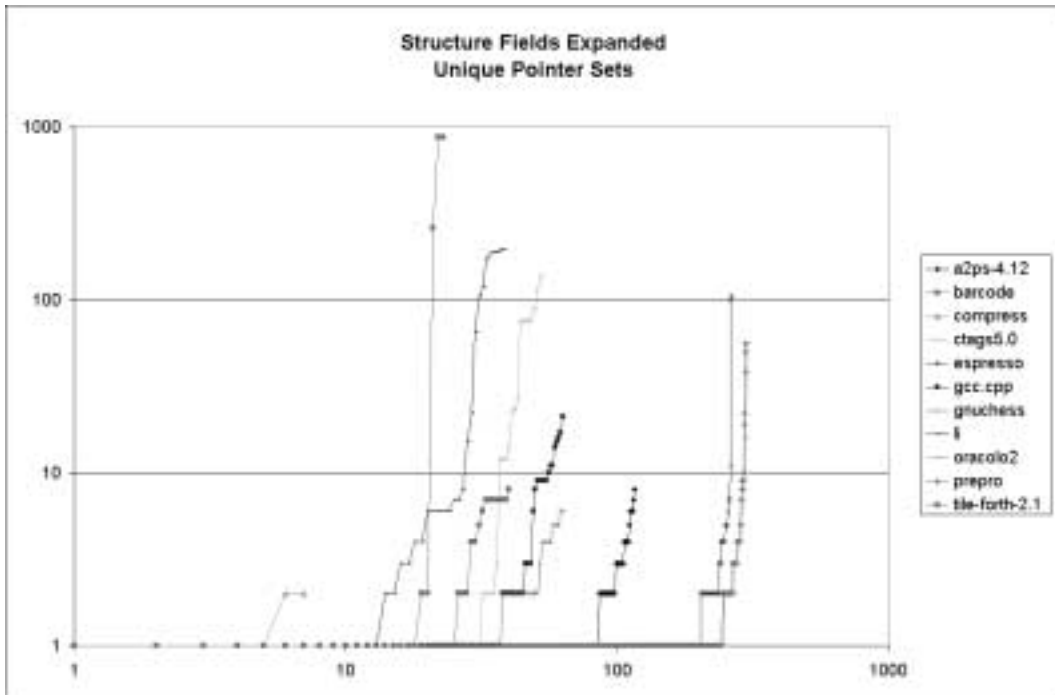


Figure 5. The size of the pointers sets for each unique pointer set. Sorted by size.

computed by Anderson's algorithm. In contrast, the similar shapes in the two graph for the `prepro` program indicate that preprocess would not help much for this program.

Finally, comparing Figures 3 and 5, expanding pointer fields does not change the shape for the unique pointers sets. The expansion has a more pronounced affect on the graphs of all pointer variables as seen by comparing Figures 2 and 4.

#### 4. Summary

The output of points-to algorithms is used by an increasing number of program analysis tools. The data presented in the paper is a starting point for considering improvements to algorithms that produce points-to information and also improvements to the algorithms that use this information.

This paper has identified several areas for future work. For example, the time variation in computing points-to sets is unfortunate as it leads to a lack of stability in the analysis and consumers of the analysis. Thus, one area of future work is developing more stable algorithms for pointer analysis.

Algorithms for reducing the time taken to compute points-to sets can exploit patterns in the data that as exemplified in Section 3. Such algorithms can attempt to statically precompute or dynamically compute nodes that will have the same points-to sets. Such nodes come from cycles in the graph. For example, if  $a \xrightarrow{G} b$ ,  $b \xrightarrow{G} c$ , and  $c \xrightarrow{G} a$  then  $a$ ,  $b$ , and  $c$  will all have the same points to sets because the three G edges imply that the pointer information of  $a$  is a subset of that for  $b$  which is a subset of that for  $c$ , which finally is a subset of  $a$ . A related idea is to look for other patterns besides cycles. Examples include "chains" or "ladders" with in "forks." If nodes included in such structures can be shown to have the same points-to sets then computation time can be reduced.

Finally, as the limits of true algorithmic improvement are reached, statistical techniques can be considered. For example, hardware branch prediction is not guaranteed to improve execution performance; however, in reality it works quite well. Similarity, the data presented in Section 3 can be used as a basis to guide the construction of

algorithms that may not have clearly better theoretical complexity, but that perform better on the kinds of programs that are used as input to pointer analysis algorithms.

#### REFERENCES

1. Anderson, L.O., "Program analysis and specialization for the C programming language." Ph.D. thesis, DIKU University of Copenhagen (DIKU report 94/19) (May 1994).
2. ANSI, "American National Standard for Information Systems — programming Language — C," *ANSI X3.159-189/FIPS PUB 160*, (December 1989).
3. Burke, M., Carini, P., Choi, J.D., and Hind, M., *Flow insensitive interprocedural alias analysis in the presence of pointers*. August 1994..
4. Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 47-56 (July 1988).
5. Choi, J.D., Burke, M., and Carini, P., "Efficient flow sensitive interprocedural computation of pointer induced aliases and side-effects," *In ACM Symposium on Principles of Programming Languages*, pp. 232-245 (1993).
6. Emami, M., Ghiya, R., and Hendren., L., "Context sensitive interprocedural points-to analysis in the presence of function pointers," *In SIGPLAN Conference on Programming Languages Design and Implementation*, (1994.).
7. Heintze, N. and Tardieu, O., "Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," *In Proceedings of the SIGPLAN 01 Conference on Programming Language Design and Implementation (Snowbird, UTAH)*, (June 2001).
8. Horwitz, S., "Precise flow-insensitive may-alias analysis is NP-Hard," *ACM Transactions on Programming Languages and Systems* **19**( 1)(January 1997).
9. Landi, W. and Ryder, B., "A safe approximate algorithm for interprocedural pointer aliasing," *In*

*SIGPLAN Conference on Programming Languages Design and Implementation*, pp. 235-248 (June 1992).

10. Ruf, E., "Context-sensitive alias analysis reconsidered," *In SIGPLAN Conference on Programming Languages Design and Implementation*, pp. 13-22 (June 1995).
11. Shapiro, M. and Horwitz, S., "Fast and accurate flow-insensitive points-to analysis," *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Paris, France), (January 1997).
12. Steensgaard, B., "Points-to analysis in almost linear time," *International Conference on Compiler Construction*, (April 1996).
13. Su, Z., Fähndrich, M., and Aiken, A., "Projection Merging: Reducing Redundancies in Inclusion Constraint Graphs," *In the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*. Boston, MA., (January 2000).
14. Wilson, R. and Lam., M., "Efficient context-sensitive pointer analysis for C programs," *In SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-12 (June 1995).
15. Yong, S., S.Horwitz, and Reps, T., "Pointer analysis for programs with structures and casting," *In Proceedings of the SIGPLAN 99 Conference on Programming Language Design and Implementation (Atlanta, GA)*, (May 1999).
16. Zhang, S., Ryder, B. G., and Landi, W., "Program decomposition for pointer aliasing: a step towards practical analyses," *Proceedings of the 4th Symposium on the Foundations of Software Engineering (FSE'96)*, (October, 1996).