Experiments with Combined Analysis for Pointer Aliasing *

Sean Zhang Barbara G. Ryder William A. Landi

Department of Computer Science
Rutgers University
Hill Center, Busch Campus
Piscataway, NJ 08855
{xxzhang,ryder}@cs.rutgers.edu

Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
wlandi@scr.siemens.com

Abstract

We present initial empirical experiments with *combined analysis*, a scalable analysis technique that uses a program decomposition to apply different aliasing algorithms to independent program segments. The effectiveness of the solution strategy is validated through application to side-effect and reference analysis of C programs.

1 Introduction

For programming languages with general-purpose pointer usage (such as C), pointer aliasing is crucial to compile-time side-effect and semantic change analyses. It is also essential for software tool applications such as data-flow-based testing, debugging, semantics-based program understanding and program integration. Many techniques for pointer aliasing analysis have been presented in the literature. These analyses vary in the cost and precision of the aliasing information produced; their *scalability* is a difficult problem, as precision is usually sacrificed for acceptable cost on large programs. We have experimented with the application of aliasing analyses with different precision/cost characteristics to separate areas of a program to try to achieve scalability. The initial results of the experiments are reported here; the goal is to gain better understanding of how to obtain a scalable pointer aliasing analysis of acceptable precision for large programs. Measurement of analysis effectiveness is accomplished by using both side-effect and reference calculations on C programs.

Previously, we presented a program decomposition technique [23], which partitions the assignments in a program with respect to their aliasing effects. This is accomplished by calculating an equivalence relation on the names in the program. The equivalence relation induces a program decomposition which allows different aliasing analyses to be applied to independent parts of a program, that is, *program segments*. This technique is termed *combined analysis*. Given a program decomposition, a *particular* combined analysis assigns a specific analysis method to each program segment. In [23], we also compared (i) a whole-program analysis by a flow-sensitive and context-sensitive method (FS) [10], and (ii) a whole-program analysis by a flow-insensitive and context-insensitive method (FA), with (iii) one combined analysis using both FS and FA on different segments.

We have extended the decomposition technique and the FA algorithm to handle more features in C such as indirect calls through function pointers, unions and type casting [22]. For the first combined analysis, we repeat the comparison in [23] using more data. We wish to explore the cost/precision tradeoffs in the three methods. At present, FS is not scalable to all large (>10,000 lines of code) programs [19]. Investigating the application of FS to certain segments of large programs is of interest, because it has better precision than flow-insensitive methods. For the second combined analysis, we use another flow-insensitive and context-insensitive aliasing method (PT) [22] on those segments which have FA applied in the first one. In both combined analyses, we use the existence of recursive data structures in a program segment as a heuristic predictor of when FS will be too expensive; we assign FA or PT as the analysis for such a segment.

For the third combined analysis, we use a heuristic predictor of approximation in the FA analysis to choose which analysis, PT or FA, to assign to a program segment. FA, faster and less precise than PT, is used on the program segments where FA is expected to be sufficiently accurate; the other segments are analyzed with PT, presuming they need a more precise (but slower) analysis to obtain usable results. With this analysis, we hope to obtain an analysis of similar precision to that of PT, but more efficient.

These three sets of experiments represent initial findings; clearly, more experimentation is needed to provide strong evidence for choices between combined analyses.

*This research was supported, in part, by NSF grants CCR95-01761 and GER90-23628.

Nevertheless, the experiments did show that: (a) our heuristics for assigning analyses to program segments do not validate the utility of FS analysis over PT, but other heuristics must be tried before we can conclude that the precision of FS is not worth the cost, and (b) combined analysis allows application of FS to segments of a program too large to be analyzed by FS as a whole-program analysis. Several other interesting conclusions are suggested by our experiments, namely: (c) PT analysis seems better than FA analysis singly, and in combination with other techniques (e.g., FS) and (d) the second heuristic is more useful than the first because the gains in precision seem worth the cost.

The paper is organized as follows. In Section 2, we briefly discuss the program decomposition technique. In Section 3, we present the empirical results of the three combined analyses. Finally, we discuss related work in Section 4 and give conclusions in Section 5.

2 Program Decomposition

The following is an informal description of the program decomposition technique in [23]. The key statements that affect pointer aliasing in a C program are assignments involving addresses, pointer values, and structures[1], referred to as *ptr-assignments*. From them, we calculate an equivalence relation (PE relation) on the names in the program. Specifically, the names on the left hand side and right hand side of a ptr-assignment are put into the same equivalence class. For example, for ptr-assignment $*p = q$, $*p$ and q will be in equivalence class E. Moreover, names derived from two names in an equivalence class by applying a dereference ($*$) or a same field operator will be in a same equivalence class; that is, $**p$ and $*q$ will be in class F and $(*p)\rightarrow f$ and $q\rightarrow f$ will be in class H. The PE relation is represented as a directed multi-graph ($G_{\rm PE}$), whose nodes are the equivalence classes and whose edges record the dereference or the field operator which defines the relation between names in connected nodes. For example, an edge labeled $*$ joins class E to class F and an edge labeled f joins F to H. Each name in the program is associated with one node in $G_{\rm PE}$; each ptr-assignment thus is associated with the node to which its constituent names belong. The weakly connected components in $G_{\rm PE}$ partition the ptr-assignments into independent sets in terms of their pointer aliasing effects; each of these sets induces an independent *program segment* including the ptr-assignments and other control statements. Each program segment can be analyzed for pointer aliasing separately.

[1] Note that we consider parameter-argument associations at a call site as assignments.

program name	ICFG nodes	lines of codes	indirect calls	unions	ptr assgn w/ casting
chomp	745	448	0	0	8
loader	1564	1219	0	0	3
stanford	1772	887	0	0	2
pokerd	1896	1241	0	2	3
sim	3034	1439	0	0	18
dinerolII	3477	2961	0	0	2
assembler	3602	2673	0	0	3
smail	3697	3270	0	0	5
archie-client	3856	4680	1	3	29
023.eqntott	4748	3548	9	3	26
rolo	5169	4860	1	0	9
simulator	5575	3733	0	0	3
flex	7377	6970	0	0	75
agrep	8568	3801	0	0	17
bc	8602	7760	19	3	41
zip	9290	7462	1	0	37
bison	9569	7420	0	0	101
022.li	10687	7443	3	0	5
larn	21185	9550	1	0	22
008.espresso	30416	13619	15	0	191
T-W-MC	51628	23788	18	0	273

Figure 1: Test Programs

We have extended the original decomposition algorithm to handle features of C programs such as indirect calls through function pointers, unions and type casting [22] and have used this general decomposition for the experiments described here.

In Figure 1, we present the test programs, which include Unix utilities, Spec benchmarks, and programs used in [10, 19]. The programs are ordered by the number of nodes in their internal representation (ICFG), a better size estimate than lines of code. Also shown are the number of indirect calls, unions, and ptr-assignments with type casting.

3 Experiments: Combined Analysis

Combined analysis chooses an aliasing or a points-to analysis algorithm for the program segment associated with each weakly connected component in $G_{\rm PE}$ and thus derives the aliasing information for the whole program. For our experiments with three combined analyses, we have used the following analysis algorithms:

1. the Landi/Ryder flow-sensitive and context-sensitive aliasing algorithm (FS) [10, 19]
2. a flow-insensitive and context-insensitive aliasing analysis algorithm (FA) [22, 23]
3. a flow-insensitive and context-insensitive points-to[2] analysis algorithm (PT) [22]

[2] Aliasing information can be easily derived from points-to information, although with a potential loss of precision.

Both the FA and PT analyses can handle unions and type casting with some minor restrictions. The major difference between them is that the ptr-assignments are treated *symmetrically* in the FA analysis; that is, if there is a ptr-assignment $lhs = rhs$ in a program, the FA analysis will union lhs and rhs, and thus effectively assume there is also an assignment $rhs = lhs$. We believe this is the main cause of approximation in the FA algorithm and our empirical results confirm this.

In each of our combined analyses, we use only two of the three algorithms. Our experiments involve grouping the weakly connected components of G_{PE} into *two* sets, and assigning one algorithm to program segments associated with components in the first set, and another to those corresponding to components in the second set. We have tried two grouping approaches. In one, we applied the FS analysis to the first set and either the FA or the PT analysis to the second set (*FSandFA* and *FSandPT* analyses); in the other, we applied the FA analysis to the first set and the PT analysis to the second set (*FAandPT* analysis). We also used both the FA and PT analyses on the whole program for comparison.

The aliasing solution obtained is used to determine the locations modified or referenced through each name containing pointer dereferences (e.g., $*p$, $p \rightarrow f$). Specifically, the two problems are:

- *Thru-deref MOD* problem: for each dereferenced name appearing as the left hand side of an assignment, the locations whose values may be modified by this assignment due to aliasing is determined.
- *Thru-deref REF* problem: for each dereferenced name used in a non-lhs context in the program, the locations whose values may be referenced due to aliasing is determined.

We calculate the average number of locations modified or referenced indirectly through dereferenced names for each program as a measurement of the precision of the aliasing solution obtained by the various analyses.

Our implementation is written in C and compiled by *gcc* with *-O2*. The timing results of each analysis are obtained by averaging over ten runs of the analysis on a SUN SPARCstation 20 running Solaris 2.5 operating system.

3.1 Combining a Flow-sensitive and a Flow-insensitive Analyses

The FS analysis is quite precise, but is sometimes slow. The FA and PT analyses are faster, but may not yield as precise a solution as the FS analysis. For the first two combined analyses, we want to assign the FS analysis to those program segments for which it is suitable, and assign either FA or PT analysis to the other segments. The weakly connected components of G_{PE} are grouped such that any component which satisfies *all* of the following conditions is in the *first* set and the other components are in the *second* set.

- There is no cycle in the component.
- There is no location of union type among the names associated with the component.
- There is no type casting in any ptr-assignment associated with the component.

The presence of recursive data structures seems to be a good predictor of where FS analysis will be expensive; our experience shows that FS is slow in dealing with aliases involving recursive data structures. In our program decomposition, recursive data structures are associated with cyclic components. For efficiency, we will not apply FS analysis to program segments for these components. Also, FS analysis handles unions and type casting differently from the FA and PT analyses, which makes it impossible to compare the Thru-deref MOD/REF results of FS with either PT or FA. Therefore, we use FA or PT analysis for program segments with unions or casting.

By restricting the FS analysis to the first set of weakly connected components, we have the following combinations:

- FSandFA analysis: apply the FS analysis to program segments for the first set of components and apply the FA analysis to segments for the second set.
- FSandPT analysis: apply the FS analysis to program segments for the first set of components and apply the PT analysis to segments for the second set.

In Figure 2, we show the Thru-deref MOD/REF results for these two analyses, FA, and PT. Overall, the results of the FSandFA analysis are better than the results of the FA analysis, especially for *pokerd*, *assembler* and *simulator*. In some cases, the results of the two analyses are very close (e.g., *smail*, *bc*, and 008.*espresso*). We believe the FA part of the FSandFA analysis for these programs yields a very approximate aliasing solution and dominates any improvement made by the FS part. The results of the FSandPT analysis are the most precise, but the results of the PT analysis are almost identical. Also, the PT analysis is shown to be consistently much better than the FA analysis; this indicates that the symmetrical treatment of ptr-assignments is the cause of approximation in the FA analysis.

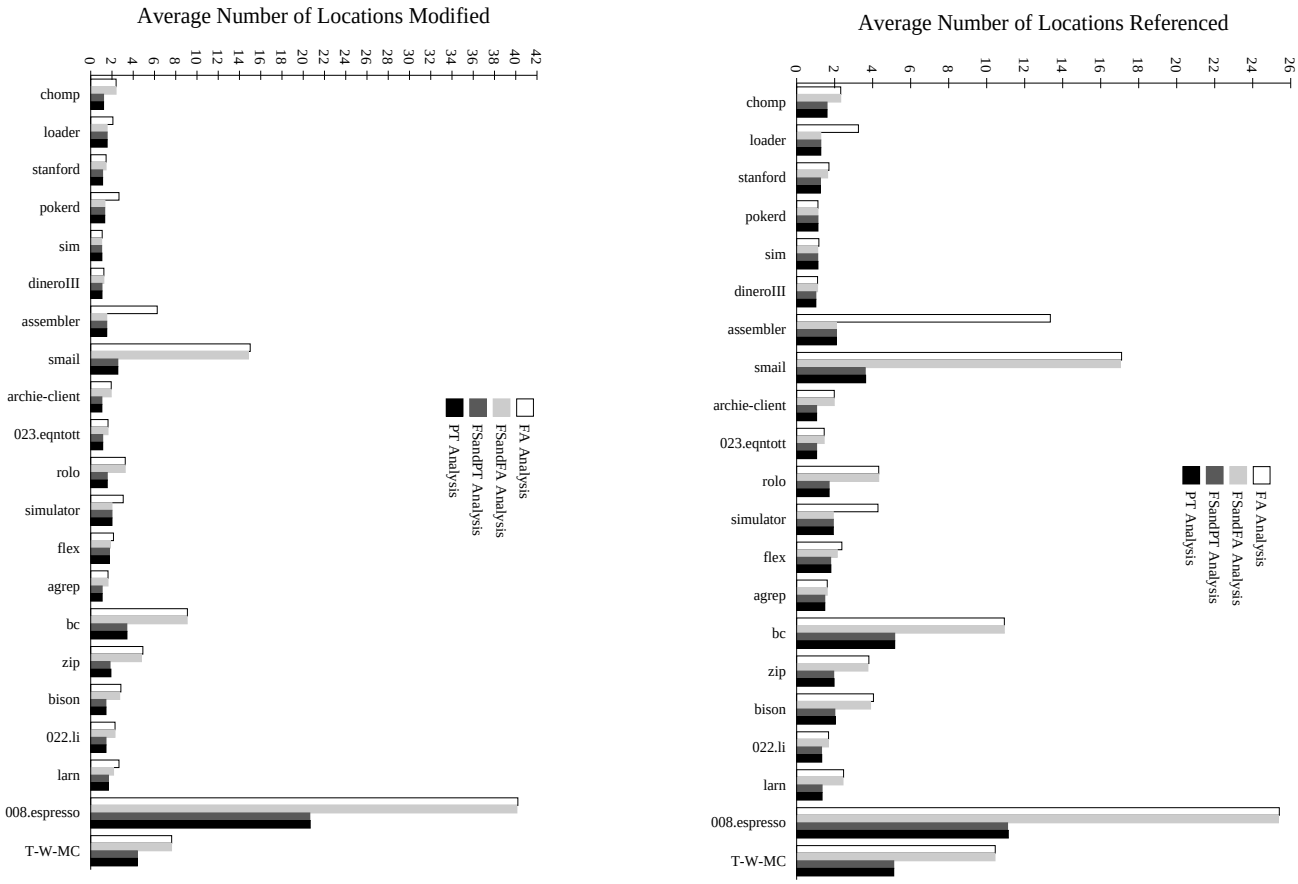

Figure 2: Thru-deref MOD/REF Results for the FSandFA and FSandPT Analyses

To further investigate the impact of the FS part of the two analyses, we present the Thru-deref MOD/REF results for the first sets of components only in Figure 3. A Thru-deref MOD/REF site in a program is associated with the first set of components if the dereferenced name is associated with a component in the set. For the program 022.*li*, there is no such Thru-deref MOD site. The results of the FS analysis are better than those of the FA or PT analysis; that is, the FS analysis yields more precise solutions. However, the PT analysis is very close in terms of the Thru-deref MOD/REF results for most programs; this indicates that flow-sensitivity and context-sensitivity may not play a very important role in aliasing analysis for these programs.

In Figure 4, we present the timings for the two combined analyses, FA, and PT. The timings for the two combined analyses do not include the time required for program decomposition. The FSandFA and FSandPT analyses are slower than either the FA or the PT analysis, especially for large programs. This is caused by the FS part of the analyses. The timings for FSandFA and FSandPT analyses are similar for most programs.

These two analyses allow the FS analysis to be used for parts of a program, even if it may be too slow to be applied to the whole program. However, the Thru-deref MOD/REF results show that the precision of the FS analysis may not be worth the cost because the PT analysis is almost as precise.

3.2 Combining Two Flow-insensitive Analyses

The FA analysis is very fast, but may yield a very approximate aliasing solution because assignments are treated symmetrically. The PT analysis remedies this problem, but is slower than the FA analysis. For the third combined analysis, we want to use the FA analysis as much as possible, but for program segments where we expect it to be overly approximate, we will use the PT analysis.

The grouping of weakly connected components is based on the maximum number of names of the form &*o*

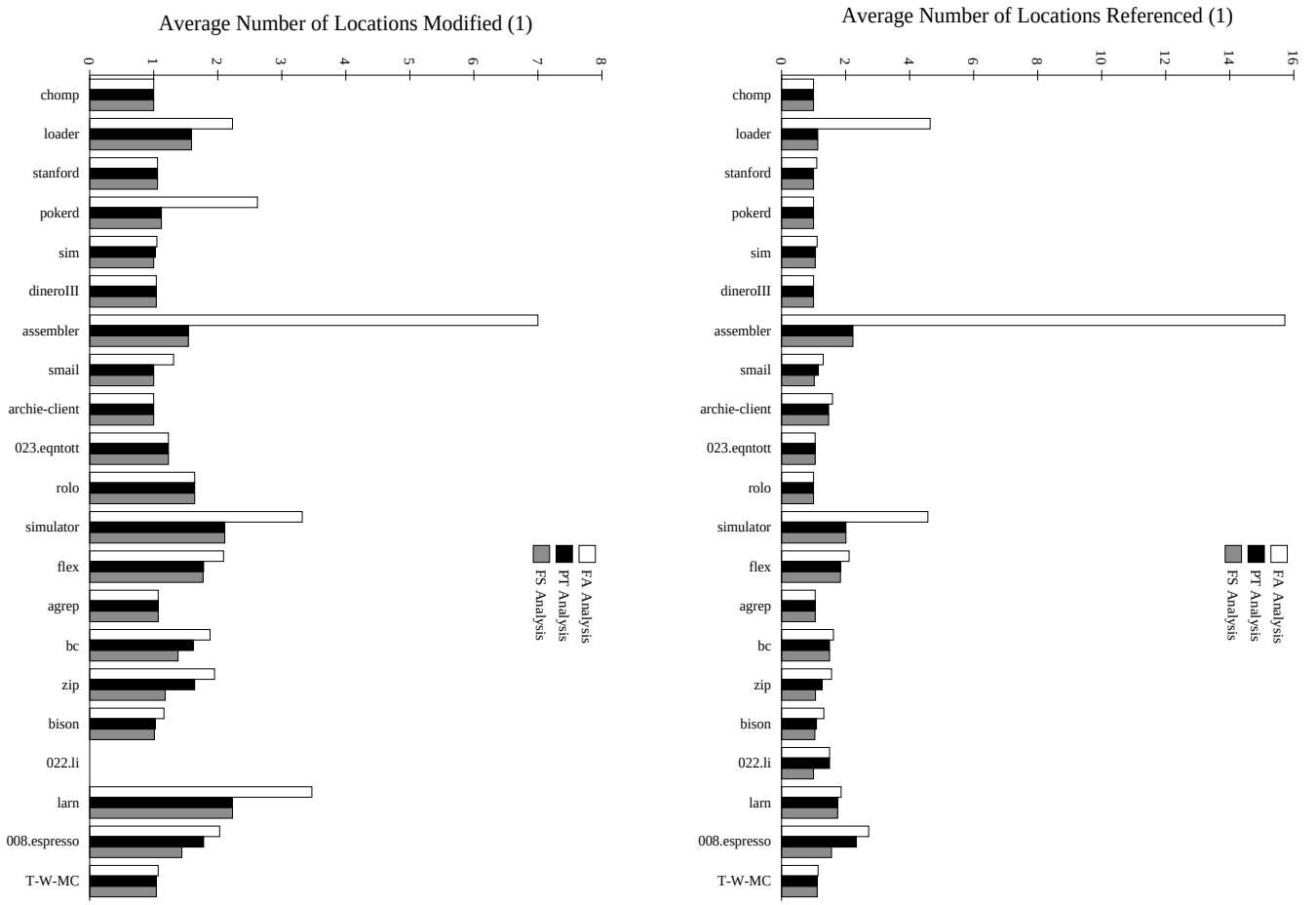

Figure 3: Thru-deref MOD/REF Results for the First Sets of Weakly Connected Components

associated with any equivalence class in a component. If there are more than one name of the form &o in an equivalence class, a name aliased to *any* of the names o will be considered to be aliased to *all* of the names by the FA analysis. Thus, the maximum number of these names associated with any node in a component is a heuristic predictor of the approximation of the FA analysis for the program segment associated with the component.

For this combined analysis, any weakly connected component with the maximum number of names of the form &o less than or equal to a threshold, is put in the *first* set; all other components are placed in the *second* set. We apply the FA analysis to program segments for the first set of components and apply the PT analysis to segments for the second set. The resultant analysis is called *FAandPT*. The idea is that when there are lots of names of the form &o in one equivalence class, the aliasing solution obtained by the FA analysis for the component is likely to be poor and we want to use the PT analysis to get a more precise solution.

We have chosen 5 for the threshold. By varying the threshold value, we can experiment with a spectrum of analyses. The extreme cases are: (1) the threshold is 0 forcing the first set to be empty, which means we use the PT analysis on the whole program; (2) the threshold is sufficient large forcing the second set to be empty, which means we use the FA analysis on the whole program.

In Figure 6, we show the Thru-deref MOD/REF results for this combined analysis, FA, and PT. The FAandPT analysis yields results of similar precision to these of the PT analysis. For those programs, where the FA analysis produces approximate solutions (e.g., *assembler*, *smail*, *bc*, 008.*espresso*), the FAandPT analysis is able to improve the precision. For six programs (*chomp*, *stanford*, *sim*, *dineroIII*, 023.*eqntott*, and *agrep*), the FAandPT analysis is effectively same as the FA analysis because the second sets of components are empty. For these programs, the precision of the FA analysis is comparable to the PT analysis. With the

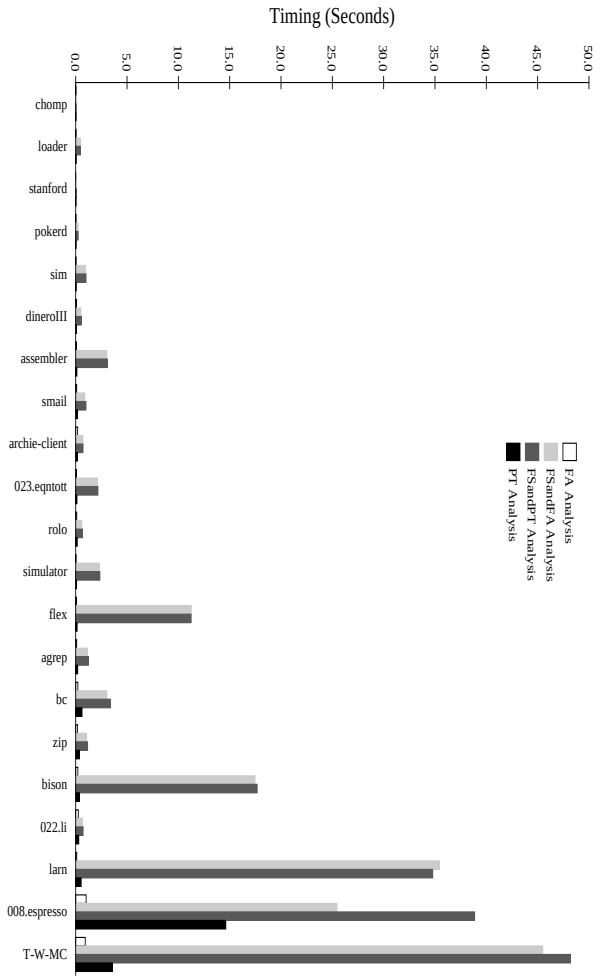

Figure 4: Timings for the FSandFA and FSandPT Analyses

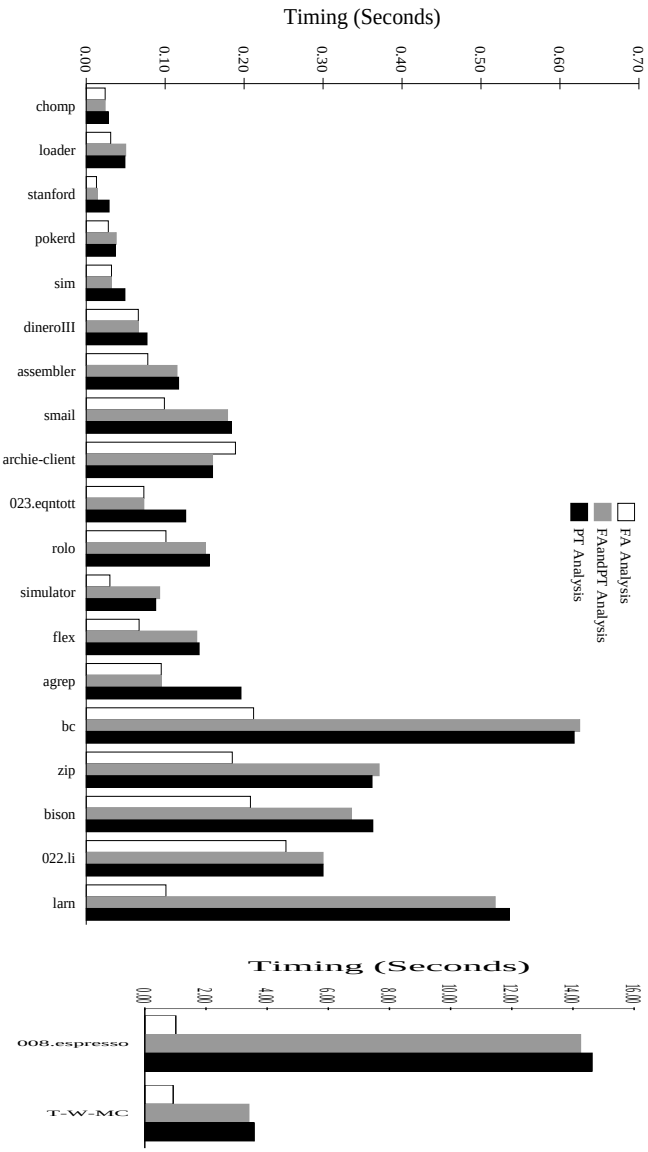

Figure 5: Timings for the FAandPT Analysis

threshold of 5, the first set of components is not empty for any of the test programs, which means the FA part of the combined analysis is always applied for each program.

In Figure 5, we present the timings for this combined analysis, FA, and PT. In general, the FAandPT analysis is slower than the FA analysis and faster than the PT analysis. In many cases, the FAandPT analysis is very close to the PT analysis in cost. This is because the PT part of the FAandPT analysis is the major cost.

These figures indicate that this combined analysis is able to improve the aliasing precision when the FA analysis yields a very approximate solution. The heuristic predictor is effective in identifying program segments, for which the FA analysis is not good. In term of efficiency, the FAandPT analysis is little faster than the PT analysis. We plan to conduct more experiments by varying the threshold to explore the tradeoff of efficiency and precision.

4 Related Work

Many pointer aliasing analysis or points-to analysis algorithms have been proposed in the literature. These algorithms can be classified into flow-sensitive and context-sensitive [4, 5, 6, 7, 10, 12, 21], flow-insensitive and context-insensitive [2, 15, 16, 17, 18, 20], flow-sensitive and context-insensitive [3, 13], or flow-insensitive and context-sensitive [1]. They can also be organized into stack-based aliasing analysis [6], heap-based aliasing analysis [3, 5, 7, 8, 9, 11], or both [4, 10, 13, 17, 18, 21].

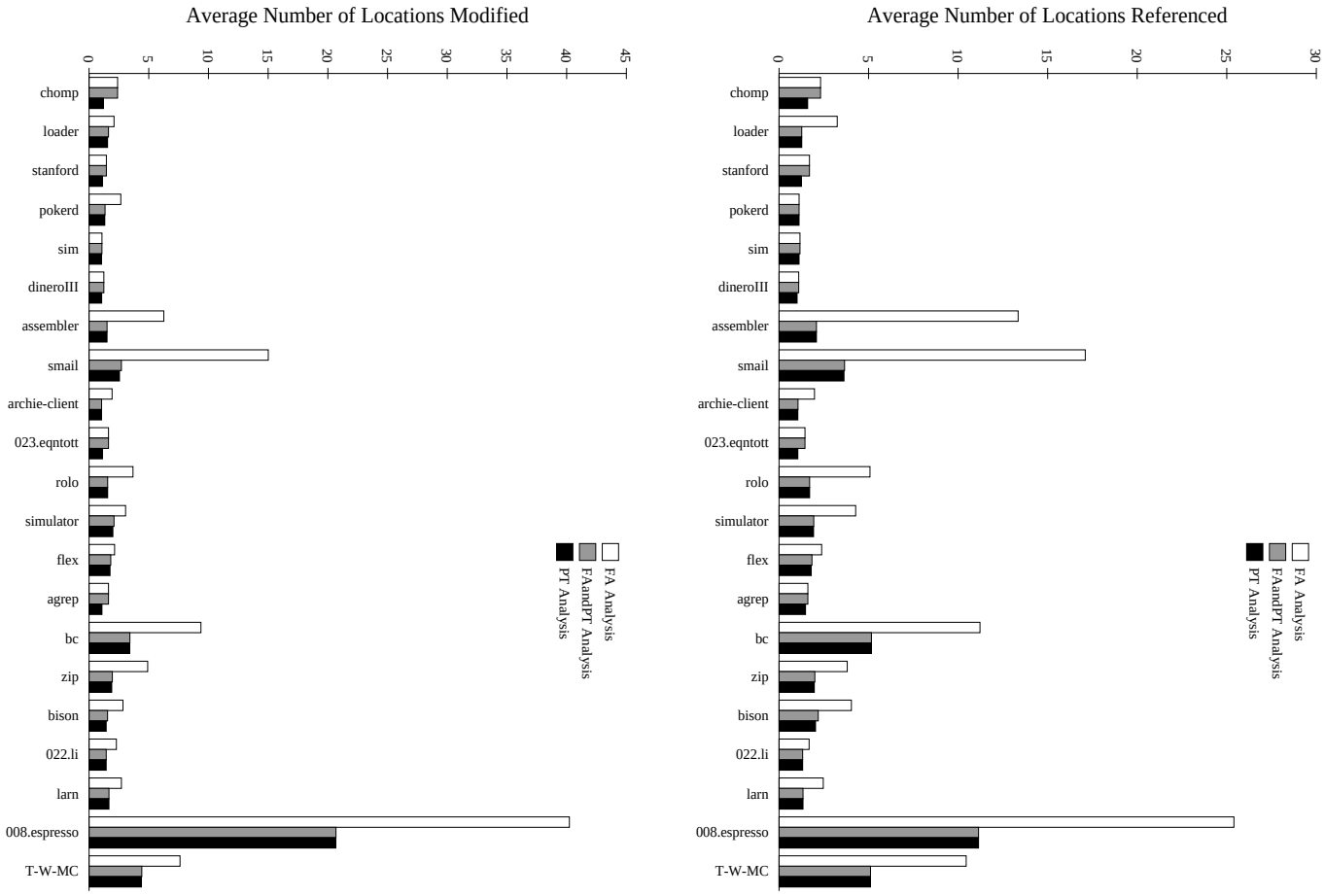

Figure 6: Thru-deref MOD/REF Results for the FAandPT Analysis

Any of these algorithms can be employed for program segments associated with individual weakly connected components in our program decomposition and thus can be used in a combined analysis. Shapiro and Horwitz investigated a family of flow-insensitive and context-insensitive points-to algorithms [16]; we believe the heuristic in our third combined analysis is related to the property they use to distinguish between the degree of approximation in these algorithms.

We first presented a program decomposition technique for pointer aliasing [23]. Ruf [14] suggested using declared types or using type inference techniques such as [17, 18] for program decomposition.

5 Conclusion

Combined analysis uses a program decomposition technique to apply aliasing algorithms that differ in cost and precision to independent segments of a program. We have experimented with three combined analyses using three different aliasing methods – two that are flow-insensitive and context-insensitive (FA and PT), and one that is flow-sensitive and context-sensitive (FS). The experimental results obtained yielded several conclusions:

- The first two combined analyses verify that FS can be applied to segments of a large program, which may be too large to be completely analyzed by FS.
- The use of FS on program segments based on our heuristics, does not yield increased precision unobtainable by other means (e.g., PT). We need to study other means of choosing where to use FS, for example, based on interesting variables designated by a programmer.
- For all the programs, especially the large ones such as *larn, 008.espresso*, and *T-W-MC*, PT analysis, and combinations with PT are much better than FA analysis at relatively little increased cost.

There is evidence that for many data-flow applications, the FA analysis may not yield useful results.

- The heuristic predictor used in the third combined analysis proves more useful than the one used in the first two combined analyses. The FAandPT analysis provides accurate aliasing solutions; although more expensive overall, the precision gained per unit of cost seems worthwhile.

The methodology of combined analysis can be extended to compile-time analyses other than pointer aliasing. There is the potential for achieving scalability in this manner as well as investigating the tradeoff between efficiency and precision. We plan to explore this extension further.

References

[1] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Department of Computer Science, University of Copenhagen, may 1994.

[2] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science*, number No. 892, pages 234–250. Springer-Verlag, 1995. Proceedings from the 7th International Workshop on Languages and Compilers for Parallel Computing.

[3] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[4] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of te 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[5] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[6] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[7] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Conference Record of te 23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, Jan. 1996.

[8] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[9] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[10] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of 1992 ACM Symposium on Programming Language Design and Implementation*, June 1992.

[11] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[12] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. Pointer-induced aliasing: a clarification. *ACM SIGPLAN Notices*, 28(9):67–70, 1993.

[13] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 13–22, june 1995.

[14] Erik Ruf. Partitioning dataflow analyses using types. In *Conference Record of te 24th ACM Symposium on Principles of Programming Languages*, Jan. 1997.

[15] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis*, September 1997.

[16] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of te 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[17] Bjarne Steensgaard. Points-to analysis by type inference of prorams with structures and unions. In *International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150. Springer-Verlag, April 1996.

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of te 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[19] Philip Stocks, Barbara Ryder, William Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceeding of the International Symposium on Software Testing and Analysis*, March 1998. To Appear. Also available as DCS-TR-335.

[20] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the 7th ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

[21] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, june 1995.

[22] Sean Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, Rutgers University, 1998.

[23] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A setp toward practical analyses. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.