Call Graph Extraction in the Presence of Function Pointers

Darren C. Atkinson
Department of Computer Engineering
Santa Clara University
500 El Camino Real
Santa Clara, CA 95053-0566

Abstract *Software engineers need to understand programs in order to effectively maintain them. The call graph, which presents the calling relationships between functions, is a useful representation of a program that can aid understanding. For programs without the use of function pointers, the call graph can be extracted by parsing the program. However, for programs with function pointers, call graph extraction is nontrivial. Many commonly used C programs utilize function pointers for efficiency and ease of implementation. We present different techniques for extracting the call graph in the presence of function pointers and demonstrate our techniques on several commonly available programs. Our results show that unless function pointers are taken into account, the call graphs of these programs are erroneously small. We also show that performing a simple, conservative pointer analysis yields graphs that are too large to be useful. However, both filtering of the points-to sets and the use of run-time pointer data can be used to obtain a closer approximation to the true graph.*

Keywords: Program Analysis, Pointer Analysis, Software Maintenance

1 Introduction

Software engineering is the complex task of specifying, designing, implementing, testing, and maintaining programs and their associated documentation. However, the majority of time and effort is spent on program maintenance [1]. Therefore, to significantly reduce the cost of software, we need to reduce the cost of maintenance.

Software maintenance can have a variety of forms. A programmer may need to incorporate an enhancement requested by the customer. The software may need to be adapted for a new architecture or platform. Defects in the design or implementation (i.e., bugs) may need to be corrected. Finally, the engineer may simply wish to restructure the system, improving its design and organization, to ease incorporation of future changes.

For all of these applications, the engineer needs to thoroughly *understand* the system in order to correctly maintain it. A lack of understanding could lead to additional defects being introduced. Ideally, requirements, specifications, and design documentation would be available for the engineer to use. Unfortunately, these documents often do not exist or are out of date. Consequently, the engineer is left with the tedious and error-prone task of examining the source code by hand in order to understand the system.

To assist the software engineer in understanding a system, automated tools such as program slicing tools and invariant checkers have been proposed as a solution. For example, a backward program slicer [2] computes the set of statements that may have affected the value of a given variable, which may aid programmers during debugging. As another example, an invariant checker infers facts about the state of the program and checks those facts against assertions provided by the programmer [3]. Such tools analyze the program source, computing information about the program, and present that information in a way that is most useful to the tool user.

An extremely useful analysis is the extraction of the *call graph* of a program. The call graph illustrates the relationship between calling functions or procedures and the function or procedures that they

call. Not only is the call graph useful alone, but the proper determination of calling relationships is a prerequisite for other analyses as well. For example, determining how a proposed change to one module of the system will affect the rest of the system requires the knowledge of which functions call which other functions. Accurate call graph extraction is also useful in software testing, since one common goal of testing is to ensure that each function is executed on at least one test run [4].

Unfortunately, the problem of call graph extraction is nontrivial. Modern programming languages allow functions to be used in ways other than just in traditional call expressions. For example, most languages allow functions to be passed as parameters to other functions. Languages such as C and C++ allow the addresses of functions to be taken and used as pointers. Many significant, large scale programs rely heavily on function pointers to implement dispatch tables (tables in which the key is an integer value designating an operation and the corresponding value is the address of a function that performs that operation) or object-oriented dispatch. In effect, functions and procedures are used more as "first-class" objects such as integers or pointers than solely through simple function and procedure calls. These uses are necessary for efficiency and often greatly simplify design. However, they complicate the construction of the program's call graph since the calling relationships are not apparent simply by examining the lexical and syntactic structure of the program.

Most static (i.e., compile-time) program analyses account for these problems by first computing the sets of pointer values (points-to sets) for functions, and then using the pointer values to construct an accurate call graph. Although precise algorithms exist for determining points-to sets, they typically have running times that are not acceptable for use on larger programs (e.g., programs with at least 50,000 lines of code). For example, many algorithms have $O(n^2)$ or $O(n^3)$ running times, where n is the number of lines of code [5]. Therefore, less expensive and less precise analyses are often used that trade precision for performance. These analyses are typically not fully flow-sensitive or context-sensitive [6, 7]. For example, efficient, near-linear time points-to algorithms are well known [8, 9] and are often used in practice. Unfortunately, the results of these analyses are often less than desirable. For example, the resulting data may indicate that a call made through a function pointer could possibly call any function in the system whose address is taken. This leads to many "false" edges in the call graph, hindering program understanding.

Previous work has focused on filtering the points-to sets for function pointers in an attempt to recover the true pointer relationships [10]. Such an approach may be done automatically by the software tool using heuristics or may be done by hand by the tool user. For example, one heuristic approach is to use the function prototypes (signatures) to infer the set of legal call targets from the set of conservatively computed targets. A more *ad hoc* approach is for the targets of function pointers to be specified explicitly using a compact, flexible notation such as regular expressions.

More recent work has focused on using dynamic (run-time) information as a replacement for static (compile-time) information in program analyses [11]. Dynamic data has the advantage that it captures the true calling relationships, rather than some approximation of the relationships. The principal disadvantage of using dynamic data is that it may be optimistic, since it may not capture every possible calling sequence. That is, the dynamic data will always be a subset of the true set, and may therefore fail to indicate that a relationship exists when it in fact does. However, given a large number of test runs, the dynamic data can be made close to optimal. Furthermore, since dynamic data is a subset of the true set, it can be used to establish a lower bound on the size of the call graph.

In this paper, we perform an empirical evaluation of the effectiveness of several techniques for constructing the call graphs of C programs with function pointers. Our results show that if function pointers are ignored, the call graph is often erroneously too small. However, if only a simple, although efficient, flow-insensitive and context-insensitive pointer analysis is performed to compute the points-to sets for functions, then the resulting graph is too large to be generally useful. Through the use of heuristics and dynamic data, a more faithful approximation of the true call graph

can be obtained without placing an inordinate burden on the tool user.

2 Extraction Techniques

Edges in the extracted call graph may be classified as either direct edges or pointer edges. The direct edges can be computed using a simple syntactic analysis of the program text. The compute the direct edges, we used the `cgraph` utility (based on the standard UNIX `cflow` utility), available as part of the ICARIA tool set. To compute the pointer edges, we modified the `sprite` program slicing tool [10, 11, 12] to display the call graph without performing a program slice. `Sprite` uses Steensgaard's flow-insensitive, context-insensitive points-to analysis, which models storage as equivalence classes of locations [9]. We modified this standard points-to analysis in several ways.

Function prototype filtering: In this modification, we use type information to reduce the sizes of the points-to sets. In particular, the user may specify whether the program uses weakly (old-style "K&R" C) or strongly ANSI-compliant function prototypes. The filtering rules for both levels of checking are shown in Figure 1. Function prototypes provide additional typing information for static semantic checking by ensuring that the types and number of formal and actual arguments agree. After retrieving the points-to set for a function pointer reference, the prototypes of the resultant set of function definitions are compared against the prototype implied by the function call. The prototypes are computed from the actual function definition and the function call since the program may be ANSI-compliant, but not be written using explicit ANSI-style prototypes. Enabling this option does not affect the construction of the points-to classes, but rather filters the classes based on the calling statement, reducing the number of functions that may be called for a given function call expression. This option is unsafe if the program does not use function pointers in an ANSI-compliant manner.

Lexical specification of call patterns: In this modification, we specify a lexical pattern (i.e., regular expression) for filtering the static points-to data. Both the function call expression and the set of called functions can be specified as regular expressions. For example, for the `find` application, we specified a pattern indicating that any call through a function pointer named "`parse_function`" resolved to any function whose name began with "`parse_`". For some programs such as `find`, lexical specification of call patterns is simple due to naming conventions. However, for some programs, this type of specification is not practical without a deep understanding of the program's behavior. Use of this option is, in the general case, unsafe, and requires more knowledge of the code than simple prototype filtering.

Use of dynamic data: In this case, we instrumented the programs in our test suite, inserting code that captures the run-time addresses of pointer targets. The instrumented application is compiled and then executed on some representative inputs. Upon termination, the instrumentation code will save the set of locations that were referenced at each call site, thereby producing a dynamic points-to set for each function pointer use. This data can then be used as a replacement for the static points-to data computed by Steensgaard's algorithm. As previously mentioned, the dynamic function pointer data may be optimistic, in that it may be a subset of the true data, and may therefore fail to indicate a calling relationship where one in fact does exist.

3 Results

Table 1 lists the programs we used in our experiments. We used programs from the SPEC 2000 benchmark suite along with programs used by other researchers in call graph extraction and program slicing experiments [12, 13], many of which are commonly used UNIX utilities. Table 1 also shows the number of edges in the call graphs extracted using the various techniques. An empty entry indicates either that the dynamic data was unavailable due to the program's size or that specification of the lexical patterns was impractical.

	Strong checking	Weak checking
arguments	number of actuals must match number of formals	number of actuals must be at least number of formals
specifiers	one is assignable to other, structure tags must match	one is assignable to other, structure tags need not match
declarators	must match exactly, unless one is pointer to void and other is pointer	match at outermost level only, unless one is pointer and other is integer
qualifiers	ignored	ignored

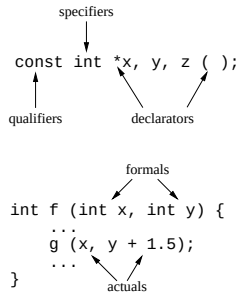

Figure 1: Prototype filtering rules for both strong and weak prototype filtering.

For all applications, we do not know the size of the true call graph. Rather, we are using the experimental data to try to obtain an estimate of the true size. However, we do know that (1) the call graph extracted using dynamic pointer data is a subset of the true call graph, and (2) the programs in our test suite are mostly ANSI-compliant, at least with respect to function calls. Therefore, the number of edges in the true call graph most likely lies somewhere between the numbers presented in the "dynamic data" and "strong prototypes" columns.

We see for `gap` and `mesa` that if we ignore function pointers entirely, then the call graph extracted is too small by several orders of magnitude. On the other hand, we see that the call graph extracted using Steensgaard's pointer analysis without any filtering performed on the points-to sets is almost certainly overly conservative and in many cases is too large to be generally useful.

For most programs, it was too difficult to specify lexical patterns for the function calls, given the complexity of the programs and our unfamiliarity with them. Furthermore, our attempt to specify the call patterns for `mesa` generated unsound information since the resulting call graph has fewer edges than the call graph extracted using dynamic function pointer data. However, for `find` and `burlap`, we were able to specify patterns that yielded good results. For `find`, the number of edges is equal as the number of edges obtained using strong prototype filtering. For `burlap`, the results are slightly better than those obtained with prototype filtering and were verified by hand to be fully precise. Therefore, for certain applications, lexical specification of the function calls is possible and yields better results than what could be otherwise obtained.

4 Conclusion

The call graph is a useful representation of a program that can greatly aid understanding. In this paper, we presented different techniques for performing call graph extraction in the presence of function pointers. We showed that simply ignoring calls through function pointers can result in a call graph that is erroneously too small. We also showed that using a conservative but efficient static points-to analysis can result in a call graph containing many "false" edges and is too large to be generally useful. To overcome these problems, we used a variety of techniques such as filtering of the points-to sets and the incorporation of dynamic data. Using these techniques a more accurate call graph can be constructed. Future work is required to determine how the accuracy of the call graph affects various program analyses.

References

[1] M. Hanna. Maintenance burden begging for a remedy. *Datamation*, pages 53–63, April 1993.

Table 1: Number of edges in the call graphs extracted using various techniques.

	Lines of code	**Syntax analysis**	**Dynamic pointer data**	**Lexically filtered**	**Strong prototypes**	**Weak prototypes**	**Unmodified ptr. analysis**
diff	11,755	348	350	—	360	360	360
grep	13,084	371	377	—	393	440	488
find	13,122	388	424	710	710	1,010	1,059
ammp	13,263	590	606	—	891	891	891
less	18,305	1,131	1,132	—	1,157	1,157	1,157
gap	49,482	1,956	17,336	—	97,216	152,597	237,499
mesa	49,701	1,919	11,670	7,832	22,976	120,309	245,574
burlap	49,845	2,441	2,500	2,759	2,878	4,416	5,785
vortex	52,633	4,654	4,695	—	4,660	4,746	4,886
gcc	205,743	10,189	—	—	11,521	15,365	17,658

[2] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[3] G. N. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proceedings of the 4th ACM Symposium on the Foundations of Software Engineering*, pages 93–105, San Francisco, CA, November 1996.

[4] G. J. Myers. *The Art of Software Testing*. Wiley, New York, NY, 1979.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Ph.D. dissertation, University of Copenhagen, DIKU, May 1994.

[6] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[7] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995.

[8] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, Vancouver, BC, June 2000.

[9] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.

[10] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the 6th ACM International Symposium on the Foundations of Software Engineering*, pages 46–55, Lake Buena Vista, FL, November 1998.

[11] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing with dynamic points-to data. School of Engineering Technical Report COEN-2002-03-15, Santa Clara University, Department of Computer Engineering, March 2002.

[12] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the 2001 International Conference on Software Maintenance*, pages 52–61, Florence, Italy, November 2001.

[13] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*, pages 90–99, Berlin, Germany, March 1996.