Flow Insensitive Points-to Sets

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

David Binkley[†]
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@grammatech.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 1,168 to 87,579 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.

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 1,168 to 87,559 lines of code). The data illustrates the relative complexities of computing points-to sets. It should be useful in developing better pointer analysis software and other source code analysis algorithms.

Pointer analysis can be performed in a flow-*sensitive* or a flow-*insensitive* manner. Flow-sensitive analysis [10, 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 [17, 3, 13, 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, 15, 11].

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 appears that `s`, `s.x`, and `s.y` may all point to `a` and `b`. Various techniques for handling expanded structure files are possible.

[†] While on sabbatical leave from Loyola College in Maryland.

The algorithms use to generate the data presented in Section 3 are based on the work of Yong et. al. [16].

This paper considers two algorithms that perform flow and context insensitive points-to analysis. Both algorithms are first considered with structure fields collapsed and then with structure fields expanded. 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 background on points-to analysis and the algorithms used to collect the data presented in the next section. In general, flow insensitive points-to analysis is an NP-Hard problem [9]; consequently, existing practical algorithms are all approximations. The precision of these algorithms ranges from the $O(n\alpha(n,n))$ (there α is the inverse of Ackermann's function) algorithm of Steensgaard [13] to the $O(n^3)$ algorithm of Andersen [1]. Shapiro and Horwitz [12] have presented a spectrum of point-to analysis algorithms that range in cost and precision from Steensgaard's to Andersen's. Figure 1, which is based on the figure from Shapiro and Horwitz, highlights the key differences between the two approaches.

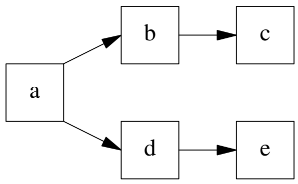
The data presented in Section 3 was produced by two algorithms. Both have the same precision (produce the same output). The first is an implementation of Andersen's algorithm [1] and second an extension of Fahndrich's algorithm [7]. Both algorithms read in normalized statements that represent the pointer manipulations in the program. A graph is constructed from these statements and then a closure of this graph is computed. Finally, points-to sets are extracted from the closed graph.

The two approaches differ in the graph they manipulate. The graph built by Andersen's algorithm consists of nodes that represent memory locations and edges that represent relations between 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:

Input (unordered):

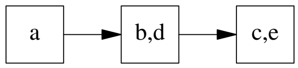
a = &b   b = &c
a = &d   d = &e

Andersen



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 Andersen's and Steensgaard's algorithms. (Only Address (A) edges between variables are shown.)

(1) $p \xrightarrow{A} q \;=_{df}\;$ q is in the points-to set of p.

(2) $p \xrightarrow{G} q \;=_{df}\; \textit{points-to}(q) \subseteq \textit{points-to}(p)$.

(3) $p \xrightarrow{R} q \;=_{df}\;$ p represents the dereference of q

(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.

$p = q$: $\quad p \xrightarrow{G} q$

$p = *q$: $\quad p \xrightarrow{G} *q \xrightarrow{R} q$

$p = \&q$: $\quad p \xrightarrow{G} \&q \xrightarrow{A} q$

$*p = q$: $\quad *p \xrightarrow{R} p$ 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-to*$(b) \subseteq$*points-to*(a) and c is a member of *points-to*(b) then c is also a member of *points-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.

The second algorithm is based on the work of Fahndrich et. al. [7]. Their technique is actually a general constraint solver. The following description is specialized to the problem of pointer analysis.

The approach manipulates "*ref*" structures, which are used to represent taking the address of and dereferencing variables. The graph can be thought of as having three parts (often drawn as columns). Column 1 entries, denoted by $ref(V,V)$, represent the address of a variable, Column 2 entries, denoted simply as V, represent variables, and Column 3 entries represent dereferenced variables. There are two forms. Dereferences on the right-hand side of an assignment are denoted by $ref(V,\emptyset)$, while left-hand side dereferences are denoted by $ref(1,V)$, where "1" represents the universal set.

The Fahndrich algorithm differs from the Andersen algorithm in that a complete closure of the graph is not computed. Rather, the points-to set for a variable V is computed "on the fly" when it is needed. This set contains all the variables A for which there is a path from $ref(A,A)$ to V. It is obtained by walking edges backwards from V and gathering up nodes from Column 1.

The algorithm finds paths from Column 1 entries to Column 3 entries. Take, for example, a path from Column 1 entry $ref(V,V)$ to Column 3 entry $ref(A,B)$. In terms of the constraint solver, this path is equivalent to the constraint $ref(V,V) \subseteq ref(A,B)$. Because both terms are *ref* terms the corresponding arguments yield additional constraints. These constraints differ in that the first argument of a *ref* is *contravariant*, while the second is *covariant* [7]. The contravariant constraint generates the new constraint $V \subseteq A$, while the covariant constraint generates the new constraint $B \subseteq V$. These new constraints are represented as new edges in Column 2 of the graph.

For pointer analysis, the algorithm repeats the following process until there are no changes. For each entry $ref(1,V)$ in Column 3, let P represent its point-to set (*i.e.*, entries from Column 1 that have a path to $ref(1,V)$). For each such entry $ref(U,U)$ in P, add an edge (in Column 2) from V to U. This edges arises from the constraints $U \subseteq 1$ and $V \subseteq U$. Next, for each entry $ref(V,\emptyset)$ in Column 3, let Q represent its points-to set. For each entry $ref(U,U)$ in Q, add an edge (in Column 2) from U to V. This edges arises from the constraints $U \subseteq V$ and $\emptyset \subseteq U$. When no changes occur, the points to set for each variable V is the set of Column 1 entries having a path to V.

Fahndrich et. al. describe four optimizations that improve the efficiency of their constraint solver. Two of these are essential for pointer analysis. The first collapses cycles found in Column 2. Cycles occur when a set of variables must all have the same points-to set. For example, when $A \subseteq B \subseteq C \subseteq A$. By collapsing such cycles to a single node, significant redundant work is avoided.

The second optimization caches points-to sets associated with Column 2 nodes. The cache is invalidated on start of each iteration of the main loop. Note that within an iteration the cache may be out of date (always a subset of the actual points-to set). This only occurs as the result of a change; thus, an updated value will be computed and used on subsequent iterations.

The implementations of both algorithms can handle C structures and casting (in the absence of casting, structure fields can essentially be treated as separate variables). C casts significantly complicate the analysis. The technique for handling casting used in both approaches is based on the work of Yong et. al. The technique handles casts through the use of three functions *normalize*, *lookup*, and *resolve* [16]. 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 29 mid-sized programs using implementations of the Andersen and Fahndrich algorithms. Section 3.1 presents data collected with structure fields collapsed. For some of the programs, Section 3.2 presents data collected with structure fields expanded. This is not done for all programs for one of three reasons: either, the output with structure fields expanded is quite similar to that with fields collapsed, the output was similar to another example, or in the case of some of the larger programs using the Andersen algorithm, a lack of memory prevented the analysis from completing. Finally, Sections 3.3 and 3.4 present edge count histograms, and output data, respectively.

3.1. Collapsed Fields

To begin with, Table 1 shows base data for all 29 programs with structure fields collapsed. In the table, the first four 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, particularly for Andersen, is that program size is a very poor indicator of the processing time required. Several smaller examples (*e.g.*, `tile-forth` and `li`) require considerable processing time, while the fourth largest example (`ntpd`) takes comparatively little processing time. 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 unique function pointer sets (post analysis).

Pointers to functions are separated out 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 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. This column 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 will end up with the same points-to set, can save considerable computation time [14, 8].

Table 1 includes two versions of `flex`; thus, it is possible to observe 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 grew just under 10%. This may indicate that there is little maintenance baggage in the code.

Finally, an apparent anomaly. Program `which` has no functions whose addresses are taken and yet there are 2 indirect calls. This occurs with utilities that include fragments similar to the following (where `p` is a function pointer called only when it is defined and thus provides a *hook* to other programmers).

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

Table 2 shows the total edge counts before and after closure, collected using the Andersen algorithm. Since *R* and *W* edges are not added during closure, they are only

Name	LOC	time (sec) Andersen	Fahndrich	PV	ADDRs	FN ADDRs	PT sets	Indirect call sites	sets
`copia`	1168	0.09	0.08	5	3	0	1	0	0
`compress`	1423	0.02	0.04	29	5	0	7	0	0
`which`	2052	0.03	0.09	76	10	0	12	2	0
`barcode`	3164	0.15	0.22	311	48	21	33	6	2
`wdiff 0.5`	3641	0.04	0.13	71	6	0	4	1	0
`tile-forth 2.1`	3717	278.03	8.72	774	313	258	23	2	1
`gcc.cpp`	4079	0.41	0.23	941	39	11	56	5	1
`bc`	4962	0.13	0.23	670	70	11	45	23	5
`indent 1.10.0`	6119	1.61	0.22	761	124	1	23	1	1
`byacc`	6337	0.25	0.22	846	56	0	66	0	0
`li`	6916	69.48	2.03	2412	323	190	35	4	2
`gnuchess`	8722	0.32	0.33	190	78	6	62	3	1
`ed 0.2`	10172	1.03	0.51	1634	165	3	207	0	0
`capd`	11068	0.70	0.45	1254	73	57	52	101	4
`oracolo2`	11524	0.32	0.35	1070	249	0	301	0	0
`prepro`	11524	0.30	0.35	1053	251	0	298	0	0
`flex 2-4-7`	15143	0.21	0.25	651	25	0	54	0	0
`ctags 5.0`	15564	2.56	0.62	1760	82	62	46	5	3
`find`	16890	2.21	0.50	1178	110	117	115	22	3
`flex 2-5-4`	18100	0.25	0.39	712	27	0	66	0	0
`diff`	18374	1.00	0.64	1500	480	15	110	3	3
`espresso`	22050	31.22	2.36	4016	284	17	250	15	7
`ijpeg`	24814	147.70	3.27	2714	66	132	58	622	1
`go`	28547	1.05	1.14	43	137	0	30	0	0
`sendmail`	37576	40.04	2.39	2701	154	60	213	31	4
`ntpd`	45635	6.08	1.27	2759	308	162	160	13	7
`a2ps-4.12`	53131	296.72	3.72	974	104	3	115	0	0
`gnugo`	79125	-	55.17	5345	1648	310	1848	4	2
`cvs-1.11.1p1`	87557	-	9.49	9699	779	145	1300	126	18

Table 1: Basic program data (fields collapsed)

shown once. The table also shows, in final two columns, the change in the number of A and G edges. These are given as multipliers; thus, for example, `ijpeg` has 1151.8 times as many A edges in the final graph graph as in the initial graph.

For the most part, the trend is as expected: a greater number of initial edges produces a greater number of final edges. An interesting exception is the program `go` which ends up with comparatively few A edges given that it starts with quite a few. The trend is most pronounced in the programs `ijpeg`, `espresso`, `li`, and `tile-forth`. Each has a growth of over 500 times in the number of A edges. A similar growth occurs in the G edge count for the program `li` and `tile-forth`.

This growth accounts for significant processing time. One area of ongoing research on reducing this time, is to detect, as early as possible, those variables that will have the same final points-to set. Avoiding the rediscovery of the same (large) points-to sets can save significant computation time.

name	Initial Graph R	W	A	G	Final Graph A	G	Change ΔA	ΔG
a2ps-4.12	1094	410	826	3536	3893	5957	4.7	1.7
barcode	239	70	291	1000	8943	4501	30.7	4.5
bc	255	87	287	967	3118	2962	10.9	3.1
byacc	736	178	392	2282	8547	5900	21.8	2.6
capd	811	214	822	2447	39689	18783	48.3	7.7
compress	59	18	44	286	127	432	2.9	1.5
copia	28	4	264	133	304	791	1.2	5.9
ctags 5.0	967	333	1307	2987	165929	51899	127.0	17.4
diff	865	322	622	3744	78537	26685	126.3	7.1
ed 0.2	754	274	778	4066	70160	25308	90.2	6.2
espresso	2241	952	1570	6781	954048	320184	607.7	47.2
find	793	366	662	2523	114764	50944	173.4	20.2
flex 2-4-7	693	182	415	2388	5458	5715	13.2	2.4
flex 2-5-4	888	193	465	2894	8582	7872	18.5	2.7
gcc.cpp	588	207	273	1896	38406	14133	140.7	7.5
gnuchess	1570	351	1331	6032	2559	8913	1.9	1.5
go	8797	676	8317	25509	10771	40065	1.3	1.6
ijpeg	2506	1397	730	7045	840821	448604	1151.8	63.7
indent 1.10.0	348	164	407	1447	101271	23210	248.8	16.0
li	411	155	1104	1991	1152624	198971	1044.0	99.9
ntpd	1571	991	2022	5897	201893	89788	99.8	15.2
oracolo2	459	401	690	1052	13322	6899	19.3	6.6
prepro	452	393	694	1031	12896	6804	18.6	6.6
sendmail	1500	742	2041	4531	302328	139711	148.1	30.8
tile-forth 2.1	648	107	1341	2253	759758	247418	566.6	109.8
wdiff 0.5	87	26	99	368	205	551	2.1	1.5
which	53	7	94	317	199	442	2.1	1.4

Table 2: Andersen algorithm edge counts computed with collapsed structure fields

One method for avoiding rediscovery is to identify cycles of G edges For example, x and y will have the same points to sets if $x \xrightarrow{G} y$ and $y \xrightarrow{G} x$, (*i.e.*, $point\text{-}to(x) \subseteq points\text{-}to(y)$ and $points\text{-}to(y) \subseteq points\text{-}to(x)$). In general, such variables appear as strongly connected components (SCCs). The challenge in designing an algorithm for SCC collapse is to balance the cost of identification against the saving from collapsing SCCs.

While it does not compute a complete closure, the Fahndrich algorithm does perform dynamic detection of SCCs. The key to the Fahndrich algorithm's effectiveness is its balancing of the identification cost with the closure cost. The algorithm does not attempt for find all SCCs, but rather identifies only those that it encounters during the computation of a points-to set.

Table 3 shows the data collected for the Fahndrich algorithm. For each program, the second, third, and fourth columns count the number of edges that cross from Column 1 to Column 2, those between variables within Column 2, and those that cross from Column 2 to Column 3. With structure fields collapsed, edges are only ever added within Column 2. Thus, only Column 2 edges are reported in the final graph. The final Column shows the change in Column 2 edges.

Two things of note. First, there is a significant increase in Column 2 edges for the programs `copia` and `ijpeg`. Although it is less pronounced than with the Andersen algorithm, this increase requires significant computation effort. Program `ijpeg` is of particular interest as it also causes a large increase with the Andersen algorithm. Future work includes gaining a better understanding of why `ijpeg` (and `copia`) causes this behavior. Hopefully, this will lead to algorithm improvements.

The second thing to note is that the multipliers in the last column of Table 3 are considerable smaller than those for the Andersen algorithm. The largest growth factor is 8.7. In contrast, the largest growth factor for *A* and *G* edges shown in Table 2 are 1151.8 and 109.8 respectively. This is accounted for by two factors. First, the collapsing of SCCs, and second the Andersen algorithm adds edges to represent the transitive closure, while the Fahndrich algorithm repeatedly walks paths of edges. One way to understand this distinction is to consider the output of a points-to set. For the Andersen algorithm variable *V*'s points-to set consists of the variables to which *V* was an outgoing *A* edge. In contrast, the Fahndrich algorithm must perform a graph walk. (It is the presence of the points-to set cache that makes the Fahndrich algorithm tractable.)

	Initial Graph			Final Graph	
program	c12	c22	c23	c22	change
`a2ps-4.12`	2272	9664	4280	17642	1.8
`barcode`	217	3245	820	3939	1.2
`bc`	277	3200	839	3946	1.2
`byacc`	338	2313	912	3550	1.5
`cadp`	559	2462	1022	8546	3.5
`compress`	42	276	78	291	1.1
`copia`	264	133	32	779	5.9
`ctags 5.0`	1126	3243	1313	7288	2.2
`cvs-1.11.1p1`	4459	17824	6450	35150	2.0
`diff`	735	6966	2098	9075	1.3
`ed 0.2`	556	6038	1735	7101	1.2
`espresso`	1366	8279	3467	11435	1.4
`find`	705	3247	1357	5697	1.8
`flex 2-4-7`	365	2356	865	3855	1.6
`flex 2-5-4`	456	5585	1714	7762	1.4
`gcc.cpp`	198	1666	793	2640	1.6
`gnuchess`	518	6009	1978	7657	1.3
`gnugo`	4705	21823	5578	41700	1.9
`go`	858	25439	9487	36922	1.5
`ijpeg`	713	6959	4129	60592	8.7
`indent 1.10.0`	275	1365	514	1836	1.3
`li`	909	2012	565	3106	1.5
`ntpd`	1620	5666	2398	9763	1.7
`oracolo2`	460	3574	1418	5250	1.5
`prepro`	457	3559	1403	5261	1.5
`sendmail`	1833	4732	2309	10273	2.2
`tile-forth 2.1`	678	2264	764	2782	1.2
`wdiff 0.5`	134	2776	661	2976	1.1
`which`	65	2616	590	2660	1.0

Table 3: Fahndrich edge counts computed with collapsed structure fields

3.2. Expanded Fields

Tables 4, 5, and 6 repeat Tables 1, 2, and 3 with structure fields now expanded. Table 4 includes basic data for the 14 selected programs. Many of these show the expected increase in pointer variables that accompanies the more precise analysis (see for example `tile-forth`). However, some interesting "anomalies" arise. For example, the number of pointer variables for `barcode` , `cvs`, and `gnugo` actually *drop*. The drop for `barcode` and `cvs` is mild (about 10%). The drop for `gnugo` is almost 50%—from 5345 to 3065.

This counter-intuitive drop occurs because non-pointer fields of a structure must be treated as pointer fields in the collapsed approach. For example, in `barcode` initialization and command line argument processing is performed using an array of structures describing command lines options. This array includes pointers to specific initialization functions. Expansion of pointer fields decreases the number of "pointers" by disambiguating the pointers from other fields in the structure.

The only other attribute shown in Figure 4 that shows any significant change when compared with Figure 1 is the number of source of addresses taken (ADDRs). One cause of this increase is when the addresses of multiple structure fields are taken. In the collapsed model all such addresses are the same, while expansion represents each as unique addresses. The changes in pointer variables and addresses taken have a mild ripple effect in the other statistics reported on the table.

Name	LOC	time Andersen	time Fahndrich	PV	ADDRs	FN ADDRs	PTS	Indirect call sites	Indirect call sets
compress	1423	0.02	0.03	29	5	0	7	0	0
barcode	3164	0.09	0.39	288	48	21	37	6	3
tile-forth 2.1	3717	2275.81	132.10	4515	313	260	22	2	1
gcc.cpp	4079	0.27	0.22	1647	41	11	62	5	1
li	6916	107.64	20.65	7094	323	190	36	4	2
gnuchess	8722	0.37	0.43	190	69	6	62	3	1
oracolo2	11524	0.30	0.48	1025	261	0	301	0	0
ctags 5.0	15564	12.13	1.85	5682	90	67	50	5	3
espresso	22050	3.57	2.08	5220	290	17	256	15	7
ijpeg	24822	-	282.66	2413	69	48	113	622	1
ntpd	45647	-	616.49	2999	318	118	534	12	10
a2ps-4.12	53131	440.32	13.88	1253	111	3	116	0	0
gnugo	79125	-	5.61	3065	1647	310	1865	4	3
cvs-1.11.1p1	87557	-	73.40	8870	793	145	1355	126	22

Table 4: Basic program data (fields expanded)

name	Initial Graph A	Initial Graph G	Initial Graph R	Initial Graph W	Final Graph A	Final Graph G	ΔA	ΔG
a2ps-4.12	1102	425	1057	3857	4700	11567	4.4	3.0
barcode	239	74	291	1009	940	1708	3.2	1.7
compress	59	18	44	286	127	432	2.9	1.5
ctags 5.0	973	386	1455	3082	438112	227769	301.1	73.9
espresso	2268	1032	1571	6974	189554	69777	120.7	10.0
gcc.cpp	589	226	279	1974	18690	9268	67.0	4.7
gnuchess	1568	351	1382	6076	2586	9989	1.9	1.6
li	412	160	1124	1995	1404316	1086305	1249.4	544.5
oracolo2	459	414	729	1058	3429	7232	4.7	6.8
tile-forth 2.1	659	114	1490	30928	4274768	4495493	2869.0	145.4

Table 5: Andersen total edge counts computed with expanded structure fields

program	Initial Graph c12	Initial Graph c22	Initial Graph c23	Final Graph c12	Final Graph c22	Δc12	Δc22
a2ps-4.12	3403	10281	3169	5596	36111	1.6	3.5
barcode	223	3330	587	223	5035	1.0	1.5
compress	42	276	59	42	291	1.0	1.1
ctags 5.0	1381	3385	977	1469	13424	1.1	4.0
cvs-1.11.1p1	5163	16536	4616	9943	49461	1.9	3.0
gnugo	5371	21838	4667	5375	192860	1.0	8.8
ijpeg	3323	7025	2526	12491	79209	3.8	11.3
li	975	2016	413	969	7664	1.0	3.8
ntpd	3687	7045	1486	31305	38384	8.5	5.4
tile-forth 2.1	829	30937	666	895	4125	1.1	0.1

Table 6: Fahndrich total edge counts computed with expanded structure fields

Finally, the expansion of structure fields does not change the computation times for most programs. All the rest, with two exceptions, show the significant increase in computation time, expected with an increase in pointer variables (Column PV of Table 4). This is most pronounced in ntpd, which the Andersen algorithm failed to produce a solution and the Fahndrich algorithm takes 200 times longer.

The two exceptions are espresso and gnugo. For espresso, the Andersen algorithm drops it's computation time by a factor of 10. The result is comparable with the Fahndrich algorithm's time. For gnugo, only the Fahndrich algorithm produces a solution. Its time drops from by a factor of 10, from 55.17 seconds to 5.61.

Table 5 reports edges counts from the initial and final graphs along with the change in these counts for *A* and *G* edges collected from Andersen algorithm. Some of these multipliers are quite large. For example tile-forth has 2869 times as many *A* edges in the final graph as in the initial graph. For *G* edges the largest increase is 544.5 for the program li.

To get a feeling for the change caused by structure field expansion, compare the data in Table 5 with that of Table 2. Here the increase in precision has the expected increase in edges counts and multipliers. This can be seen, for example, in tile-forth, which shows a rise in the *A* edge multiplier from 1041 to 2869. The programs li and ctags show the largest rise in the *G* edge multiplier.

An unexpected result observed while preparing these tables is illustrated by barcode, espresso, and to a lesser extent by oracolo2. For these programs, the final number of *A* edges, and thus, the final points-to sets are *smaller* when the more precise structure field expansion is used. This has a noticeable affect on the computation time. Especially for espresso, which also shows a drop in the *G* multiplier, its processing time dropped from 31 seconds to just over 3 seconds.

Table 6 shows the initial and final edge counts for the Fahndrich algorithm. When structures fields are expanded it is possible to add edges from Column 1 to Column 2, so the table includes these counts and their change multiple.

The increase in precision that accompanies expanded structure fields has its cost. The largest increases for c22 edges are 11.3 for ijpeg and 8.8 for gnugo. While the largest increases for c12 edges are 8.5 for ntpd and 3.8 for ijpeg.

The most interesting number in Table 6 is the change in c22 edges from tile-forth, which actually drops (its multiplier is 0.1). This is caused by collapsing SCCs. Recall that the Fahndrich algorithm is the adding new edges and collapsing SCCs. These have the opposite effect on the edge count. Program tile-forth is unique in that collapsing is the dominant effect.

To compare the numbers of Tables 6 and 3, first note that c12 edges are not added when structure fields are collapsed; thus, Δc12 in Table 3 would be 1.0 for all programs. Only the programs ntpd and ijpeg show a significant increase in multiplier for c12 from 1.0 to 8.5 and 3.8, respectively. For c22 edges, expanding structure fields quadrupled the multiplier for gnugo and tripled the multiplier for ntpd.

3.3. Edge Histograms

The final two tables, Tables 7 and 8, show edge count histograms for the two algorithms. The histograms include edges counts before and after closure for both collapsed and expanded structure fields. In each histogram, variables (graph nodes) are placed in to one of six categories based on the number of outgoing edges. The categories are for variables with 1, 2, 3, 4-10, 11-99, and 100 or more outgoing edges.

For the Andersen algorithm, only *A* and *G* edges are shown. For *A* edges the histograms give a break down of the size of the points-to set. Nodes with a large number of outgoing *G* edges represent *core* collections of addresses.

For the Fahndrich algorithm only c22 edges are shown. These histograms can be compared with the *G* edge histograms for the Andersen algorithm and show a similar trend.

As expected there are fewer edges for the Fahndrich algorithm. One place for further study is indicated by the histogram for ijpeg, which shows more than 10 times as many edges in the final 100+ category when compared to the other examples in the histogram. Such nodes are particularly expensive to process.

3.4. Output Data

This section describes data collected from the output of the Andersen algorithm. The Fahndrich algorithm produces the same data (although per pointer data is more difficult to extract because of the SCC collapsing). The data concerns the size of the pointer information. It is presented in Figures 2-5. These four figures present two different views each with collapsed and expanded structure fields. A log scale is used on both axes of all four figures to stop larger programs from obscuring the data for smaller programs. In all four figures, the vertical axis measures set size. Note the factor of 10 difference in the vertical axis of Figure 3.

In Figures 2 and 4 each point on the horizontal axis represents a variable (a graph node). These values are sorted by 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 thus this indicates the potential for improvement in the closure process. In Figures 3 and 5, each point on the horizontal axis represents a unique set of pointers in the final points-to output. Again sorted by set size.

In essence, Figures 3 and 5 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 shows the optimal preprocessing. Since no preprocessing was performed when generating the data shown in Figures 2 and 4, the possibilities for preprocessing and cycle detention are highlighted by comparing, for corresponding programs, the horizontal bar length in corresponding graphs.

For example, compare the program `tile-forth` in Figures 2 and 3 (the tallest line in each figure). The shorter top horizontal bar in Figure 3 represents the actual number of sets required (for this particular size), while the corresponding longer bar in Figure 2 represents the number computed by the Andersen algorithm. Comparing the lengths of the two bars illustrates the considerable room for improvement. Evidence of this can be seen in the times reported in Table 1 where the Fahndrich algorithm takes considerably less time than the Andersen algorithm. In contrast, the similar shapes in the two graph for `pre-pro` indicate that preprocessing would be of little help (in Table 1 Andersen in actually faster on this example).

The effect of expanding structure fields on the pointer variables can be seen by comparing Figures 2 and 4. The figures have similar shapes. However, expanding structure fields has two effects on the points-to set sizes for each pointer variable. First the average set size decreases (the graphs in Figure 4 are in general shorter than those in Figure 2. `tile-forth` is an exception). Second, the expansion increases the number of pointers and thus the length of the horizontal bars for corresponding programs.

In contrast, to the Figure 2-4 comparison, comparing Figures 3 and 5, shows that the expansion does not change the shape for the unique pointers sets graphs. The only significant difference is difficult to see as the vertical axis in Figure 3 is 10 times longer (recall that the graph uses a log scale). Sizes of each pointer set in Figure 3 run about 50% times larger than in Figure 5.

4. Summary

The output from points-to analysis 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. Finding SCC is one such pattern. Other patterns include "chains" or "ladders." 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

program	edge kind	Collapsed Structure Fields 1	2	3	4-10	11-99	100+	Expanded Structure Fields 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
`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 7: Andersen edge count histograms

program	Collapsed Structure Fields 1	2	3	4-10	11-99	100+	Expanded Structure Fields 1	2	3	4-10	11-99	100+
`a2ps-4.12`	4340	502	250	240	66	2	4555	535	267	235	80	3
(post closure)	5945	871	350	382	150	6	6383	973	369	424	191	7
`ctags 5.0`	2028	113	38	47	13	3	2095	101	59	48	13	3
	2637	348	148	167	49	4	2752	370	169	383	80	3
`cvs-1.11.1p1`	9587	714	289	461	125	5	9103	747	281	436	106	5
	10529	1245	599	817	320	9	11246	1473	673	906	387	10
`ed 0.2`	2681	137	82	235	53	0	2720	132	86	233	59	0
	2548	174	55	182	53	4	2586	169	75	208	66	4
`gcc.cpp`	676	76	38	59	17	0	754	74	42	61	17	0
	704	104	59	66	24	1	796	133	60	98	41	2
`ijpeg`	2736	384	237	337	53	0	3190	363	204	297	50	0
	2637	407	228	247	115	126	2947	449	245	267	121	136
`li`	1587	108	30	20	1	0	1653	89	28	16	1	0
	1173	150	126	46	7	1	1220	151	125	44	10	1
`ntpd`	2617	271	103	155	37	4	2801	368	102	146	67	4
	2860	434	180	296	94	3	3182	588	180	341	241	3
`tile-forth 2.1`	1018	51	13	20	8	5	904	52	15	16	1923	6
	823	229	46	44	8	5	836	229	48	43	8	11

Table 8: Fahndrich edge count histograms

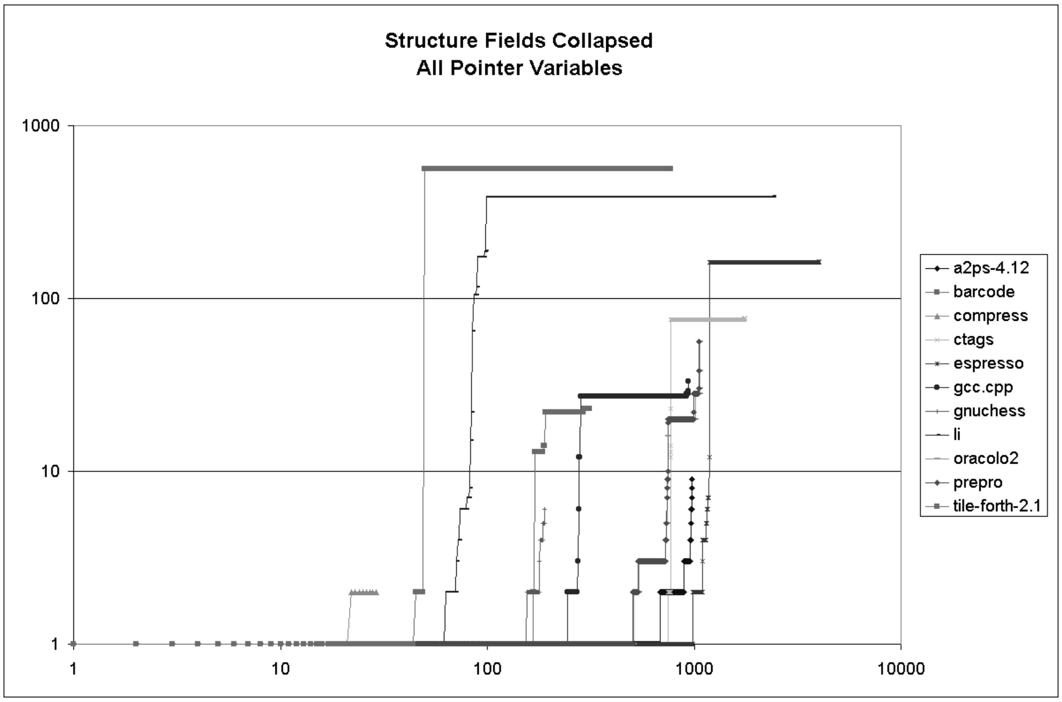


Figure 2. The size of the pointers sets for each pointer variable. Sorted by size.

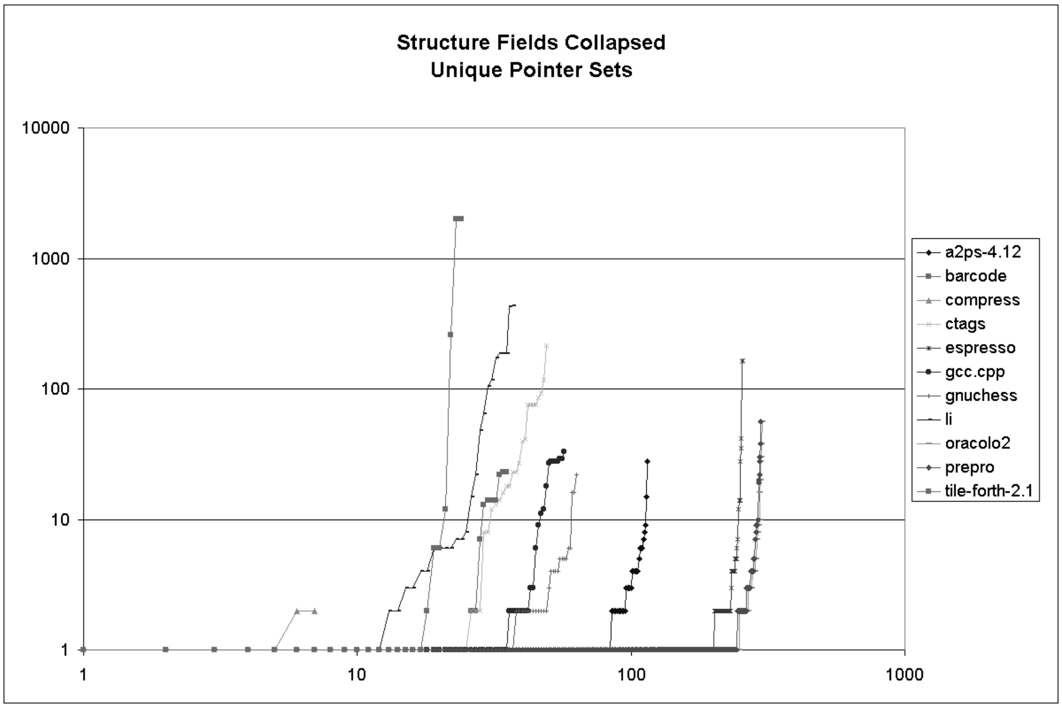


Figure 3. The size of the pointers sets for each unique pointer set. Sorted by size.

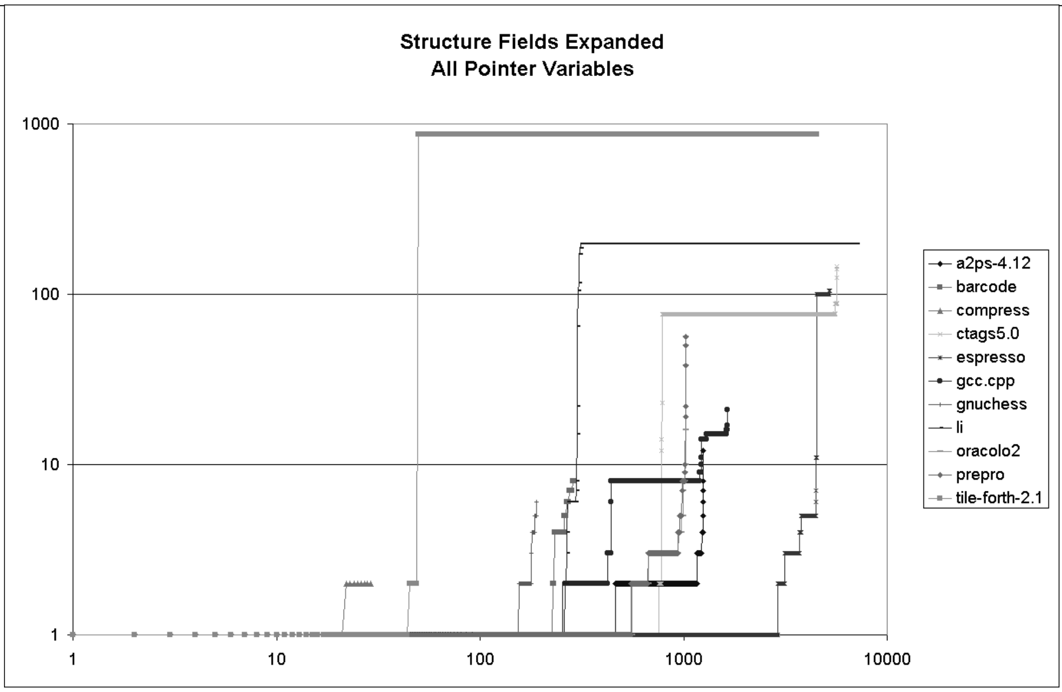


Figure 4. The size of the pointers sets for each pointer variable. Sorted by size.

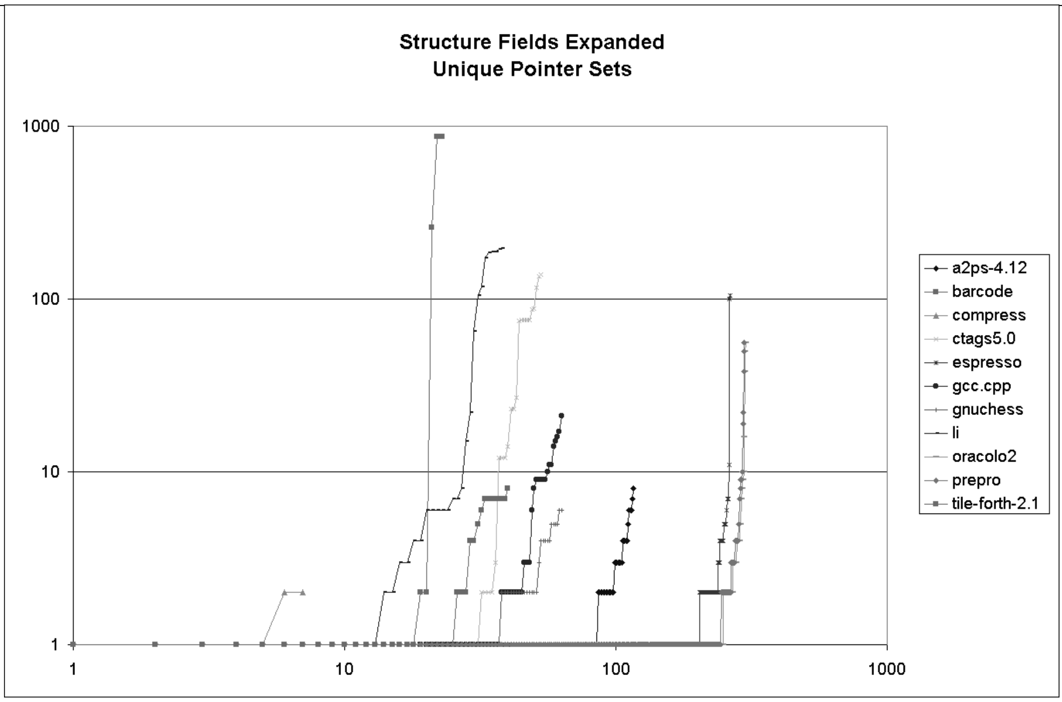


Figure 5. The size of the pointers sets for each unique pointer set. Sorted by size.

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. Andersen, 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. Fahndrich, M., Foster, J., Su, Z., and Aiken, A., "Partial Online Cycle Elimination in Inclusion Constraint Graphs," *Proceedings of the ACM SIGPLAN 98 Conference on Programming Language Design and Implementation,* (Montreal, Canada, 17-19 June 1998)*, ACM SIGPLAN Notices* **33**(5) pp. 85-96 (1998).

8. 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).

9. Horwitz, S., "Precise flow-insensitive may-alias analysis is NP-Hard," *ACM Transactions on Programming Languages and Systems* **19**(1)(January 1997).

10. 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).

11. Ruf, E., "Context-sensitive alias analysis reconsidered.," *In SIGPLAN Conference on Programming Languages Design and Implementation*, pp. 13-22 (June 1995).

12. 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).

13. Steensgaard, B., "Points-to analysis in almost linear time," *International Conference on Compiler Construction.*, (April 1996).

14. 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).

15. 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).

16. 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).

17. 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).