UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Design and Implementation of Practical and Task-Oriented Whole-Program Analysis Tools

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

Darren C. Atkinson

Committee in charge:

Professor William G. Griswold, Chairperson
Professor Jeanne Ferrante
Professor Larry Carter
Professor Philip Gill
Professor Debra Richardson

1999

The dissertation of Darren C. Atkinson is approved, and it is acceptable in quality and form for publication on microfilm:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

1999

To my friends and family

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Bill Griswold, for his time and patience over the past six years. His guidance, support, and practical knowledge were always helpful and encouraging.

The list of fellow researchers I should thank is long. During my time at UCSD, I've seen many students come and go. I'd like to thank Robert Bowdidge and David Morgenthaler for being such good coworkers during the first several years of my tenure as a graduate student. The many lunches we had together were always relaxing, even when Bill tagged along. Collin McCurdy helped me immensely by retargeting my tools to the C programming language, saving me a lot of time. Whenever I had to change or enhance his implementation, I gained a better appreciation for just how much work he saved me. Morison Chen, Andy Gray, Walter Korman, and Jim Hayes are just a few of the other students with whom I've had the opportunity to share my ideas.

I would also like to thank Rick Ord and Cindy Paloma for giving me a way to earn some real money, gain practical experience, have fun, and still do my research. My time as a member of the Computer Systems Group has always been enjoyable, even if it did get a bit hectic at times. Glenn Little, Steve Hopper, Dave Wargo, and David Hutches have always been great coworkers. Everyone was always so understanding of my need to finish my research. I thank them all for their patience.

My family has always been so supportive during my many years of school. Even when it seemed like I would never finish, they encouraged and supported me. For that, I will always be thankful.

Finally, I'd like to thank Van Nguyen for giving me the greatest gift of all. Her patience, understanding, caring, and support have been the greatest things to come from my years at UCSD. Without her, all of my accomplishments would mean nothing.

The text of this disseration, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998] and [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of these publications directed and supervised the research that forms the basis for this dissertation.

VITA

November 21, 1968	Born, San Diego, California
1991	B.S., University of California, San Diego
1994	M.S., University of California, San Diego
1999	Ph.D., University of California, San Diego

PUBLICATIONS

D. C. Atkinson and W. G. Griswold, “Effective Whole-Program Analysis in the Presence of Pointers”, 6th ACM Symposium on Foundations of Software Engineering, Lake Buena Vista, FL, 1998.

D. C. Atkinson and W. G. Griswold, “The Design of Whole-Program Analysis Tools”, 18th International Conference on Software Engineering, Berlin, Germany, 1996.

W. G. Griswold, D. C. Atkinson, and C. McCurdy, “Fast, Flexible Syntactic Pattern Matching and Processing”, 4th Workshop on Program Comprehension, Berlin, Germany, March 1996.

W. G. Griswold and D. C. Atkinson, “Managing the Design Trade-Offs for a Program Understanding and Transformation Tool”, Journal of Systems and Software, Vol. 30, 1995.

W. G. Griswold and D. C. Atkinson, “A Syntax-Directed Tool for Program Understanding and Transformation”, 4th Systems Reengineering Technology Workshop, Monterey, CA, 1994.

J. I. Gobat and D. C. Atkinson, “The FElt System: User’s Guide and Reference Manual”, Computer Science Technical Report CS94-376, University of California, San Diego, 1994.

FIELDS OF STUDY

Major Field: Computer Science

Studies in Software Engineering.

Professor William G. Griswold

ABSTRACT OF THE DISSERTATION

The Design and Implementation of Practical and
Task-Oriented Whole-Program Analysis Tools

by

Darren C. Atkinson
Doctor of Philosophy in Computer Science
University of California, San Diego, 1999
Professor William G. Griswold, Chair

Building efficient tools for understanding large software systems is difficult. Many existing program understanding tools build control-flow and data-flow representations of the program *a priori*, and therefore require prohibitive space and time when analyzing large systems.

By customizing the tool to the task and analysis being performed, significant time and space can be saved. Since much of these representations may be unused during an analysis, we construct them on demand, not in advance. Furthermore, some representations may be used infrequently during an analysis. We discard these and recompute them as needed, reducing the overall space required. Finally, we permit the user to selectively trade-off time for precision and to customize the termination of these costly analyses to provide finer user control, thereby improving the flexibility of the tool. We revised the traditional software architecture for compilers to provide these features without unnecessarily complicating the analyses themselves.

These solutions improve the effectiveness of whole-program analysis tools by making the analysis more practical (i.e., faster and scalable) and task-oriented. However, the use of pointers in most modern programming languages introduces additional problems. The lessons of adaptability and flexibility must be applied to points-to analysis if our approach is to remain effective on large systems.

First, we use a fast, flow-insensitive, points-to analysis before traditional data-flow analysis. Second, we allow the user to parameterize the points-to analysis so that the resulting data-flow information more closely matches the actual program behavior. Such information cannot easily be obtained by the tool or might otherwise be deemed unsafe. Finally, we present data-flow equations for dealing with pointers to local variables in recursive programs. These equations allow the user to select an arbitrary amount of calling context in order to better trade performance for precision.

To validate our techniques, we constructed program slicers for the MUMPS and C programming languages. We present empirical results using our slicing tools on the Comprehensive Health Care System (CHCS), a million-line hospital management system written in MUMPS, and on several C programs with aggressive pointer usage. The results indicate that cost-effective analysis of large programs with pointers is feasible using our techniques.

Chapter 1

Introduction

1.1 Motivation

Today, software development is market-driven. Developers rush their products to market in order to meet customer demands, to gain a greater share of the market, and to provide new features that they hope will become *de facto* standards. Perhaps the most apparent example of this strategy is the ongoing "Internet browser war" between the Microsoft and Netscape corporations. Unfortunately, this philosophy places pressure on programmers and managers to develop software fast and to reach the market before their competitors, rather than to develop software that is extensible, well-designed, and thoroughly tested.

Although the results of such a design philosophy can be beneficial in the short term, they are often detrimental in the long term, when maintenance costs become the dominant cost of the software life-cycle. Therefore, significantly reducing the cost of these software systems entails reducing the cost of maintenance.

In the course of maintenance, software systems may be debugged, restructured, extended, or perhaps completely rewritten from scratch with a better design in mind. Software designers and maintainers need to understand their systems in order to perform any of these tasks correctly and if the successful development and maintenance of their systems is to continue.

Unfortunately, large software systems are difficult to understand, in part because of their age. Some of these systems were not implemented using modern programming techniques that can help reduce the complexity of a system, such as information hiding [Parnas, 1972]. Additionally, many modifications to these systems were not anticipated in the original design, resulting in global modifications being made to incorporate the change. A global change distributes design information that is preferably hidden within a single module in order to ease future changes dependent on that information. As a result, the structure of the system is degraded, and maintenance costs are increased, since the programmer needs global, rather than local, knowledge to successfully implement a desired change. Finally, these large systems have been evolved in this fashion over several years, with modification after modification being layered upon the original implementation by several generations of programmers. The resulting complexity may be exponential in the number of changes made to the system [Lehman and Belady, 1985].

Large systems are also often written in an aggressive programming style and use sophisticated language constructs such as function pointers. The use of these constructs is typically necessary to achieve good performance or to ease implementation. However, their use hinders program understanding. For example, function pointers are commonly used to implement dispatch tables. However, if function pointers are used, then a call-graph of a program cannot be inferred without first determining the side-effects due to pointers.

Many of these large systems are still in use today, such as the Comprehensive Health Care System (CHCS), a 1,000,000 line hospital management system, written in the MUMPS programming language [Lewkowicz, 1989] and currently maintained by Science Applications International Corporation (SAIC). Other large, well-known systems such as the GNU C compiler (GCC) and Emacs editor, although an order of magnitude smaller, exhibit many of the same problems.

Because of their complexity, large systems can benefit from automated support for program understanding. Several automated semantic techniques have been de-

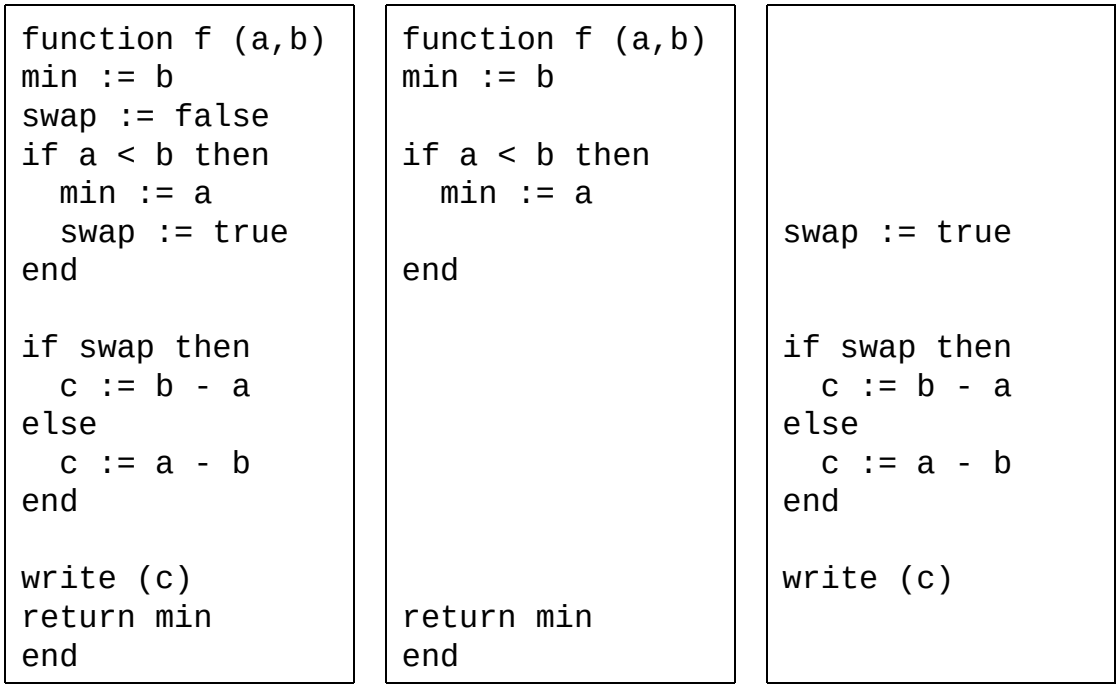


Figure 1.1: An example program and slices: (a) an example function, (b) a backward slice from `min` at the return statement, (c) a forward slice from the assignment to `swap`.

---

veloped for understanding software. For instance, a program slicer computes the set of statements in a program that *may affect* the value of a programmer-specified variable [Weiser, 1984]. This type of program slicing is known as *backward slicing*, since the slicing tool traces the flow of values backward through the control-flow of the program, and is useful during debugging. In contrast, *forward slicing* traces the flow of values forward through the program, computing the set of statements that *may be affected by* the value of a programmer-specified variable. Forward slicing can be used by a programmer to predict the effects of a proposed change. Figure 1.1 shows an example program written in a high-level language, along with an example backward and forward program slice. A programmer or designer can better understand the system by using these tools to answer their queries about the system, but only if they run in an economical amount of time and space.

Other examples of semantic tools include assertion and invariant checkers. A static assertion checker such as ASPECT checks the consistency of a program's data-flow and control-flow characteristics against declared computational dependencies [Jackson, 1991]. An invariant checker such as FLAVERS infers facts about the state of the program and checks those facts against assertions provided by the programmer [Dwyer and Clarke, 1994].

In order to analyze a program, such tools construct control-flow and data-flow representations of the program, similar to those used by an optimizing compiler. One such representation is the program dependence graph (PDG) [Ferrante et al., 1987], in which nodes denote operations and edges denote dependencies between operations. Program slicing using the PDG is simple and algorithmically efficient, once the PDG has been constructed [Ottenstein and Ottenstein, 1984]. A similar representation is the value dependence graph (VDG) [Weise et al., 1994], in which nodes denote operations and edges denote demands for stores (mappings from variables to values) by operations. More traditional representations include the abstract syntax tree (AST), control-flow graph (CFG), dominator trees, and def-use chains [Aho et al., 1986].

However, unlike compilation, program understanding tasks are interactive and an analysis such as slicing is often applied iteratively to answer a programmer's question about the program. For example, a programmer may need to perform several slices with different slicing criteria, incorporating the knowledge gained from previous slices (e.g., which functions were included in the slice), in order to successfully answer a query. Thus, a whole-program analysis tool must perform analyses quickly in order to answer effectively many of the questions posed by programmers and designers. Two problems arise when applying traditional compiler techniques to the construction of whole-program analysis tools.

First, unlike an optimizing compiler, which typically analyzes a few procedures at a time, these tools analyze the entire program in order to provide the programmer or designer with a global view of the system [Barth, 1978]. Consequently, both the running time and space required for many traditional interprocedural compiler algo-

	representation	comments on construction	time	space
(1)	source	1 million lines	N/A	40
(2)	AST	construction for entire program; 18 million nodes	13.1	414
(3)	AST	construction on a per-routine basis; 25% greater CPU utilization than above	6.9	30
(4)	CFG - symbolic	preserving routine's AST after use; exhausts virtual memory capacity	unknown	≈ 800
(5)	CFG - symbolic	discarding routine's AST after use; 6.5 million three-address statements	17.9	397
(6)	CFG - complete	symbolic labels replaced by graph edges; 39% CPU utilization	27.5	397

Table 1.1: Statistics for constructing various representations of CHCS. Time is given in minutes and space in megabytes.

rithms may be prohibitive for a large program, especially in an interactive context such as software maintenance. For example, the size of a PDG can be quadratic or greater in the size of the program (depending on the handling of pointers), and therefore a PDG for a large system may exceed the virtual memory capacity of a typical desktop machine. Even the cost of constructing simple, linear-space representations such as an AST can be prohibitive. As the second item in Table 1.1 illustrates, although the size of an AST is linear in the size of the program, the space consumed by an AST constructed for CHCS, 414 MB, exceeds the capacity of the main memory of the machine, even though care was exercised in its design [Griswold and Atkinson, 1995].[1] In such cases, the time required to first construct the representation and then later retrieve it for actual use after it has been paged out of main memory may be unacceptable. The space required for both an AST and a CFG can exceed the virtual memory capacity of the machine (Table 1.1, item 4). Furthermore, the additional iteration over the three-address statements of the CFG that resolves symbolic references to labels into graph edges (Table 1.1, item 6) re-

[1] All statistics for CHCS were gathered on a SparcStation 10 Model 61 with 160 MB of physical memory and 450 MB of swap space. All statistics for other programs were gathered on a 200 MHz Sun UltraSparc 2 with 192 MB of physical memory and 1 GB of swap space. Experiments were performed with the machine otherwise idle.

quires an additional 9.6 minutes, illustrating poor performance due to heavy use of the slower parts of the memory hierarchy. Based on the per-routine cost of constructing the AST (Table 1.1, item 3), we estimate that the actual cost of constructing the CFG is 11 minutes, only 1.4 minutes longer than this additional iteration.

The second problem is that a program understanding tool must be able to answer a wide variety of questions about a program. Since program understanding tools are difficult to construct, general tools are typically built to amortize the high cost of construction. However, because program analysis algorithms are complex, it is not always feasible for the tool user to program a new algorithm to answer a specific question. Consequently, program understanding tools tend to provide a small set of general analysis algorithms that together can answer a wide variety of questions. Unfortunately, although these algorithms can be used to answer most questions, their generality can result in an unacceptably long running time and the gathering and reporting of extraneous information. For example, if a tool user desires to know only if a procedure *P* is included in a forward slice (e.g., if the procedure may be affected by a proposed change) then the entire slice may not need to be computed. In particular, a statement in *P* may appear in the slice during the first few iterations of the analysis. If so, then computing the entire slice is unnecessary, saving not only computation time but also time spent by the tool user interpreting the results.

1.2 Approach: Adaptability and Flexibility

One might argue that simply buying more memory, disk, and a faster processor could solve these problems, but this solution is not cost effective. The size of many modern systems is several times greater than the million lines of CHCS and is always growing. A project may also have many programmers requiring such resources to perform analyses. The real problem is waste of computational resources, not lack of them.

For simplicity, program representations are often constructed in their entirety *in batch*, or once before an analysis is begun. The representations are typically never

discarded once constructed. Should changes to the representations be necessary (e.g., the program has been transformed by the analysis), the representations are either reconstructed in batch or are incrementally updated, if the changes are small and structured.

However, the per-routine construction of the representations shown in Table 1.1, which requires much less space and time, and exhibits much better CPU utilization, suggests that the prohibitive computational costs are largely due to the computation and movement of program representations within the virtual memory hierarchy, regardless of the analysis algorithm to be subsequently applied to the representations. The underlying tool infrastructure *fails to adapt* to the nature of the analysis being performed and the program being analyzed.

Our first goal, then, is that the cost of an analysis be a function of the size of the relevant portions of the program, rather than of the size of the entire program. For example, the cost of computing a program slice should be a function of the number of statements in the slice. To meet this goal, the execution of the analysis algorithm needs to drive the construction of the representations that it accesses. In particular, we propose that a whole-program analysis tool:

- Construct *all* program representations on demand, rather than *a priori*: Demand-driven construction reduces the space and time required since portions of the program that are irrelevant to the analysis are ignored. Current approaches [Choi et al., 1991; Horwitz et al., 1995] only demand-derive portions of the analysis.

- Discard and recompute infrequently used representations that are large but relatively inexpensive to compute: Many representations such as the AST are infrequently used but can exhaust virtual memory if retained. The recomputation cost for these representations may be no worse than the cost of moving them to the slower portions of the memory hierarchy and later retrieving them.

- Persistently cache frequently used representations that are small but relatively expensive to compute: Resolving interprocedural labels in the CFG is expensive and impractical to demand incrementally, but requires little space. Time can be saved

space required	time to construct: fast	time to construct: slow
small	*build and retain* (points-to and data-flow sets)	*persistently retain* (call-graph, function pointers)
large	*discard if infrequently used* (AST, CFG)	*rework approach to improve time / space* (PDG, VDG)

Figure 1.2: Decision space showing how a program representation should be handled.

by saving this information on disk and only recomputing it when the analyzed software is changed.

In general, the properties of a representation such as the space occupied, cost to construct in its entirety, cost to demand in parts, and frequency of access determine whether it should be discarded, retained during execution, or retained persistently across uses of the tool, as shown in Figure 1.2.

The second source of waste is performing an analysis that is more general than the tool user requires because of *lack of flexibility* in these tools. Our second goal, then, is that the tool user should be able to customize the parameters of the analysis—possibly saving computation time—to better match the tool user's needs. Since our tool is designed for interactive program understanding rather than for batch compilation, we can take advantage of information provided by the tool user. For example, if the tool user only wishes to know whether a certain procedure is in a slice, then the analysis should terminate when this fact becomes known. In particular, we propose that a whole-program analysis tool:

- Allow the user to control the precision of the analysis algorithm: The user can provide additional information based on external factors such as the desired precision of the result, urgency, and system load. For example, by reducing precision the tool user can reduce the time of an iteration of an iterative analysis and thus receive an answer more quickly.

- Allow the user to customize the termination criterion for a particular analysis: For example, the number of iterations required can be substantially reduced because iterative algorithms tend to have an initial rapid convergence and so might discover the needed information quickly.

These new features risk complicating analysis algorithms that are already complicated. Consequently, we have a designed a software architecture [Garlan and Shaw, 1993; Perry and Wolf, 1992] that is event-based and exploits the structure of interprocedural analysis to support demand-deriving and discarding data without complicating these algorithms [Atkinson and Griswold, 1996].

1.3 Issues Due to Pointers

The use of pointers in most modern programming languages complicates our approach. The lessons of adaptability and flexibility must be applied to points-to and alias analyses if our approach is to remain effective on large systems. If an analysis is implemented naively, many of the benefits that we have discussed will be reduced or negated. In particular, three problems arise in dealing with pointers in large systems.

First, the use of pointers negates the performance benefits of demand-driven techniques [Atkinson and Griswold, 1996; Duesterwald et al., 1995; Horwitz et al., 1995] since determining the memory locations possibly referenced through a pointer typically requires a global analysis over the program. For example in the C programming language [Kernighan and Ritchie, 1988], all files must be analyzed to account for the use of pointers in initializers for static variables, regardless of whether a file contains a function that might be reachable during subsequent data-flow analysis. All three of our example programs in Table 1.2 use function pointers in static variables to implement late binding or dispatch tables.

Second, in a flexible and performance-oriented language such as C, the way in which pointers are used complicates performing a points-to analysis that is suffi-

[2] After processing with CPP, all blank lines were removed.

	lines of code		AST		CFG	
	before CPP	after CPP[2]	time	space	time	space
GCC	217,675	224,776	24.0	55.3	42.4	51.6
EMACS	99,439	113,596	16.9	39.3	22.1	29.3
BURLAP	49,601	88,057	10.0	23.3	14.8	16.3

Table 1.2: Statistics for constructing representations of three programs written in the C programming language. Time is given in seconds and space in megabytes.

ciently precise for the subsequent data-flow analysis. For instance, the use of specialized memory allocators can reduce precision by hindering the analysis's ability to accurately model heap storage. Furthermore, pointer arithmetic on arrays and structures limits the points-to analysis's ability to accurately discern distinct memory locations. Because these aggregates often store pointers to functions, the imprecise analysis can result in an overly conservative call-graph, degrading both the performance and precision of inter-procedural data-flow analysis.

Finally, pointer usage can complicate performing the subsequent data-flow analysis. Pointers to local variables are commonly used in C programs to emulate passing parameters by reference, which the language itself does not support. Pointers to local variables in the presence of recursion require changes to the traditional bit-vector equations for data-flow analysis [Aho et al., 1986], since different activations of a local variable may be referenced in functions other than the function in which it is declared. If the equations are not changed, the data-flow analysis will be in error. Naive solutions to this problem can be prohibitively expensive unless local variables are handled specially in the implementation of the data-flow equations.

1.4 Applying Our Approach to Pointer Analysis

We need to apply our lessons of adaptability and flexibility to the implementation of points-to analyses in order to construct practical whole-program analysis tools for programs written in languages such as C.

First, because points-to information cannot be demand-derived, we use Steensgaard's near-linear time, context-insensitive, flow-insensitive, points-to analysis algorithm [Steensgaard, 1996b]. To avoid the cost of an extra pass over the program, the points-to analysis is "piggybacked" with the demand construction of the control-flow graph (CFG). During points-to analysis, the representations (e.g., CFG) for functions needed for the data-flow analysis are retained, while other representations are discarded, saving space and improving reference locality. Since our approach previously saved the call-graph to disk to speed-up subsequent executions of the tool (Page 8), the saved call-graph now includes calls to functions through function pointers (as computed by the points-to analysis).

Second, to increase the precision of pointer analysis without unnecessarily increasing algorithmic complexity, we allow the user to parameterize the analysis. Since our tool is designed for interactive program understanding rather than for batch compilation, we can take advantage of information provided by the tool user. For example, the user might specify that the program being analyzed has only strict ANSI-compliant function prototypes, helping to more accurately determine which functions may be called through a function pointer. Such information cannot be obtained automatically by the tool without substantial additional cost, if at all. This information may be optimistic (i.e., "unsafe") or conservative. Although the information may be unsafe for any general program and therefore cannot typically be used, it may be safe for the specific program being analyzed. As long as the tool user can readily discern the unsafe results, or ensure that such information is in fact safe, not only can accuracy be substantially increased, but also time and space can be saved.

Finally, we derive new data-flow equations for dealing with pointers in the presence of recursion and pointers to local variables. Our new data-flow equations extend our work with user-controlled precision for C programs with pointers. By examining the characteristics of the data-flow analysis and adapting the implementation of the equations to the analysis, significant space can be saved.

1.5 Overview

In the following chapters, we discuss our approach to designing an adaptive, flexible, whole-program analysis tool. We feel that the problems introduced by pointers are sufficiently complicated that they warrant special discussion. Consequently, pointers are discussed in separate chapters. To evaluate our design, we discuss the application of our design choices to the construction of program slicers for CHCS and for C programs.

We use program slicing as our example data-flow analysis because it is a non-trivial, interprocedural analysis that is useful to programmers and designers and has a variety of potential applications [Weiser, 1984; Gallagher and Lyle, 1991]. We do not address the value of slicing and whether or not the computed program slices are useful to programmers or designers. However, the infrastructure that we have developed could be used to assess this.

Our results indicate that effective whole-program analysis is feasible using our approach. The time and space required to perform a program slice are a function of the size of the slice, not of the size of the entire program. For example, our tool can compute program slices of CHCS and GCC in less than one hour. We show that an iterative analysis such as program slicing converges quite rapidly, with substantially fewer and fewer statements being added during later iterations. For our example programs, 90% of the total statements in the slice are obtained within the first 20% of the iterations. This suggests that our decision to allow the user to suspend the program slicer and view the partially computed slice is warranted.

Our results also indicate that parameterization of the points-to analysis can dramatically increase the number of points-to classes. For programs that use function pointers heavily, the precision of the constructed call-graph can be substantially improved. As a result, program slices can be computed an order of magnitude faster and contain fewer unnecessary statements. Otherwise, we have found that the subsequent data-flow analysis is mostly insensitive to the improvement in precision [Shapiro and Horwitz, 1997a].

We have found that each aspect of our approach is essential to effective whole-program analysis. Should one aspect be omitted for our approach, the performance and effectiveness of the resulting whole-program analysis tool will suffer. In particular:

- Without demand-driven computation, the space and time required to perform analyses is necessarily a function of the size of the overall system, and is likely to exhaust the memory resources of most computers. Precomputing this data and storing it persistently does not solve the problem because using secondary storage may be no faster than computing the data on demand (Chapter 2).

- Without discarding, virtual memory can be exhausted by sizable representations that are not currently involved in the computation. Additional time is also expended in moving these representations out to disk (Chapter 2).

- Without persistent storage of key representations, deriving data that is costly to construct, albeit compact, can increase the start-up time of an analysis substantially (Chapter 2).

- Without providing control of precision, an analysis can take unnecessarily long if a high degree of precision is not required. On the other hand, providing only a low degree of precision may be ineffective in answering sensitive queries (Chapter 4).

- Finally, without the ability to control the termination of an analysis, it may run unnecessarily long to answer the question at hand (Chapter 4).

To overcome the problems associated with the use of pointers in modern programming languages, we present an approach for integrating points-to analysis with our demand-driven analysis, thus making the analysis more adaptive to the task at hand. We also present techniques for improving the flexibility of the points-to analysis by parameterizing the analysis to achieve better points-to results. Finally, we present data-flow equations for computing an interprocedural slice in the presence of pointers to local variables in recursive programs. Each aspect of our approach to handling pointers contributes to its effectiveness. In particular:

- Piggybacking the construction of the CFG with the computation of the points-to sets eliminates an extra pass over the program, saving time (Chapter 3).

- Persistently retaining the call-graph on disk allows only the reachable portions of the CFG to be retained, saving space and also time by avoiding the use of virtual memory (Chapter 3).

- Parameterization of the points-to analysis increases the effectiveness of the subsequent data-flow analysis. In the absence of function pointers, the increase in the number of points-to sets does not result in a substantial increase in precision, due to the transitive effects of the data-flow analysis. However, in the presence of function pointers, the computed call-graph is substantially more precise, which greatly increases the precision of the data-flow analysis with respect to function calls and realizable paths (Chapter 5).

- Through an aggressive implementation of the data-flow sets, significant space can be saved, making whole-program analysis practical in the presence of pointers to local variables in recursive programs (Chapter 6 and Chapter 7).

To conclude the dissertation, we summarize our work, discuss some of the open issues and how our work can be extended in different ways and to other languages, and briefly discuss our infrastructure for constructing whole-program analysis tools.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998] and [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of these publications directed and supervised the research that forms the basis for this chapter.

Chapter 2

Adaptive Whole-Program Analysis Tools

Since program understanding tools extract detailed information from the program source, their designs have tended to borrow heavily from optimizing compilers. However, the added requirements of full interprocedural analysis and the wide range of user queries stress traditional compiler designs. Demand-driven computation, discarding, and persistence of data on disk can improve performance substantially, but these are not accommodated by standard compiler practice. Because the algorithms used in compilers are quite complicated, our goal is to introduce techniques that minimally perturb these algorithms, while also giving us the performance we desire.

2.1 Background: Compiler Architectures

Figure 2.1 presents a typical software architecture for the front-end of a compiler, which iterates over each file in the program. The general flow of control is from top-to-bottom and left-to-right. The flow of data is left-to-right. The space required in a typical optimizing compiler is not prohibitive, since the program representations for one file are discarded before processing the next file, as they are no longer needed. However, if the representations were to be retained for later use, as required by a whole-program

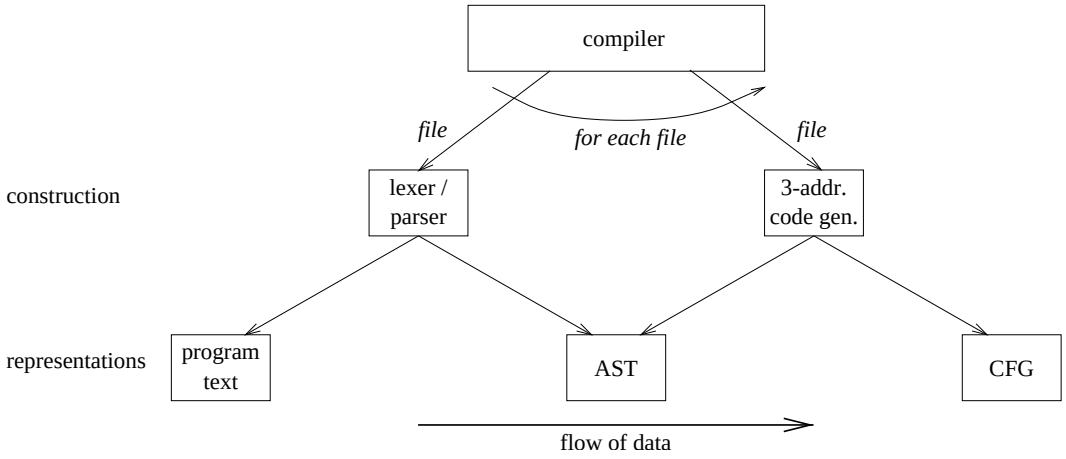


Figure 2.1: Typical front-end compiler architecture, showing iteration over each file of the program. Boxes indicate modules and arrows indicate calls. Italicized items designate program components being accessed.

---

analysis tool, then the resulting space could be prohibitive. Deriving required representations on demand, discarding representations that are infrequently used (but not too expensive to reconstruct if needed), and retaining representations that are expensive to compute (but require little space) are obvious solutions to this problem.

2.2 Approach: Overview

A demand-driven algorithm [Choi et al., 1991; Choi et al., 1994; Horwitz et al., 1995] can reduce the space (and time) requirements of an analysis by ignoring portions of the program that are irrelevant to the analysis. However, we have found that demand-driven construction of a single program representation does not sufficiently reduce the space requirements, since many program representations are derived from other representations. For example, a representation such as the PDG is often derived from other representations such as the CFG and the control dependence graph (CDG). The CDG may itself be computed from the CFG and the reverse dominator tree of the program [Cytron et al., 1991]. Although in some analyses the intermediate representations

may be *a priori* discarded, in others many must be retained [Griswold and Notkin, 1993]. For example, our program slicer depends upon a CFG, dominance frontiers [Cytron et al., 1991], and an AST for display. Thus, there are many representations for each procedure in the program, but large portions of some of these representations are used infrequently or not at all. In computing a backward slice, for example, the only needed portions of the program representations are those that are on the control-flow path from the beginning of the program to the slicing criterion.

Basic demand-driven computation does not provide all the savings possible. In particular, our analysis in Table 1.1 shows that retaining an infrequently used representation can exhaust the main memory or even the virtual memory resources of the computer. Thus, we choose to discard such representations—in our case the AST—and recompute them when they are required. Although this adds time to recompute any discarded data that is later needed, we can still achieve savings by avoiding the cost of (1) moving out retained data to the slower parts of the memory hierarchy, and (2) retrieving it later when needed.

Other representations are expensive to compute and are used frequently, but require little space. For instance, our slicer needs to compute the callers of a procedure, which would normally be resolved by the second pass over the CFG, as discussed in Chapter 1. Although this information is demanded like other representations, it is stored on disk rather than discarded. Subsequent runs of the slicer on the same program can reuse this information as long as the program has not changed.

2.3 Approach: Software Architecture

Many control-flow and data-flow analyses such as interval analysis [Aho et al., 1986] or alias analysis [Landi and Ryder, 1992; Choi et al., 1993] are sufficiently complicated without the additional burden of requiring the algorithm to demand-derive additional data structures. It is desirable to make minimal changes to these algorithms when addressing the problems encountered when analyzing large systems. We have re-

vised the standard software architecture for compilers to allow us to make only small changes to existing analysis algorithms and yet support the demand-driven construction and subsequent discarding of the program representations.

The primary problem with the standard architecture is that the flow of control largely follows the flow of data from source to sink. This flow is controlled from the top-level analysis algorithm. However, demand-driven computation requires that the sink must be able to "demand" data from the source, reversing the control-flow relation to be not only right-to-left, but also coming from the bottom of the hierarchy, not the top. One solution to this problem is to have the CFG module directly call the AST module, and so forth. However, this solution significantly reduces the independence of the CFG module. For instance, it no longer could be constructed easily from representations other than the AST. A solution that instead modifies the analysis algorithm would further complicate an already complicated algorithm. Additionally, each new algorithm would require essentially the same (complex) modifications. The redundancy distributes the design decisions regarding demand-driven computation across several system components, potentially complicating future changes related to those decisions.

To accommodate the needed changes in control-flow without compromising independence, our solution is to modify the existing architecture to use events and mappings. This architecture borrows from our previous experience with layered and event-based architectures [Griswold and Notkin, 1995; Griswold and Atkinson, 1995], but these architectures do not accommodate demand-driven computation or discarding. Figure 2.2 presents an example of our architecture containing three program representations (the program text, the AST, and the CFG) with each representation fully encapsulated inside a module. Accessed data structures, shown italicized, are program components, rather than the entire program or whole files as in Figure 2.1. Unlike in the compiler architecture of Figure 2.1, the analysis algorithm does not call the construction modules directly, since the program representations are demand-derived as they are accessed through their module abstractions. The architecture's underpinnings, described below, take care of computing the required structures.

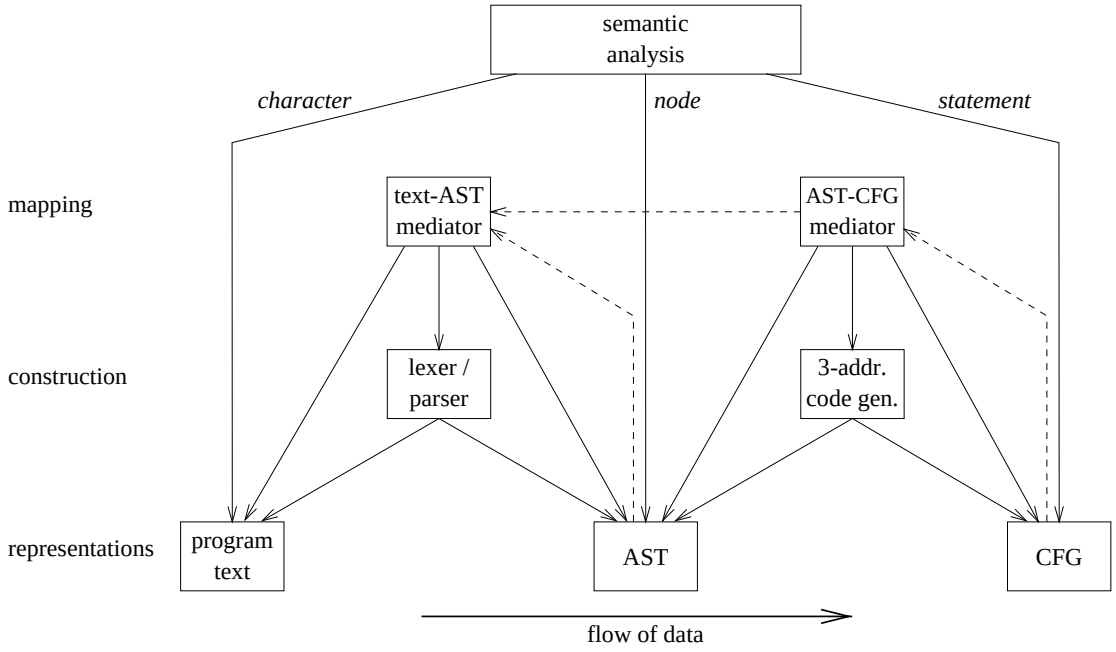


Figure 2.2: Software architecture for a whole-program analysis tool. Boxes indicate modules, solid arrows indicate calls, and dashed arrows indicate events. Italicized items designate program components being accessed.

2.3.1 Mediator modules

Many semantic tools must maintain mappings between the various program representations. For example, in a program slicer, when the user selects a variable in the program text represented by the AST, the corresponding AST node must be mapped to a three-address statement in the CFG to begin slicing. When the resulting slice is displayed, the CFG statements in the slice must be mapped back to their corresponding AST nodes. These mappings could be maintained explicitly within each module (e.g., by having each AST node contain a pointer to its corresponding CFG statement), but this would reduce the independence of the individual modules [Sullivan and Notkin, 1992]. Instead, we use separate *mediators* to maintain mappings between the modules [Sullivan, 1994; Sullivan and Notkin, 1992]. However, since the representations need to be constructed on demand, each mediator may call the required construction module for

the representation, as shown in Figure 2.2. For example, the AST-CFG mediator module, which maintains a mapping between AST nodes and three-address statements in the CFG, calls the code generator if there is a need to map an AST node to its corresponding CFG statement, but that statement has either never been constructed or has been discarded.

2.3.2 Events, callbacks, and protocols

Giving mediators the ability to construct representations on demand does not allow the program representations to demand-derive *each other*. The `called_routine` operation on a CFG `call` statement, for example, may need to access a three-address statement that has not been constructed yet, which may in turn require construction of the AST nodes from which it is to be derived. Rather than have the CFG module call the AST-CFG mediator, which would require a modification to the CFG module and consequently reduce its independence, our solution is to use events [Sullivan and Notkin, 1992], shown in Figure 2.2 as dashes. The CFG module can send an event "announcing" that it is about to execute the `called_routine` operation. The mediator module "hears" this announcement, and thus responds to the event by calling the code generator, if necessary. For the mediator to hear the announcement, the event handler of the mediator module must be registered with the CFG module by an initialization module (not shown in Figure 2.2).

If an event were announced for every exported CFG operation, the resulting overhead could be prohibitive. This cost can be reduced by having a high granularity for event announcements: the CFG module announces an event for accesses to major program components such as a procedure, and as a result the CFG for an entire procedure may be constructed. This concept of processing granularity for events and the construction of program representations unifies the entire architecture, since it naturally exploits the structure of the problem [Johnson, 1978], namely interprocedural analysis. The intraprocedural algorithms are unaffected. If the CFG was constructed incrementally for each statement and the AST constructed incrementally for each file, the resulting architecture would be more complicated.

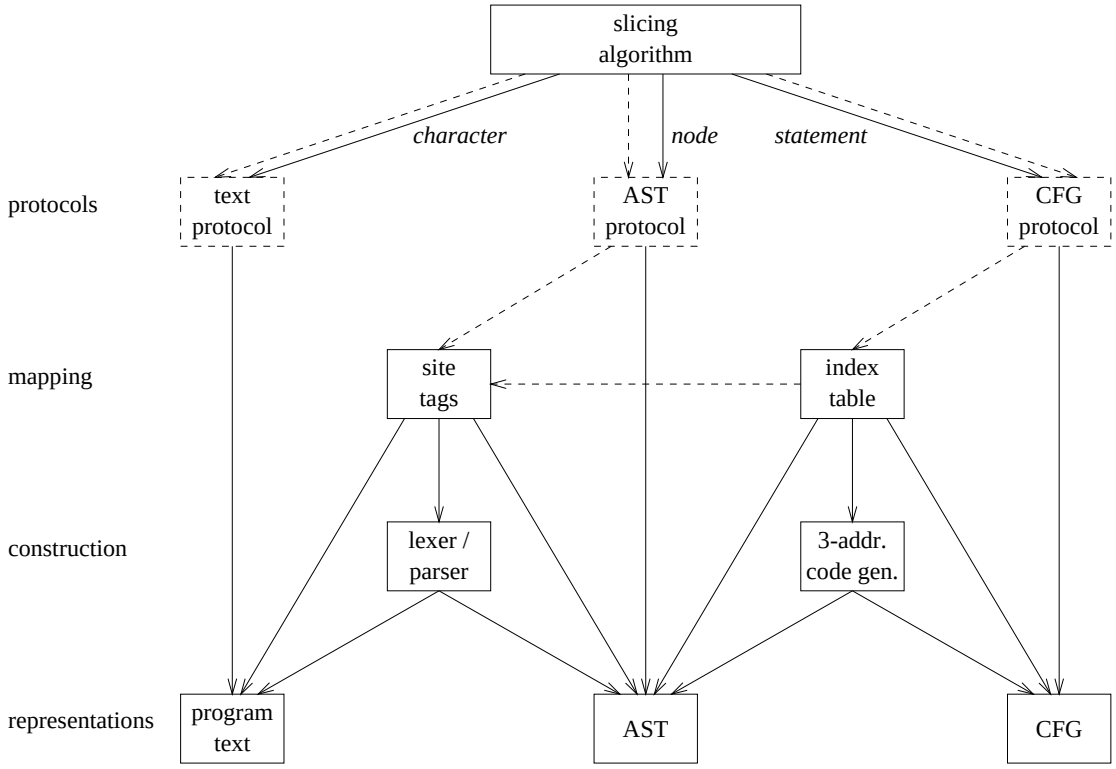


Figure 2.3: Implementation of the program slicer for CHCS, adapted from Figure 2.2 with the addition of a protocol layer. Boxes indicate modules, solid arrows indicate calls, and dashed arrows indicate protocols. Italicized items designate program components being accessed.

Our architecture can also be described and implemented in terms of *callbacks* or *protocols*. Callbacks are similar to events with the distinction that events are usually asynchronous but callbacks are usually synchronous [Nye, 1990]. In some event models, events also cannot return data to the module that announced the event. However, both events and callbacks provide "late binding" between modules. In our architecture, we do not wish the program representation modules do not contain explicit references to the mediator modules.

To use either events or callbacks, the representation modules must be instrumented with the appropriate operations. However, this task may not always be possible or practical. For example, the representation modules might be provided externally by a

software library and cannot be modified. To avoid these problems and minimally perturb the underlying tool infrastructure, we can instead add a *protocol layer*. As an example, the architecture for our program slicer for CHCS is shown in Figure 2.3. In our slicer, the *site-tags* module maintains mappings between the program text and the AST, and the *index-table* module maintains mappings between the AST and the CFG.

The modules of the protocol layer are virtual in that they do not manifest themselves as functions, but rather as requirements. If the slicer wishes to call the `called_routine` function of the CFG module, it must first obey the protocol that requires it to first "announce" an event to the index module. Rather than events flowing from a lower layer to a higher layer, the protocol requests flow from a higher layer to a lower layer. The protocol layer requires no modifications in the lower layers, but instead places a burden on the client (i.e., the slicer) of these layers. However, to both minimize additions to the client and increase performance, the protocol uses the same high level of granularity between requests as proposed for events (i.e., procedure granularity).

2.3.3 Address-independent mappings

If a representation may be discarded, then the mapping module must support *address-independent* mappings. These are in essence a pointer abstraction similar to that provided by virtual memory, but resulting in the rederivation of data, rather than the movement of data. Since the AST may be discarded, the AST-CFG mediator must support this type of mapping. Address-independent mappings can be implemented, for example, by assigning each AST node a unique index number that can be reassigned to a reconstructed node, or by using the file name and character position as a key for each AST node.

For example, in our program slicing tool for CHCS, the index-table module functions as a mediator, maintaining mappings between AST nodes and three-address statements in the CFG, as shown in Figure 2.4. The address-independent mappings are maintained using *index numbers*. When the AST for a routine is constructed, each node is assigned an increasing index number during a preorder traversal of the AST. The index

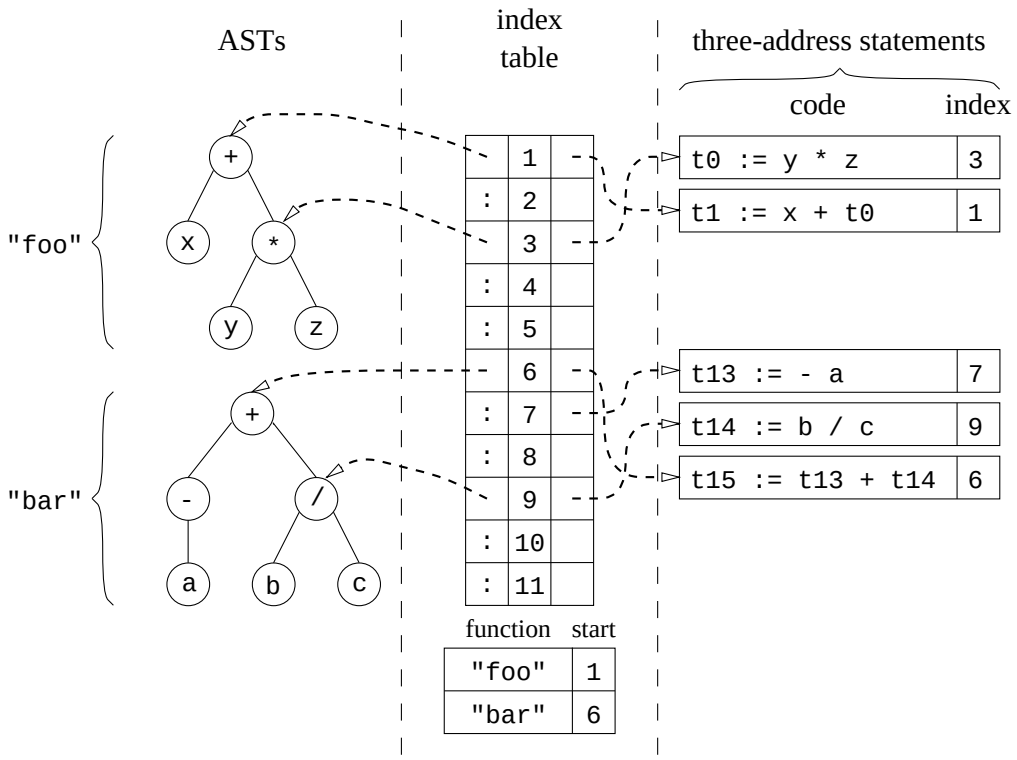


Figure 2.4: The index-table module maintaining mappings between the ASTs for two functions and the associated three-address statements.

numbers of each root node are also stored in a small auxiliary table for later use. The index-table itself contains bidirectional mappings from AST nodes to three-address statements using hash-tables. Each three-address statement also contains an index number representing its associated AST node.[1] If the AST for a routine is destroyed, its corresponding entries in the index-table are removed. If the AST needs to be reconstructed for a given three-address statement (i.e., a miss occurs in accessing the hash-table), the AST for the entire routine containing the statement is reconstructed. The routine name and its starting index number are determined by searching the auxiliary table using the index number of the three-address statement, and a preorder traversal of the AST is performed to update the index-table.

[1] An index number is explicitly stored within a three-address statement for simplicity. A design with more separation would be to use another hash-table for mapping statements to index numbers.

2.4 Conclusion

Demand-driven construction of program representations can greatly improve the performance of whole-program understanding tools. However, some program representations are large but accessed infrequently and require little time to compute. We advocate discarding such representations and recomputing them when needed in order to reduce to overall space required and also save time by avoiding use of the slower parts of the virtual memory hierarchy. Finally, some representations require a long time to compute, but require little space. We argue that these representations should be constructed once and then persistently retained on disk across invocations of the tool. This technique reduces the start-up time of the tool since it is faster to read the information from disk than it is to compute it.

To implement our approach, we designed a new software architecture, since the standard compiler architecture does not accommodate many of our new features. However, to minimize the impact on the top-level analysis algorithm, our architecture uses mediators, events, and address-independent mappings to transparently demand-derive the program representations.

Using our approach, we successively implemented program slicing tools for the MUMPS and C programming languages [Atkinson and Griswold, 1998]. Our slicing tool for MUMPS can compute slices of CHCS, a 1,000,000 line hospital management system, in under 10 minutes (Chapter 8). Although some slices can take longer to compute, the space and time consumed by the analysis are proportional to the size of the slice, not to the size of the entire program. However, before we could successfully compute slices of C programs, we needed to solve the problems introduced by pointers (Chapter 3, Chapter 5, and Chapter 6).

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of this publication directed and supervised the research that forms the basis for this chapter.

Chapter 3

Points-To Analysis with Demand-Driven Analyses

By using our adaptive, demand-driven framework described in Chapter 2, we are able to construct an efficient, task-oriented program slicing tool for large programs written in MUMPS, a language without pointers (Chapter 8). However, we need to integrate pointers, and in particular performing a points-to analysis, into our demand-driven framework if we wish to compute slices of more modern languages such as C.

3.1 Background

Points-to analysis simply determines, for each variable in the program, the set of variables that may be pointed to by a given variable [Emami et al., 1994; Steensgaard, 1996b; Shapiro and Horwitz, 1997b]. For the purposes of explanation, we use the term *variables* to mean programmer-declared variables together with other locations that may be distinguished by the points-to analysis, including members of structures, dynamically allocated data, and other temporary locations used by the points-to analysis or subsequent data-flow analysis. Figure 3.1 shows a small C program with the results of its points-to analysis. Figure 3.1b shows the results of a *flow-sensitive* points-to analysis, and Figure 3.1c shows the results of a *flow-insensitive* analysis.

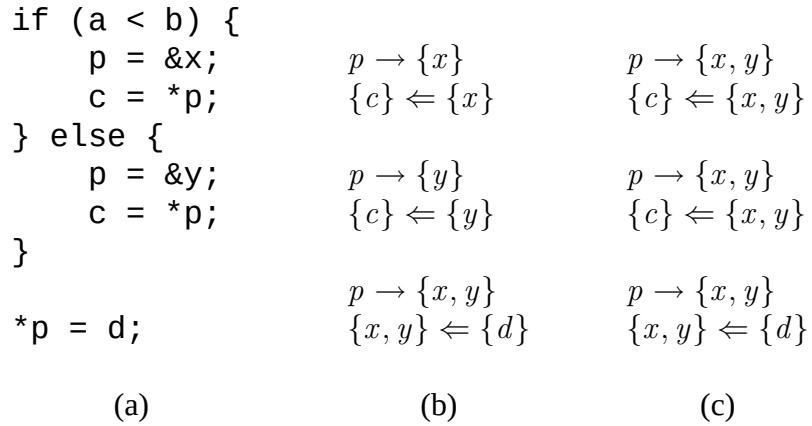


Figure 3.1: An example C program with points-to analysis: (a) an example program, (b) annotations for a flow-sensitive analysis, and (c) annotations for a flow-insensitive analysis. An annotation of $i \rightarrow \{j\}$ indicates that i now points to j. An annotation of $\{i, j\} \Leftarrow \{k\}$ indicates that i and j are both assigned the value of k.

For the flow-sensitive analysis, the points-to set for the variable `p` changes according to the flow of data through the program. For the flow-insensitive analysis, the points-to set is the same throughout the program. Although, flow-sensitive analyses are more precise than flow-insensitive analyses, they are not suitable for use on large systems because they have quadratic to exponential time and space requirements.

Dereferencing the pointer variable `p` yields the contents of its points-to set. The simple assignments involving `c` and `d` are now assignments from and to sets of variables.[1] Larger points-to sets are usually the result of a more conservative points-to analysis. The sizes of the points-to sets can greatly affect the precision and running time of the subsequent data-flow analysis. For example, in backward program slicing, an assignment to a variable in the current slicing criteria results in the variable being removed from the criteria and the variables used at the assignment statement being added to the criteria. If the variables used are the result of a pointer dereference, then all the variables in the corresponding points-to set are added to the criteria. If the points-to set is large, the slicing criteria will also be large.

[1] If more than one variable is the target of an assignment, such a definition is a preserving definition since in fact only one variable is actually assigned a value.

3.2 Motivation

Demand-driven techniques attempt to save space and time by computing only those data-flow facts and portions of supporting representations that are necessary to perform the analysis [Atkinson and Griswold, 1996; Duesterwald et al., 1995; Horwitz et al., 1995]. In this way, large programs can be handled more economically since the amount of information computed and stored is greatly reduced. Effective demand-driven analysis depends upon quickly identifying which portions of a representation are required next and efficiently computing those portions. In backward program slicing, for example, it is necessary to quickly identify all the callers of a procedure and efficiently construct the CFG for those calling procedures [Atkinson and Griswold, 1996]. Because determining the callers of a procedure requires a global analysis of the program, our demand-driven approach saves the call-graph to disk for future invocations of the program slicing tool.

Depending on the algorithm chosen (e.g., a flow-sensitive algorithm versus a flow-insensitive algorithm), points-to analysis for large programs can require a large, possibly prohibitive, amount of time and space. Unfortunately, there are problems with either demanding or persistently storing points-to information. We discuss these problems and then present a hybrid compute-and-store solution.

Points-to information is not efficiently computable on demand because computing the effects of any particular pointer reference can require a global analysis of the program. For example, Figure 3.2 shows a small C program consisting of three source files. If a backward program slice is started at the assignment to `z` in function `f()` of file `y.c`, the points-to set of variable `p` is needed. There is an assignment to `p` in function `main()` in file `x.c`, so `x.c` must be analyzed. Ignoring pointers, a demand-driven slicer would not need to examine this file unless the user requested that slicing should continue into the calling function. File `z.c` must be also examined. Although function `g()` is not reachable during a backward data-flow analysis from `f()`, the file contains the initialization for `p` in a static initializer.

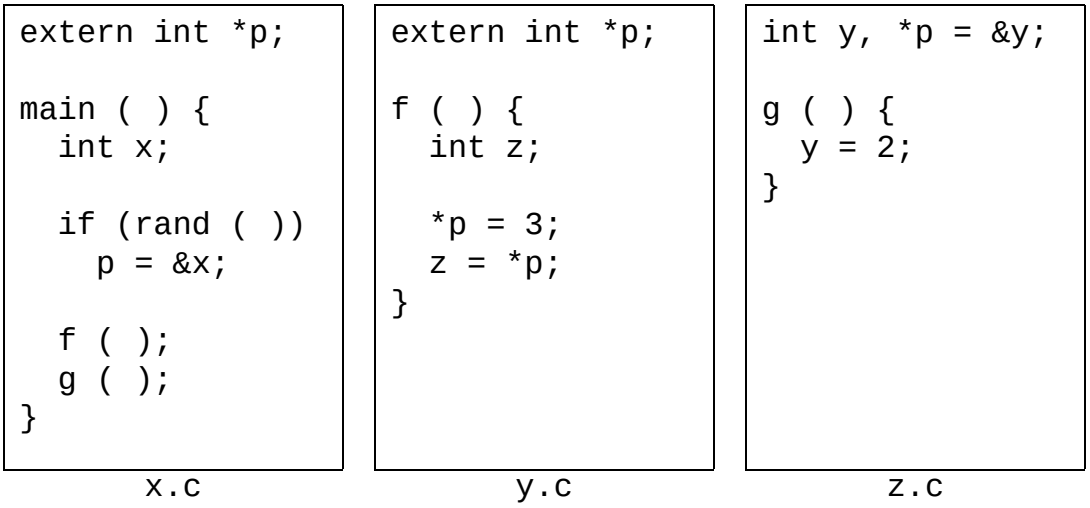


Figure 3.2: An example C program consisting of three files. The pointer variable p is referenced in all files.

An alternative to demand-driven analysis is to persistently retain the points-to information in a database, as we do with the call-graph. This approach is attractive since the call-graph requires pointer information for computing the effects of calls through function pointers anyway. However, storing the pointer information presents several difficulties. First, a representation would be needed for referencing an arbitrarily nested variable declared within a function. Second, the CFG's three-address statements and associated temporaries would need to be constructed in a reproducible order from one tool invocation to the next. Finally, the database must be recomputed if any variable in the program changes, not just if the call structure changes. Although none of these difficulties is overwhelming, their net complexity led us to consider a third alternative.

3.3 Approach

Our approach is to demand all the points-to information on invocation of the first slice, employing three techniques to minimize the impact of the required global analysis.

- We use Steensgaard's near-linear time, context- and flow-insensitive, points-to analysis, which models storage as equivalence classes of locations and computes the points-to sets (points-to classes) by computing the transitive relation over assignments [Steensgaard, 1996b].[2] Although not as precise as some techniques, its time–space characteristics are superior and the difference in precision is often not reflected in the subsequent data-flow analysis [Shapiro and Horwitz, 1997a].

- To avoid an extra pass over the program to perform the global analysis, we piggyback the computation of points-to information with the construction of the portions of the CFG required for the subsequent data-flow analysis, as shown in Figure 3.3(b). The call-graph, which was formerly used to demand only those portions of the CFG reachable from the initial slicing criterion, is now used to determine which portions of the CFG are needed only for points-to analysis and can therefore be discarded immediately after use.

- To maintain the call-graph's effectiveness in the demand-driven analysis, the call-graph saved to disk includes the effects of calls through function pointers, as determined by the points-to analysis, as shown in Figure 3.3(a). Since the points-to analysis is flow-insensitive—in particular it does not require a call-graph—performing points-to analysis in a prior pass to gather function pointer information adds little complexity to the implementation of the program slicer.

Using this approach on GCC, our largest program, computing the points-to information and other supporting data for the call-graph requires 52 seconds and 63 MB of space (Chapter 8). Although the CFG itself would require only 52 MB if fully constructed, the total savings due to CFG discarding can be substantial. For example, if only half of the CFG needs to be retained for slicing, the savings of 26 MB might be sufficient for the entire analysis to reside in main memory, eliminating paging and thus improving overall execution time.

[2] Our implementation treats relational operators differently from arithmetic operators since the former do not yield a pointer value. This fact is mentioned in the reference but not included in its equations.

```
for all files do
  for each function f do
    compute-classes (f)
    for each statement call g ( ) do
      calls [f] := calls [f] ∪ {g}
    end for
    for each statement call (*p) ( ) do
      indirect [f] := indirect [f] ∪ {p}
    end for
    discard (f)
  end for
end for all

for each f in calls do
  for each p in indirect [f] do
    calls [f] := calls [f] ∪ *p
  end for each
end for each

write-call-graph ( )
```

(a)

```
function get-reachable (f)
  reachable := φ
  dfs-over-calls (f, reachable)
  return reachable
end function

read-call-graph ( )
start := get-criteria ( )
reachable := get-reachable (start)

for all files do
  for each function f do
    compute-classes (f)
    if f ∉ reachable then
      discard (f)
    end if
  end for
end for all

compute-slice (start)
```

(b)

Figure 3.3: Example pseudocode showing how the call-graph is built and used: (a) the direct calls and points-to classes for each function is first computed and then combined to construct the call-graph which is then written; (b) the call-graph is read and used to determine the reachable portions of the CFG while computing the points-to classes.

3.4 Conclusion

Integrating points-to analysis with demand-driven analyses is difficult. Points-to analysis typically requires a global analysis over the program, negating the benefits of a demand-driven analysis. To overcome this problem, we use a near-linear time points-to analysis that can be performed in a single pass over the program. By piggybacking construction of the call-graph with the computation of the points-to sets, an extra pass over the program can be avoided. The call-graph is persistently retained on disk and read in upon starting a program slice. The call-graph is then used to determine the

reachable portions of the CFG, allowing the unreachable portions of the CFG to be discarded, thereby saving space and also time by avoiding use of the slower portions of the virtual memory hierarchy.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998]. The dissertation author was the primary researcher and author and the co-author of this publication directed and supervised the research that forms the basis for this chapter.

Chapter 4

Flexible Whole-Program Analysis Tools

4.1 Motivation

Reducing the space requirements of an algorithm will also reduce its running time by avoiding the movement of representations within the virtual memory hierarchy (Chapter 2). Because an algorithm can require between polynomial and exponential time, depending on its precision, it is also necessary to control the time complexity of the algorithm itself in order to obtain acceptable performance of a whole-program analysis tool. Since many data-flow analyses are iterative [Aho et al., 1986], significant improvements can be achieved by either reducing the running time of each iteration or by reducing the number of iterations performed.

4.2 Approach: Controlling Precision

As with demand-driven computation, providing the tool user with control over the precision of an analysis should require only minor modifications to the original iterative algorithm, since the algorithm can be quite complex. One approach is to allow the tool user to specify the interprocedural *context-sensitivity* of the algorithm. Before

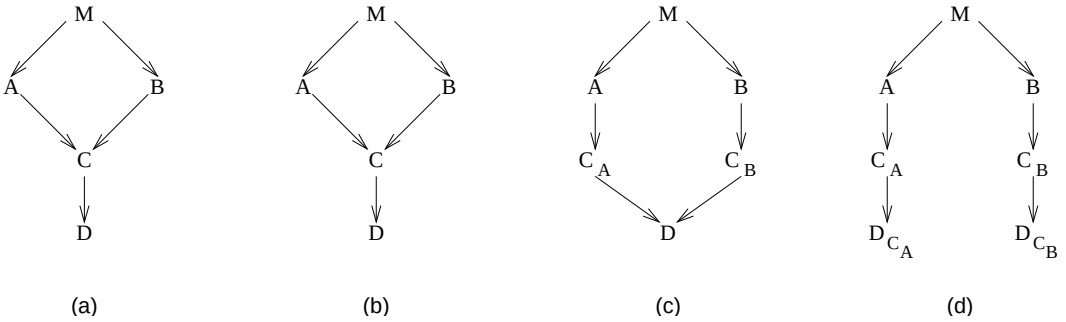


Figure 4.1: A sample program with (a) its call-graph, (b) its depth-1 context-graph, (c) its depth-2 context-graph, and (d) its unbounded-depth context-graph. A procedure with multiple contexts is annotated with its call path.

---

we describe our support for this feature, we present some background on precision and context-sensitivity.

4.2.1 Background

Recent work has focused on the trade-offs between context-insensitive and context-sensitive analyses [Wilson and Lam, 1995; Ruf, 1995]. Many approaches such as the slicing algorithm of Weiser [Weiser, 1984] use only a single calling context for each procedure—that is, there is no accounting of the calling sequence that led to the call in order to precisely estimate the calling sequence's influence—and therefore are *context-insensitive*. In contrast, the invocation graph approach [Emami et al., 1994] is fully *context-sensitive* since each procedure has a distinct calling context for each possible call sequence starting from the main procedure of the program.[1] The invocation graph can be understood as treating all the procedures as inlined at their call sites. Figure 4.1 presents the call-graph of a simple program, with *M* representing the main procedure, along with various *context-graphs*.

The nodes of a context-graph represent a calling context of a procedure and the edges represent procedure calls. Each context-graph has an associated *context-depth*.

[1] Recursion is handled by following the recursive call once and then using the resulting data-flow set of the recursive call as an approximation for subsequent calls.

Figure 4.1b shows the context-graph using Weiser's approach. Since each procedure has only a single context, this graph is identical to the call-graph of Figure 4.1a. This is the *depth-1* context-graph, since the context of a procedure is determined by searching a *single* procedure down the call stack. For example, the call stacks *M A C* and *M B C* (shown growing from left to right) are equivalent since only the topmost procedure, *C*, is examined in tracing the call stack. Figure 4.1d shows a context-graph equivalent to the invocation graph for the program, with procedures having multiple contexts annotated by their call path. This graph has effective *unbounded-depth*, since the context of a procedure is determined by searching back through the call stack as many procedures as necessary to reach the main procedure.

4.2.2 Approach

In order to control precision, our approach allows a variable degree of context-sensitivity. For example, Figure 4.1c shows the *depth-2* context-graph for the program. The call stacks *M A C* and *M B C* are not defined to be equivalent since the depth-2 call stacks *A C* and *B C* are unequal, resulting in two contexts for *C*. However, *D* still has only a single calling context since the call stacks *M A C D* and *M B C D* are equivalent, as both have a depth-2 call stack of *C D*. This approach is similar to the approach of Shivers [Shivers, 1991] for analyzing control-flow in languages with functions as first-class objects.

A depth-1 context-graph has an equal number of procedures and contexts, resulting in a high degree of imprecision but an efficient analysis. An iterative algorithm using a depth-1 context-graph with n procedures and m data-flow facts will require $O(n)$ space and $O(mn)$ time in the worst case. An iterative algorithm using an unbounded-depth context-graph will produce a precise result but will require exponential space and time in the worst case. As the context-depth increases the analysis becomes more precise, but requires more time and space.

The tool user may first perform the analysis at a low context-depth and examine the results, as shown in Figure 4.2. If the user's query has not been satisfactorily

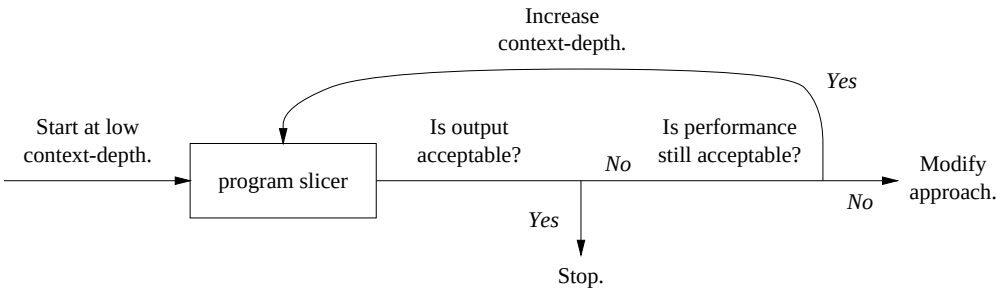


Figure 4.2: Example scenario showing how context-depth can be used in a program slicing tool to solve a tool user's question about a program.

---

answered then the context-depth is increased until either a satisfactory answer is produced or the running time of the analysis becomes unacceptable. For example, suppose that some function in a large system has recently been changed and soon afterward the system behaves incorrectly. A programmer might suspect that the recently changed function is the cause of the error. Using a backward slicing tool, the programmer can compute a slice at a low context-depth starting at the statement where the error occurs. If the computed slice does not include the suspected function, then the programmer knows that the function did not in fact cause the error. If the context-depth were to be increased, then precision would be improved and less statements—not more—would be included in the slice. However, if some statement in the suspected function is included in the slice, then the function might be the source of the error. The programmer would then need to increase the context-depth and recompute the slice if the performance of the tool is still acceptable and the function is still believed to be in error.

The context-graph approach integrates easily with our demand-driven software architecture (Figure 2.2). In order to isolate the analysis algorithm from our additions, we introduce a context module that encapsulates the control of context sensitivity. When a data-flow algorithm traverses a call edge of the CFG, a new context is demanded for the called procedure. The context module either creates a new context for the procedure or returns a pre-existing context. As a consequence, the analysis algorithm is impervious to the changes in context-sensitivity. The only real difference is the context-

graph that is implicitly traversed. Contexts are demanded with a standard procedure call to the context module, not an event, since an analysis algorithm is not logically independent of precision.

4.2.3 Calling Context Creation

When a data-flow algorithm traverses a call edge of the CFG, a new calling context must be demanded for the called function. The context module either creates a new context for the function or returns a pre-existing context. Our backward slicing algorithm performs the following operations when a `call` statement is encountered:

1. Demand-derive a context for the called function, *Q*, using the context of the calling function *P*.

2. Using the current slicing criterion, create new slicing criteria at the `return` statements of the context for *Q*. The new slicing criteria are merged with any already existing criteria using a `union` operation (Chapter 6).

3. Compute the slice of *Q* during a backward depth-first search of *Q* from each `return` statement (Chapter 7).

4. Use the updated criterion at the first statement of *Q* as the new slicing criterion for the `call` statement and resume slicing *P*.

In the depth-1 context-graph of Figure 4.1b, first *M* slices into *A*, and *A* calls *C* by demand-deriving a context for *C* and updating the slicing criteria at the `return` statements of *C*. After a depth-first search of *C*, the criteria at the first statement of *C* is used to continue slicing *A*. Next, *M* calls *B*, and *B* follows the same steps as *A*. However, since *C* has only one context, the criteria from *A* and *B* are merged in *C*. The depth-first search returns immediately, since all blocks in *C* have been marked as visited by *A*, and *B* uses the (approximate) criterion at the first statement of *C* to continue slicing. Thus, data placed in *C* by *A* flows back into *B*, and on the next iteration the data from *B* will flow back into *A*, resulting in imprecision. If the depth-2 context-graph of Figure 4.1c

is used, this imprecision will not occur; however, some imprecision may still occur since *D* has only a single context. Using a depth-3 context-graph, which is equivalent to the unbounded-depth context-graph for our sample program, will result in a precise analysis.

Unless an unbounded-depth context-graph is used, data may be propagated along *unrealizable paths* [Horwitz et al., 1995; Horwitz et al., 1990]. For example, data merged at the `return` statements of *C*, which is the source of the imprecision, is propagated along unrealizable paths (e.g., the data of *A* is propagated through *C* to *B*). Our slicing algorithm tries to avoid unrealizable paths. Since the call to *C* from *A* returns to *A* and not to *B*, should the slicing algorithm terminate before *B* is called from *M* then no imprecision will result.[2]

4.3 Approach: Customizable Termination

Controlling precision can reduce the running time of each iteration of an iterative analysis, thereby reducing the overall time needed to perform the analysis. However, the overall running time may still be unacceptable for many uses of a whole-program analysis tool. For example, in our scenario in which the programmer is trying to determine if a given function is the source of an error, the programmer needs the program slice to be computed quickly. If the program slice requires a long time to compute, the iterative scenario that we described would be impractical. Consequently, to further reduce the running time of the analysis, the number of iterations needs to be decreased. One approach is to allow the user to limit the number of iterations performed.

If an iterative analysis initially converges towards the ultimate answer quickly, but does not complete for some time, then *customizable termination* can substantially reduce the analysis time required. One way to provide user-controlled termination of an analysis is to permit the user to suspend an analysis, examine the intermediate results,

[2]The algorithm may terminate if the slicing criterion becomes empty or if the algorithm is interrupted by the tool user. Additionally, *B* may not be called from *M* if the call paths are constrained, as in chopping. [Jackson and Rollins, 1994; Reps and Rosay, 1995]

and decide if the analysis has sufficiently answered the tool user's question. Another way is to allow the user to provide a termination test procedure that is periodically applied to the current result of the analysis. A simpler but less flexible approach is for the tool to provide a fixed set of parameterized termination tests.

Supporting customized termination requires a minor modification to the analysis algorithm. Events can be used to announce that a certain slicing milestone is met—such as the end of an iteration—giving the tool's user interface an opportunity to update the display and apply the user's termination test to the current results of the analysis. The only requirement that the events impose is that the analysis's data structures should be consistent so that they can be viewed without crashing the tool. In this sense, the event protocol of a module is an inherent part of the module's behavior [Sullivan and Notkin, 1992].

Our program slicing tool currently provides suspension of an analysis for inspection of the current results. The user can unobtrusively monitor the progress of the analysis by means of an on-the-fly display. Our program slicer allows viewing the number of statements analyzed, size of the slice, and other criteria interactively.

4.4 Conclusion

By increasing the flexibility of a whole-program analysis tool, the time and space requirements can be substantially reduced. We advocate allowing the tool user to have high-level control over the data-flow analysis being performed. By allowing the user to control the precision of the analysis, substantial time and space can be saved by performing an analysis that better matches the tool user's needs. By allowing the user to control the termination of the analysis, the computation of unnecessary data-flow information can be avoided, saving time.

We have implemented our approach in our program slicing tools. We have found that the data-flow information computed in backward program slicing has an initial rapid convergence, with 90% of the total number of statements in the slice being

included within the first 20% of the total iterations (Chapter 8). By tuning the amount of context-sensitivity, significant time and space can be saved. We have found that increasing the context-depth yields only a small improvement in precision of the data-flow analysis (i.e., a small reduction in the number of statements in the program slice), but increases the running time of the analysis considerably. Our approach of improving flexibility through user-specified information can also be applied to the computation of the points-to information for C programs, substantially improving the results of the subsequent program slices (Chapter 5).

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of this publication directed and supervised the research that forms the basis for this chapter.

Chapter 5

Pointer Usage in Large Systems

In this chapter, we examine how the ways in which pointers are used can impact the precision of the points-to analysis. If the points-to analysis is too conservative, then the subsequent data-flow analysis will suffer (e.g., be too conservative, require too much time and space). In the previous chapter, we showed that by improving the flexibility of an analysis tool significant time and space can be saved by avoiding the computation of unnecessary data-flow information. In particular, we improved flexibility by allowing the tool user to parameterize the data-flow analysis. To overcome the problems introduced by stylized pointer usage in C programs, we allow the tool user to parameterize the points-to analysis. In Chapter 8, we present our results on how parameterization can significantly improve the precision of the points-to and data-flow analyses.

5.1 Motivation

C provides powerful, albeit low-level, language features like type casting, pointer arithmetic, and function pointers. Programmers often use these sophisticated language constructs in order to improve performance and ease implementation. For example, all of our example systems use an array of function pointers to implement a *dispatch table*—a table 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.

```
# define SIN 0
# define COS 1
   ...
# define POW 10

struct {
  double (*func) ( );
  int args;
} func_tab [ ] = {
  {sin, 1},
  {cos, 1},
   ...
  {pow, 2},
};
```

(a)

```
n = func_tab [i].args;
f = func_tab [i].func;

x = pop ( );

if (n == 1)
  return (*f) (x);

y = pop ( );

if (n == 2)
   return (*f) (x, y);

return 0
```

(b)

Figure 5.1: Example of dispatch tables: (a) a simple dispatch table, and (b) its use.

Sometimes this dispatch table is an array of structures that contain pointers to functions, as shown in Figure 5.1.

The way that such aggregates are allocated and manipulated often causes their points-to classes to be merged, yielding imprecise resolution of pointer references during analysis. The use of type casting, pointer arithmetic, and custom memory allocators are especially problematic. The resulting merges often cascade, yielding unacceptably conservative results. For example, if two separate dispatch tables become merged by the analysis, then the structures they contain become merged, and finally the fields within the structures are merged. Such collapsing of points-to classes not only results in overly conservative resolution of pointer references during data-flow analysis, but also during the computation of the program call-graph. As a result, the subsequent data-flow analysis can be both very inefficient and imprecise, since the analysis will traverse a large number of function calls that cannot actually occur during program execution.

Although some of these problems with points-to class merging can be overcome by using a context-sensitive points-to analysis, the analysis may then become too

	strong checking	weak checking
arguments	number of actuals must match number of formals	number of actuals 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, unless one is pointer and other is integer
qualifiers	ignored	ignored

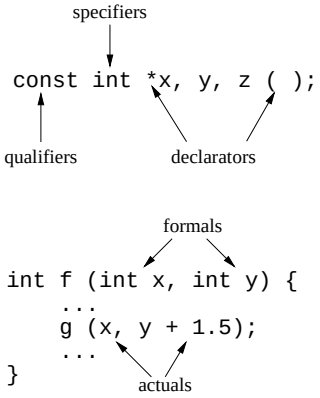


Figure 5.2: Prototype filtering rules for both strong and weak prototype filtering. The example code fragments illustrate the program components being checked.

expensive [Atkinson and Griswold, 1996]. Furthermore, the increase in precision may be small [Ruf, 1995] or may not yield substantially better data-flow information [Shapiro and Horwitz, 1997b].

5.2 Approach

Since our tool is designed for program understanding, we allow the user to provide both optimistic (i.e., “unsafe”) and conservative information. In tasks such as compilation or automatic parallelization, the meaning of the program must be preserved. However in program understanding, the tool user is attempting to gain knowledge about the system or provide reassurance of an assumption made about the system. As long as the tool user is readily aware that certain parameters may yield unsafe information, we feel the ability to provide optimistic information is justified.

We have developed several options for parameterizing (e.g., annotating [Koelbel et al., 1994]) the points-to analysis that the user of our program slicer may enable. Each has its own effect on the points-to analysis.

```
void (*p) ( );
int (*q) ( ), y;

int main ( ) {
  (*p) (1);
  (*q) (2, "a");
  (*q) (3, &y);
}

void f (int x) {
  y = x;
}

int g (int x) {
  return x;
}

int h (int x, void *p) {
  return x + *(int *) p;
}

int i (int x, char *p) {
  return *p + x;
}
```

Figure 5.3: A program fragment using function pointers.

5.2.1 Function prototypes

In many cases we found it too costly in time and space to compute sufficiently precise points-to sets for function pointers. Consequently, we turned to using type information to achieve better results. 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 5.2. Function prototypes provide additional typing information for static semantic checking by ensuring that the type 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 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.

For example, Figure 5.3 presents a small program using function pointers. Let us assume that the four functions, `f()`, `g()`, `h()`, and `i()`, have all been merged into

expression	not enabled	enabled
`(*p) (1);`	{f,g,h,i}	{f}
`(*q) (2, "a");`	{f,g,h,i}	{h,i}
`(*q) (3, &y);`	{f,g,h,i}	{h}

Table 5.1: Effect of strong prototype filtering, showing the points-to sets for each function call of Figure 5.3 with filtering not enabled and then enabled by the user.

the same points-to class and that both `p` and `q` point to this class, as shown in the second column of Table 5.1. By filtering on the prototypes of the function call and definition, the first function call in `main()` can only refer to function `f()` since the other three functions return an `int` and `p` is declared to return `void`. The second call can refer to either function `h()` or function `i()` since they both require two arguments and a string is assignable to the `void` pointer argument in function `h()`. The third call can only refer to function `h()` since a pointer to `int` is assignable to a `void` pointer, but not to a `char` pointer (i.e., string literal). The results with strong prototype filtering enabled are shown in the third column of Table 5.1.

5.2.2 Private memory allocators

Because large programs typically process lots of information, they can dynamically allocate several thousand objects. Since calling the standard C `malloc()` function for each object incurs an overhead, many large systems employ their own memory allocator. Implementing a private memory allocator is not difficult since C's own memory allocator, `malloc()`, is itself implemented in C. Typically, a private memory allocator is just a "wrapper" around calls to the underlying memory allocator such as `malloc()` that allocates larger blocks of memory and then doles them out in appropriately sized pieces, as shown in Figure 5.4(a). Another type of simple allocator, such as the `xmalloc()` function in our example programs, is one that merely calls `malloc()` and then checks the return value to see if virtual memory has been exhausted, as shown in Figure 5.4(b).

```
void *oballoc ( ) {
  static char *ptr, *end;

  if (ptr == end) {
    ptr = malloc (1024);
    end = ptr + 1024;
  }

  return p += 16;
}
```

(a)

```
void *xmalloc (int n) {
  void *ptr = malloc (n);

  if (ptr == NULL) {
    printf ("oops\n");
    abort ( );
  }

  return ptr;
}
```

(b)

Figure 5.4: Example private memory allocators: (a) an efficient memory allocator for sixteen-byte objects, and (b) a simple `xmalloc()` function from our example programs.

The use of private memory allocators can reduce the precision of points-to analysis. One method of modeling dynamically created storage is to treat each static call to `malloc()` as though it has its own heap, which is modeled as if it were a single large array of bytes from which objects are allocated. As a consequence, all pointers that are associated with a particular `malloc()` call site are treated as referencing the same memory address (assuming array indices are ignored). This approach, which we use, is simple to implement and often yields adequate precision [Steensgaard, 1996b]. For a program using the `xmalloc()` function described above, the program will contain several distinct calls to `xmalloc()`, but only one static call to `malloc()` (by `xmalloc()` itself). Thus, using `xmalloc()` rather than `malloc()` results in modeling memory as a single large, shared array, rather than several separate ones. All pointers to dynamically allocated memory are treated as referencing the same memory location. In effect, the points-to analysis is penalizing the programmer for writing efficient and modular code.

With the private memory allocator option, the user specifies the names of those functions that should be treated as if they were calls to `malloc()`. Each call site of the memory allocator is treated as if it returned the address of a temporary static variable, rather than all calls returning the address of the same variable.[1] This information may be

[1] Functions such as `free()` need to be treated similarly to avoid merging due to parameter passing.

```
a = xmalloc (10);
b = xmalloc (10);
x = malloc (10);
y = malloc (10);
```

(a)

points-to sets

not declared:	declared:
{a,b} {x} {y}	{a} {b} {x} {y}

(b)

Figure 5.5: Effect of declaring private memory allocators: (a) example code using the `xmalloc()` function of Figure 5.4, and (b) the resulting points-to sets based on whether the user has declared `xmalloc()` as a private memory allocator.

optimistic if the user is unsure which functions serve as memory allocators. The effect of using this option is to introduce more addresses to the points-to analysis, resulting in more points-to locations, as shown in Figure 5.5 in which the locations of `a` and `b` are now distinct just as the locations `x` and `y` are distinct.

5.2.3 Structure members

A typical C program uses structures quite heavily to model objects. A structure may contain pointers to other objects of different types. Since these objects are of different types, they are likely distinct. Although distinguishing structure members in points-to analysis can increase precision, sometimes the benefit is small and is not justified by the higher cost. In the worst case the analysis may require exponential time when structure members are distinguished [Steensgaard, 1996a].

To permit managing the time and space complexity of the analysis, our analysis distinguishes structure members only when chosen as an option by the user. Thus, references to `a.x` and `a.y` are normally treated as a reference to `a`. As a result, any objects pointed to by the `x` and `y` members are merged into a single points-to class.

When the user enables the structure members option, two such locations are not merged.[2] A structure assignment is treated as assigning the individual members. If

[2]The points-to analysis is similar to that described in [Steensgaard, 1996a], but assumes that adjacent structure members are infinitely far apart and thus does not take into account the size of an access during the analysis.

```
char a [10];
char *p, *q;
int i, j;

i = 1;
p = &a;
p [i] = 0;

j = (int) p;                /* j is just p */
q = (char *) i;             /* q is just i */

printf ("%d\n", p [i]);     /* same as *(p + i) */
printf ("%d\n", j [q]);     /* same as *(j + q) */
```

Figure 5.6: A program fragment showing commutativity of the array operator. Although the variable `j` is the integer and variable `q` is the pointer in the final array reference, it is `j`, and not `q`, that in fact carries the pointer information.

---

pointer arithmetic is performed on a structure pointer, then the members are “collapsed” (i.e., the points-to sets for all members are merged and the structure is thereafter treated as a single location) since a dereference through the generated pointer value may assign to any member or possibly multiple members. A variant structure or “union” type in C is considered to be a structure whose fields are already collapsed.

5.2.4 Strict arrays

In C, the array operator is commutative because array references are semantically equivalent to pointer addition, which itself is commutative. The expressions `a[i]` and `i[a]` are identical. However, the second form in which the pointer value appears within the brackets is generally not used. Normally, the points-to analysis must assume that this second form can be used. Thus, if the index `i` is used to index two distinct locations `a` and `b`, they become indistinguishable to the analysis since it assumes that `a` and `b` may be the indices and that `i` is the pointer value. (Use of the cast operator in C to override the type system makes this possible, as shown in Figure 5.6.)

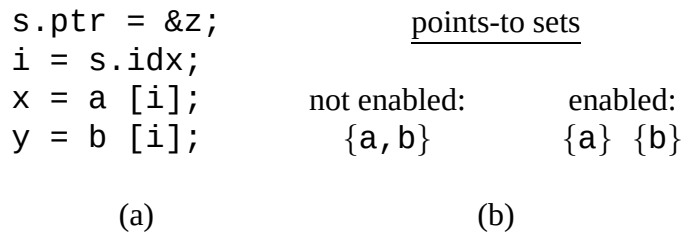


Figure 5.7: Effect of the strict arrays option: (a) example code showing accesses to two arrays with the same index, and (b) the resulting points-to sets based on whether the tool user has enabled the strict arrays option.

Although the programmer may not deliberately write code in this fashion, the points-to analysis may still detect such an occurrence. For example, if either operand of the array operator is a structure reference and either structure members have not been distinguished or the corresponding structure has been collapsed, then any integer and pointer members of the structure will have been merged, and the resulting operand may therefore be seen as carrying the pointer information. For example, in Figure 5.7 the two arrays `a` and `b` are indexed with the same variable, which is the result of a structure reference. However, the structure also holds a pointer value by the first assignment. If structure members are not distinguished or the structure has been collapsed (perhaps the structure is a `union` type), then the two arrays will be merged by the points-to analysis.

By enabling the strict arrays option, the user precludes this possibility, resulting in the two locations not being merged. Using this option may yield unsafe information unless the tool user is sure that the second form of array indexing is never used. However, a special case exists if `a` is declared as an array rather than a pointer. Since an array variable is constant and cannot be assigned, we can be sure that `i` is the index and that `a` is always the pointer value.

5.2.5 Combining parameters

Although each parameter can individually improve the precision of the points-to analysis, when combined the results are magnified. Continuing our example of GCC,

the array commutativity parameter prevents merging of separate arrays of structures, and the structure and prototype parameters distinguish the individual structure components. However, in some cases, the effects of one parameter will subsume the effects of another. In particular, we have found for function pointers that filtering the points-to classes using strong prototypes is the most beneficial (Chapter 8).

5.3 Conclusion

The use of pointers in large programs can have a significant impact on the precision of the points-to analysis. The use of private memory allocators, dispatch tables, and structures can decrease the precision of the analysis. To overcome the difficulties presented by these and other uses of pointers, we allow the tool user to parameterize the points-to analysis. By allowing the user to easily specify high-level parameterizations of the analysis, the precision of the points-to analysis can be substantially increased while imposing little burden on the tool user.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998]. The dissertation author was the primary researcher and author and the co-author of this publication directed and supervised the research that forms the basis for this chapter.

Chapter 6

Data-Flow Analysis in the Presence of Pointers to Locals

In the absence of either recursion or pointers to local variables, interprocedural slicing is simple and well-understood. However, for a language such as C that provides both of these features, traditional data-flow analyses may yield unsafe results. We first present the traditional data-flow equations and then describe how the analysis may be in error. Finally, we present our space-efficient data-flow equations that overcome these problems.

6.1 Background and Motivation

6.1.1 Data-flow equations

Figure 6.1 presents traditional data-flow equations for backward slicing. At each program point, D represents a data-flow set. At each assignment statement (Equations 6.1.3–6.1.5), some set of variables, $defs$, are defined and another set of variables, $uses$, are used. If some variable in $defs$ is also in D, then the killing definitions of $defs$ are removed from D and $uses$ are added to D. Otherwise, D remains unchanged. In Equation 6.1.5, we assume that all assignments through a pointer dereference are pre-

`return from f()`

$D_{exit} = D_{exit} \cup (D_i \cap S)$ (6.1.1)

`call to f()`

$D_i = (D_i - S) \cup (D_{entry} \cap S)$ (6.1.2)

`x := y`

if $x \in D_i$ then
$D_i = D_i - \{x\} \cup \{y\}$ (6.1.3)

`x := *p`

if $x \in D_i$ then
$D_i = D_i - \{x\} \cup *p \cup \{p\}$ (6.1.4)

`*p := x`

if $*p \cap D_i \neq \phi$ then
$D_i = D_i \cup \{p, x\}$ (6.1.5)

D = variables of interest
S = all global (static) variables
$*p$= points-to set of variable p

Figure 6.1: Traditional data-flow equations for slicing in the presence of recursion without pointers to local variables. Sets are subscripted with the program point to which they refer. Sets that are not subscripted are the same for all program points. The current statement has program point i. Unless otherwise noted, a set passes through a program point unchanged.

serving, since the points-to set may contain more than one variable, but only one variable is in fact updated. The remaining two equations (Equations 6.1.1 and 6.1.2) show how the D set is manipulated across function calls using the CFG shown in Figure 6.2. The S set contains all static (global) variables in the program and is used to partition a set into its local and global variables.[1]

6.1.2 Incorrectness of traditional equations

Figure 6.3 shows a small C program with a recursive function `f()`. Consider performing a backward program slice at the assignment to `y` in `f()`. Consequently, we are looking for an assignment to `x`, which is a local variable to `f()`. Proceeding backward through the function, the next statement examined is the recursive call to `f()`. When slicing into the recursive call, we need to remove local variables from the data-flow sets (Equations 6.1.1 and 6.1.2) [Knoop and Steffen, 1992]. This step is necessary

[1] We use the term *local variable* to mean an *automatic variable* in C. Similarly, the term *global variable* should be read as *static variable*. Since C overloads the use of the "static" keyword, we use the terms local and global variable instead.

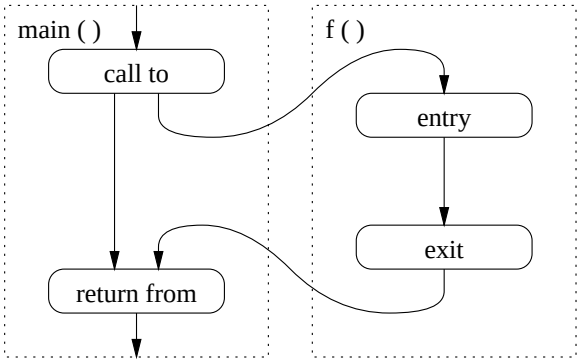


Figure 6.2: Example CFG showing program points relevant to function calls.

to avoid finding a definition of the same local variable but with a *different activation* in the recursive call. Continuing our example, the data-flow sets become empty after removing `x`, resulting in no further information being added to the slice by the recursive call. However, we have now erroneously excluded the last assignment in `f()` from the slice. This statement is clearly an assignment to `x`, although which activation of `x` we do not know. This is a *missing definition* of `x`. On the other hand, if we do not remove `x` from the data-flow set, then we can find false definitions of `x` because these definitions may in fact refer to other activations of `x`. If any of these definitions is a killing definition, then we have a *false kill* of `x` and our analysis is also in error.

6.2 Approach

The main difficulty with incorporating both recursion and pointers to local variables is that the two features require that local variables be treated in contradictory ways when slicing into a function call. To ensure correctness, local variables must be removed for the data-flow sets in the presence of recursion, but must remain in the data-flow sets in the presence of pointers to local variables. We introduce two new data-flow sets, N and P, in our equations, shown in Figure 6.4. The N set is used to solve the problem of a missing definition, and the P set it used to solve the problem of a false kill.

```
f ( ) {                          int y, *p;
  int x = 1;
                                 main ( ) {
  if (g ( ))                       f ( );
    p = &x;                        printf ("%d\n", y);
                                 }
  if (g ( )) {
    x = 2;
    f ( );                       g ( ) {
  }                                int z;

  y = x;                           scanf ("%d", &z);
  *p = 3;                          return z;
}                                }
```

Figure 6.3: Example program showing pointers to local variables in the presence of recursion. The global pointer variable `p` is conditionally assigned to the local variable `x` of function `f()`, which is recursive. As a result of the assignment, `p` may point to different activations of `x`, which introduces the possibility of a missing definition or a false kill.

6.2.1 Solving the problem of a missing definition

In the presence of recursion, a local variable must be removed from D in order to avoid finding a killing definition of the same variable but with a different activation. However, the local variable may be referenced through a pointer in a called function. In our approach, local variables are removed from D and placed into N of the called function (Equation 6.4.1c). This process is similar to the mapping and unmapping of nonvisible variables [Emami et al., 1994; Landi et al., 1993]. Consider an assignment made by dereferencing a pointer variable p (Equation 6.4.5). If the points-to set of p overlaps with N, then a local variable declared in another function has been defined. However, which activation of the variable that has been defined is unknown, and therefore the assignment must be treated as a preserving definition. The assignment statement should be added to the slice and the variables used at that statement added to D, but the variables defined are not removed.

`return from f()`

$$P_{exit} = P_{exit} \cup P_i \quad (6.4.1a)$$
$$D_{exit} = D_{exit} \cup (D_i \cap S) \quad (6.4.1b)$$
$$N_{exit} = N_{exit} \cup N_i \cup (D_i - S) \quad (6.4.1c)$$

`call to f()`

$$P_i = P_i \cup P_{entry} \quad (6.4.2a)$$
$$D_i = (D_i - S) \cup (D_{entry} \cap S) \quad (6.4.2b)$$

`x := y`

if $x \in P_i$ then
$$D_i = D_i \cup \{y\} \quad (6.4.3a)$$
else if $x \in D_i$ then
$$D_i = D_i - \{x\} \cup \{y\} \quad (6.4.3b)$$

`x := *p`

if $x \in P_i$ then
$$P_i = P_i \cup (*p - S) \quad (6.4.4a)$$
$$D_i = D_i \cup (*p \cap S) \cup \{p\} \quad (6.4.4b)$$
else if $x \in D_i$ then
$$P_i = P_i \cup (*p - S) \quad (6.4.4c)$$
$$D_i = D_i - \{x\} \cup (*p \cap S) \cup \{p\} \quad (6.4.4d)$$

`*p := x`

if $*p \cap (D_i \cup P_i \cup N_i) \neq \phi$ then
$$D_i = D_i \cup \{p, x\} \quad (6.4.5)$$

N = *nonlocal local variables of interest*
D = *variables of interest with killing defs*
P = *locals of interest with preserving defs*
S = *all global (static) variables*
$*p$ = *points-to set of variable p*

Figure 6.4: Our data-flow equations for slicing in the presence of recursion and pointers to local variables. Sets are subscripted with the program point to which they refer. Sets that are not subscripted are the same for all program points. The current statement has program point i. Unless otherwise noted, a set passes through a program point unchanged.

The N set models the transitive closure of the program stack, but only for local variables of interest rather than for all local variables. When a function is called, the local variables of interest to the caller are added to N of the called function along with the caller's N set (Equation 6.4.1c). Since N only contains *nonlocal* local variables of interest, it need only be examined in statements containing an assignment by means of a pointer dereference (Equation 6.4.5).

6.2.2 Solving the problem of a false kill

If a local variable is referenced out of scope by means of a pointer dereference, we cannot be certain which activation of the variable is actually used. To be safe, we

must therefore assume that all possible activations are referenced. The activations of a local variable can be thought of as an array, with the stack pointer referring to the last element of the array. If a local variable is referenced out of scope, we do not know the array "index" and so must assume all activations are referenced. Thus, a local variable referenced out of scope is treated just like an array—any definition is always a preserving definition. In our approach, the P set keeps track of these variables. Using the S set, the variables referenced by means of a pointer dereference are partitioned into its local and global variables (Equations 6.4.4a–6.4.4d). The local variables are added to P, and the global variables are added to D. At an assignment statement, if the variable being defined is present in P, then the statement is included in the slice and the corresponding variables used are added to D, but the variable is not removed. Consequently, Equation 6.1.3 now requires two cases (Equations 6.4.3a and 6.4.3b), as does Equation 6.1.4 (Equations 6.4.4a-b and 6.4.4c-d).

The P set contains those local variables that have been "demoted" to have only preserving definitions. The demotion occurs only if the variable is referenced out of scope by means of a pointer dereference. Once a local variable is added to P, it is never removed. Finally, the demotion propagates through all (backward) reachable program points—P is propagated into a called function (Equation 6.4.1a) and also into any calling function (Equation 6.4.2a).

6.2.3 Correctness of our equations

To provide insight into the correctness of our equations, we can examine how the equations are transformed if pointers to local variables are not allowed. In this case, the points-to set of a variable p contains only global variables. Consequently, no variables are added to the P set in Equations 6.4.4a and 6.4.4c. Since these are the only equations in which individual variables are added to P, the P set is therefore always empty and Equations 6.4.1a, 6.4.2a, 6.4.3a, 6.4.4a, 6.4.4b, and 6.4.4c can be eliminated. Also, since the N set contains only local variables and the P set is always empty, both the N and P sets can be eliminated in the conditional test for Equation 6.4.5. Therefore,

without pointers to local variables, the equations reduce to the more familiar data-flow equations for backward program slicing given in Figure 6.1.

6.3 Conclusion

Traditional equations for interprocedural slicing are incorrect if pointers to local variables are used in recursive programs. We demonstrated that the problem is due to either a false definition or a missing kill. We presented new data-flow equations, derived from the traditional equations, which solve these problems. In Chapter 7, we discuss an efficient implementation of our equations.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998]. The dissertation author was the primary researcher and author and the co-author of this publication directed and supervised the research that forms the basis for this chapter.

Chapter 7

Implementation

Because of the high space and time demands of whole-program analysis, special attention to implementation details are required. In this chapter we present several aspects of the implementation of the data-flow framework of our program slicing tools that was not presented in earlier chapters.

7.1 Block Visitation Algorithms

To perform data-flow analysis, a compiler or program understanding tool propagates the computed data-flow information along the edges of the constructed CFG. The data-flow facts along the incoming edges of a node are combined into a single set that is then transformed according to the data-flow properties of the node. The resulting set is then propagated along all output edges of the node. If the CFG is reducible (e.g., the program does not contain any unstructured jump statements), then the data-flow information can typically be propagated fully in a single pass over the CFG [Aho et al., 1986]. Otherwise, an iterative algorithm must be used that propagates the data-flow information until no further changes to the data-flow sets occur. One way to achieve good performance in an analysis tool is to design and implement an iterative algorithm that converges quickly or that has a short iteration time. Chapter 4 presents high-level techniques for achieving both goals.

```
changed := true

while changed do
  changed := false
  for each block B do
    old := out [B]
    visit (B)
    if old ≠ out [B] then
      changed := true
    end if
  end for
end while
```

(a)

```
worklist := {start}

while worklist ≠ φ do
  worklist := worklist - {B}
  old := out [B]
  visit (B)
  if old ≠ out [B] then
    for P in pred [B] do
      worklist := worklist ∪ {P}
    end for
  end if
end while
```

(b)

Figure 7.1: Example pseudocode for vistation algorithms: (a) the iterative search algorithm, and (b) the worklist algorithm.

Since both the MUMPS and C languages allow unstructured control-flow, an iterative algorithm is needed by our program slicing tool. The visitation order of the nodes does not affect the correctness of the algorithm, so long as the data-flow information is fully propagated along all edges until no more changes occur to the data-flow sets. However, the visitation order can greatly impact the performance of the algorithm. Two common visitation algorithms are used, as shown in Figure 7.1.

Iterative search algorithm: In the iterative search (i.e., "for each basic block") algorithm (Figure 7.1a), each block (i.e., CFG node) is visited once. If any changes have occurred, then each block is visited once again. This process repeats until no further changes occur to the data-flow sets. Typically, a depth-first or breadth-first search of the CFG is used to visit all blocks exactly once in an iteration, with depth-first search usually resulting in fewer required iterations [Aho et al., 1986].

Worklist algorithm: In the worklist algorithm (Figure 7.1b), the blocks (i.e., nodes) to be visited are placed on a worklist, which is typically implemented using a stack or queue, with a stack implementation usually resulting in fewer block visits. A block is

```
i = 0;

while (i < n) {
    if (sum > 20)
        j = j + 1;

    sum = sum + i;
    i = i + 1;
}

return sum;
```

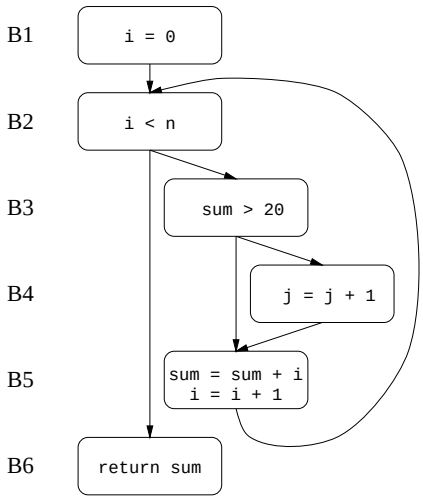


Figure 7.2: A program fragment and its annotated CFG.

removed from the worklist and visited. If any changes occur to the data-flow sets of the block, then all predecessors of the block (successors for a forward data-flow analysis) are placed on the worklist. The algorithm repeats until the worklist is empty.

Figure 7.2 shows a program fragment and its associated CFG, annotated with block numbers. Consider starting a backward program slice at the return statement located at block *B6*. An example visitation order for *one iteration* of the iterative search algorithm would be *B6*, *B2*, *B1*, *B5*, *B4*, and *B3*. For the worklist algorithm, a *complete visitation* order might be *B6*, *B2*, *B5*, *B4*, *B3*, *B2*, *B5*, and *B1*. Unlike the iterative search algorithm, the worklist algorithm can visit a block many times before visiting other blocks. For example, blocks *B2* and *B5* are visited twice before block *B1* is ever visited.

In a whole-program analysis tool, the space required to perform the analysis must be reduced in order to obtain good performance. We implemented both the iterative search and worklist algorithms for our slicing tool for C programs. The algorithms required equal implementation time. We found that the worklist algorithm results in approximately 50% less block visits than the iterative search algorithm. However, the iterative search algorithm can be implemented such that significant space can be saved, while the worklist algorithm to our knowledge cannot.

7.2 Reclamation of Data-Flow Sets

Although the worklist algorithm requires substantially fewer node visits to converge than the iterative search algorithm, we chose to use the iterative search algorithm because it has the advantage that the data-flow sets for each block can be reclaimed during an iteration. The iterative search algorithm guarantees that each block in the program is visited exactly once before any block is revisited. Recall that the input data-flow set of a block is the confluence of the data-flow sets on all incoming edges. In a backward data-flow analysis, once all predecessors of a block have been visited then the data-flow set for the block itself will no longer be needed. Consequently, the data-flow set can be deallocated. If a depth-first search is used to visit all blocks, then the maximum number of data-flow sets that need to be allocated at any one time is proportional to the width of the CFG, resulting in substantial savings in space.

In our first implementation of this reclamation approach, we found that it was cumbersome to keep a reference count on each block to keep track of the number of its predecessors that had been visited. Consequently, we chose a simpler implementation without reference counts: once the depth-first search of a called function is complete and all blocks have been visited, the data-flow set of each block (other than the entry and exit blocks) is deallocated if all predecessors of the block were visited after the block itself was visited. Therefore, data-flow analysis with reclamation can easily be done as a two-step process, in which the first step visits each block and also stores the visitation order (Figure 7.3a), and the second step reclaims the data-flow sets of those blocks that meet the mentioned criteria (Figure 7.3b). Slightly more data-flow sets remain active at any time than are minimally needed, but the implementation is much simpler since reference counting is not needed.

Unfortunately, data-flow set reclamation is not possible to our knowledge using the worklist algorithm. Whereas the iterative search algorithm is driven strictly by the control-flow properties of the program and has the inherent property than all blocks will be visited exactly once during each iteration, the worklist algorithm is driven more

```
function dfs (B)
  ordered := ordered · [B]
  visited := visited ∪ {B}
  old := out [B]
  visit (B)

  if old ≠ out [B] then
    changed := true
  end if

  for P in pred [B] do
    if P ∉ visited then
      dfs (P)
    end if
  end for
end function
```

(a)

```
for B in ordered do
  before := before ∪ {B}
  if B ≠ entry and B ≠ exit then
    reclaimable := true

    for P in pred [B] do
      if P ∈ before then
        reclaimable := false
      end if
    end for

    if reclaimable then
      delete (out [B])
    end if
  end if
end for
```

(b)

Figure 7.3: Example of data-flow set reclamation: (a) a depth-first search algorithm that also stores the visitation order, and (b) a reclamation algorithm that uses the visitation order to safely reclaim unneeded data-flow sets. The notation $a \cdot [x]$ indicates the concatenation of the list a with the single-element list containing x.

---

by the data-flow properties of the program. As a result, data-flow information is propagated more effectively, but blocks are visited in an unpredictable order and a block may be visited many times before all of its predecessors are visited. Therefore, a reference counting approach is not easily implemented.

Attempting to use our simpler reclamation implementation with the worklist algorithm also has performance problems. A small change in the data-flow set of the calling function may require that the called function be visited again, in which case the data-flow sets for the entire function need to be recomputed and reconverged. This type of nested convergence could require exponential time. Although we did implement this type of reclamation, we found that the number of block visits was substantially larger than the number of visits needed by the iterative search algorithm (regardless of whether reclamation was performed).

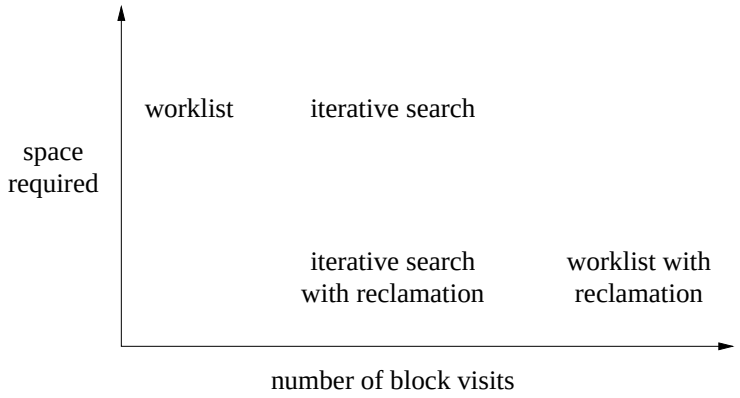


Figure 7.4: A comparison of our block visitation algorithms.

Figure 7.4 summarizes the characteristics of our different visitation algorithms with and without data-flow set reclamation. Although the worklist algorithm requires fewer block visits, for large programs the iterative search algorithm with reclamation is a better choice since it saves considerable space.

7.3 Data-Flow Set Implementation

In Chapter 6 we presented data-flow equations for correctly computing a backward program slice for recursive C programs containing pointers to local variables. If implemented naively, our data-flow equations requires three data-flow sets per block (i.e., there is one D, P, and N set per block). Given that GCC has 238,000 symbols and 120,000 blocks, a bit-set implementation of data-flow sets would require over 10 GB (238,000 *symbols* $\times$ 120,000 *blocks* $\times$ 3 *sets / block* $\times$ 1 *bit / symbol* $\div$ 8 *bits / byte*) of space. (A bit-set is implemented by consecutively mapping the elements of the input set onto the natural numbers. Each number represents the bit position in a bit-vector. An element is a member of the set if and only if its corresponding bit in the bit-vector is set. A bit-vector representation allows set operations such as union and intersection to be implemented efficiently using logical bit-wise operations. However, such operations can only be performed across bit-sets with identical mappings (bit-numberings).)

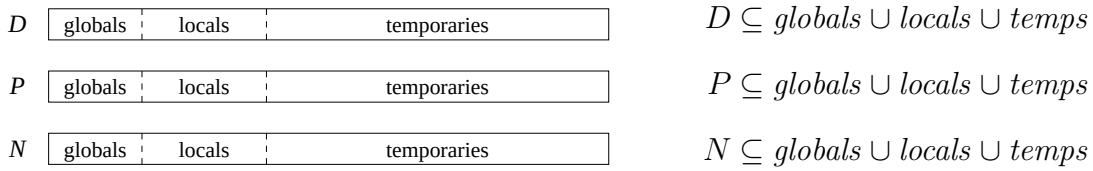


Figure 7.5: A simple implementation of the data-flow sets for our equations. The space required is approximately 3.4 GB for GCC even if the N and P sets are flow-insensitive.

However, since N does not change while slicing a function, a single N set can be used for all blocks of the function. Examining the P set, we see that it is nondecreasing in size since variables are only added to the set and never removed, unlike the D set. This fact suggests that P can be made flow-insensitive with little loss in precision. Consequently, we chose to also use a single P set for all blocks of a function. This decision sacrifices precision slightly in favor of performance. Using this implementation, the space requirements for GCC are reduced to approximately 3.3 GB (10 GB $\div$ 3), still an unacceptable amount of space. This simple implementation is shown in Figure 7.5.

An analysis of the bit-sets revealed that they are typically very sparse. Examining our equations, we see that P and N contain only local variables, while D contains local variables, global variables, and generated temporaries. Also, temporaries cannot be the target of a pointer and therefore cannot be referenced out of scope.[1] We therefore decided to partition the bit-sets into three distinct classes: global variables, local variables, and temporaries for each function, as shown in Figure 7.6. The D set now consists of three bit-sets, but requires space to store only all the global variables, local variables, and the *maximum* number of temporaries *per function*. If we assume for simplicity that the 220,000 temporaries are evenly distributed among GCC's 2,300 functions, the space requirements are reduced to approximately 60 MB, which is acceptable.

[1]There are a few cases where temporaries can be the target of a pointer such as when a structure is returned from a function. In these cases, we introduce a new type of temporary variable called a *special*. The introduction of specials allows us to treat the vast majority of temporaries as though they could not be the target of a pointer.

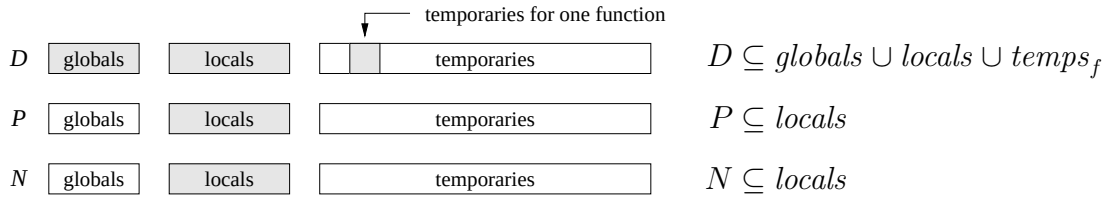


Figure 7.6: A better implementation of the data-flow sets for our data-flow equations. The sets are partitioned into their three distinct classes. Only the shaded areas of a set are actually used at any given time.

The partitioning also improves algorithmic performance and eases implementation. The data-flow equations of Figure 6.4 require that the D and P sets be partitioned into their local and global variables components. Logically, this partitioning is done using set intersections and differences. With these components maintained as separate sets, the partitioning operations are trivial. For example, rather than computing $D \cap S$ to retrieve the global variables of D, only the set of global variables of D (symbolically `D.globals`) need be retrieved, thereby changing an $O(n)$ operation into an $O(1)$ operation.

Examining the P and N sets in even greater detail, we see that they can contain only local variables that are pointed to by some pointer variable. Since there are far fewer of these *target locals* than local variables that are not pointed to by some variable (*nontarget locals*), the locals of the P and N sets can be further partitioned into two classes to save space, as shown in Figure 7.7. However, examining our data-flow equations, we see that the local variables of the P and N sets must be operated on in conjunction with those of the D sets (Equations 6.4.1c and 6.4.5). Therefore, *explicitly* partitioning the local variables into two classes would complicate these operations. Such an implementation of the data-flow equations would be complicated by the need to combine the sets of target and nontarget locals into one set for any operation involving the locals from the D set. Given that all data-flow are implemented using bit-sets, this process could be complicated if the various sets have different bit-numberings.

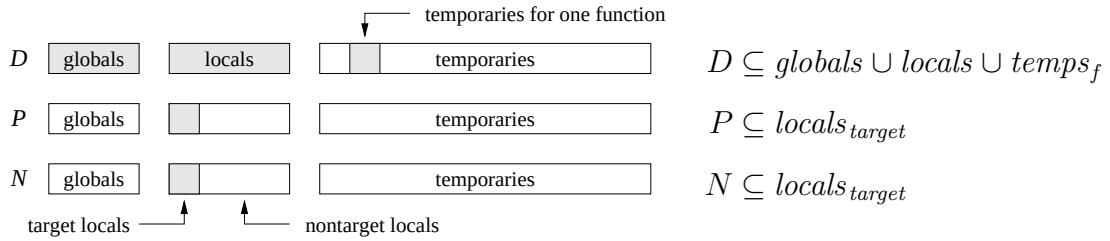


Figure 7.7: Our final implementation of the data-flow sets for our data-flow equations. The local variables of the P and N sets are further separated into target and nontarget locals. The slicing tool assigns the target locals lower bit-numbers than the nontarget locals to ensure "packing" of the bit-sets.

Rather than partitioning the local variables into two distinct sets, we elected to keep them as one set. However, since the points-to analysis is performed prior to data-flow analysis (Chapter 3), we know which local variables are target locals and which are nontarget locals. We can therefore easily ensure that the target locals are assigned lower bit-numbers than the nontarget locals. This numbering ensures that the target locals are "packed" at the start of the bit-sets, giving us the space savings we desire without the implementation complexities of splitting the locals into two bit-sets. Because programs often contain few target locals, the space allocated for the P and N sets is negligible as a result of our aggressive implementation. Consequently, the space required for the data-flow sets of our new equations is approximately the same space required for the sets of the more traditional equations given in Figure 6.1.

7.4 Control-Flow Dependencies

Program slicing requires the computation of control-flow dependencies as well as the computation data-flow information. For example, a backward program slice is defined as the set of all statements that might affect the value of a specified variable. Consequently, if the execution of a statement included in the slice is dependent upon the execution of another program statement (e.g., the included statement is within a conditional), then the variables used at the controlling statement must also be included

in the slicing criteria. We therefore need to determine which statements are control-dependent upon which other statements in the program.

We compute control-flow dependencies by computing the dominance frontiers of the reversed CFG [Cytron et al., 1991]. Computation of the dominance frontiers in turn requires the computation of the dominators of the flowgraph [Lengauer and Tarjan, 1979]. Both computations can be performed in linear time in the size of the flowgraph.

7.5 Conclusion

Because whole-program analysis imposes such high time and demands, special attention to implementation details is required. In particular, the implementation of the data-flow analysis can significantly affect a tool's time and space requirements. We presented different block visitation algorithms and showed that use of the iterative search algorithm is more appropriate in a whole-program analysis tool because it facilitates data-flow set reclamation. We also showed an efficient implementation of our data-flow equations given in Chapter 6.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998]. The dissertation author was the primary researcher and author and the co-author of this publication directed and supervised the research that forms the basis for this chapter.

Chapter 8

Evaluation and Results

To evaluate the time, space, and precision characteristics of our approach, we implemented program slicers based on our ideas for the C and MUMPS programming languages and measured their performance. The measurements given in this chapter are a representative subset of the complete data, which is given in Appendix A.

MUMPS is an interpreted programming language with a BASIC-like syntax, reference parameters, and dynamic scoping. Our slicing algorithm for MUMPS correctly handles dynamic scoping by treating each variable reference as a pointer dereference to any of the reaching variable declarations. In addition to analyzing CHCS, whose basic statistics appear in Table 1.1, we also analyzed COMPLY, a 1,000 line compliance checker for CHCS, also written in MUMPS.

We analyzed three large C programs whose basic statistics appear in Table 1.2. GCC refers to the `cc1` program of the GNU C compiler [Stallman, 1991], version 2.7.2 for SunOS 4.1.3; EMACS refers to the `temacs` program of the GNU Emacs editor [Stallman, 1993], version 19.34b for SunOS 4.1.3 without window system support; BURLAP refers to the `burlap` program of the FElt finite element analysis system [Gobat and Atkinson, 1994], version 3.02 for SunOS 4.1.3.

Our slicer for C programs correctly handles functions with a variable number of arguments and the effects due to library functions. Library functions are handled by providing a skeleton for each function that correctly summarizes its effects. Our

current implementation does not inline library functions, which is a common technique for increasing precision by adding one level of context-sensitivity. Signal handlers and the `longjmp` and `setjmp` functions are not handled.

For all slices of the C programs, we tried to choose variables that might be selected by a programmer during debugging. For the MUMPS programs, the slicing criteria were chosen to produce a reasonable distribution of slice sizes, not to be representative of typical slices, due to our lack of familiarity with the actual programs, and therefore conclusions should not be drawn about slicing itself, only about our techniques. Also, for both the C and MUMPS programs, the slices computed do not slice into the callers of the function in which the slice is initiated; consequently, the slices are akin to slicing on statements in the main procedure, which has no callers [Harrold and Ci, 1998].

8.1 Hypotheses

To test our claims about the value of demand-driven computation, discarding, precision control, and customizable termination, we performed several backward slices using our program slicers. We expected that our demand-driven techniques would reduce the time and space to be a function of the size of the slice, allowing the computation of slices that were not possible before. However, we expected the basic algorithmic cost of slicing would still be high, giving us an opportunity to evaluate the potential of the other features.

For the slices of C programs, it was our expectation that the time and space requirements would be acceptable, given our use of a near-linear time points-to analysis, our aggressive approach at implementing our data-flow equations, and piggybacking construction of the CFG with the computation of the points-to sets. We also expected that the parameterization of the points-to analysis would significantly increase the number of points-to classes, and thereby also increase the precision of the subsequent data-flow analysis. By filtering the points-to classes for function pointers based on their computed prototypes, we also expected the average number of functions called by means

of a function pointer to decrease, improving the precision of the call-graph and hence the subsequent data-flow analysis. With the exception of distinguishing structure members, which is known to take worst-case exponential time and space, we anticipated that the parameterizations of the points-to analysis would have little effect on its time and space requirements. Finally, we projected that combining parameters could significantly magnify the effects of the individual parameters.

The times reported do not include the time required to compute and write the call-graph to disk, since it is only recomputed when a file of the program to be sliced is changed (Chapters 2 and 3). For CHCS, this information is computed in 8.0 minutes on a SparcStation 10 and occupies 1.1 MB of disk space. For our largest C program, GCC, the information is computed in only 52 seconds on an UltraSparc 2 and occupies similar space (see the footnote on Page 5 for details on the gathering of results).

8.2 Demand-Driven Computation and Discarding

Figure 8.1 presents the statistics for a range of slices of CHCS. Figure 8.1a shows that the times and sizes of the slices appear to be related quadratically. Figure 8.1b shows that the space required appears to be linearly related to the slice size. These results indicate that we have met our goal of having the cost of the analysis be a function of the result's size, rather than the size of the entire program (Chapter 2).

Because the slices did not exhaust real memory—much less virtual memory—the role of discarding did not come into play. However, a separate set of measurements of slicing without discarding indicates that the cost of discarding the AST was insignificant. Recomputing slices without AST discarding results in smaller slices being a few percent faster, and larger slices being a few percent slower.

The results for the largest slices of the C programs that we performed are also presented in Figure 8.1. The time given in the table is the time necessary to perform only the slice, not to compute the points-to information, which is given in Table A.5. We believe that the time and space requirements are acceptable for a large program such

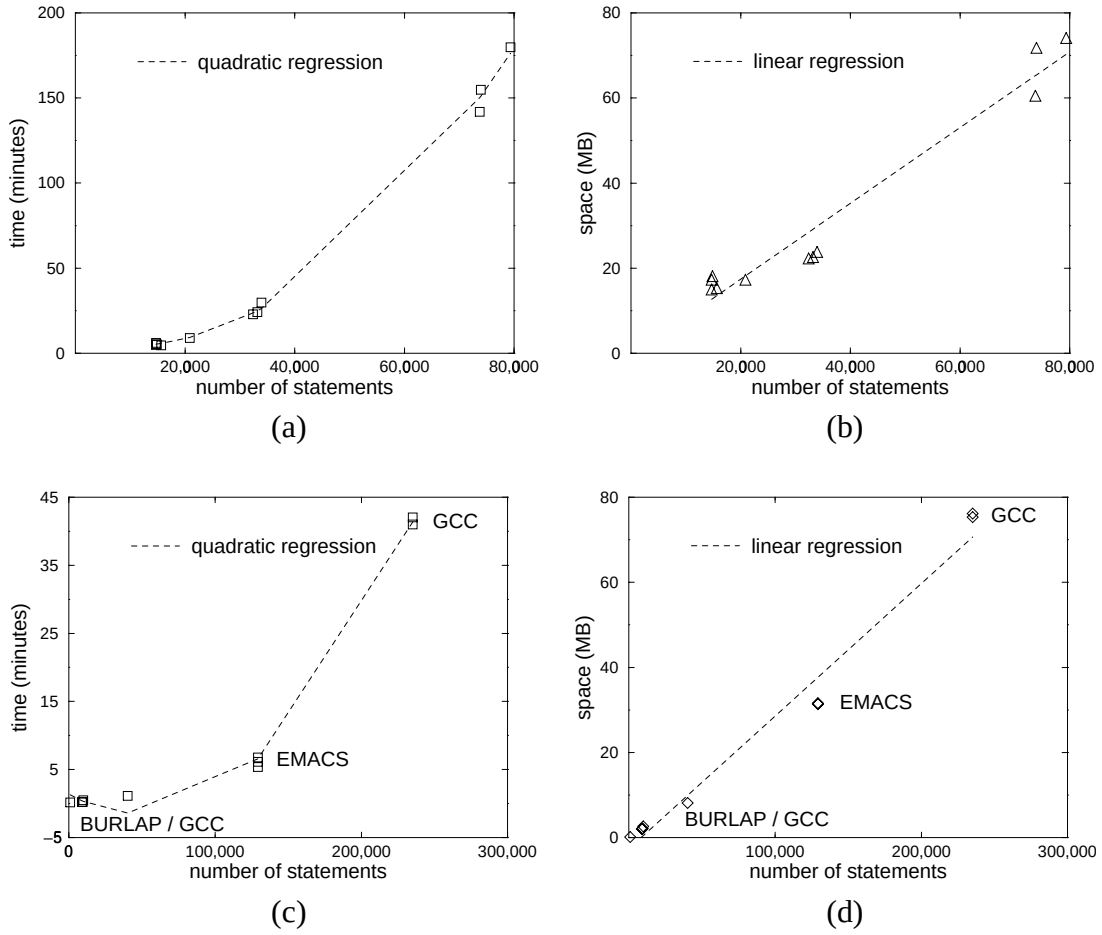


Figure 8.1: Statistics for different slices: (a) and (b) slices of CHCS with a single context per procedure, (c) and (d) slices of the three C programs using the *strong prototypes* option (Chapter 5). The complete measurements are given in Figures A.1 and A.2.

---

as GCC. For smaller programs, such as BURLAP, the slicer performs extremely well. These results indicate that slicing large programs is feasible using our approach.

8.3 Context-Depth Sensitivity

By allowing the user to control the context-depth of the data-flow analysis, we allow the user to balance the trade-offs between precision and performance (Chapter 4). Computing program slices at a high context-depth may yield better results, but at the

	criteria	depth	contexts	time	space	size of slice
COMPLY	(*COMBLK:14,*{*ERRTYP*})	1	36	0.030	0.37	780
		2	74	0.058	0.43	774
		3	121	0.092	0.55	769
		∞	602	0.484	1.42	769
CHCS	(*DIC:38,*{*DUOUT*})	1	248	5.3	19.6	14,711
		2	488	10.3	23.1	14,532
		3	2,553	121.0	53.6	14,530

Table 8.1: Statistics at different context-depths for the two MUMPS programs. Time is given in minutes and space in megabytes. The symbol ∞ is used to indicate an unbounded context-depth.

cost of an unacceptably long running time. On the other hand, computing slices at a low context-depth may not be sufficiently precise.

Table 8.1 contains statistics for one slice COMPLY and one slice of CHCS, at different context-depths. In each slice, the number of calling contexts increases rapidly with the context-depth, significantly increasing the time and space required. However, in the slices of COMPLY, a low context-depth yields a program slice equivalent to a program slice obtained at an unbounded context-depth. Although the resulting context-graphs differ, unbounded context-depth does not improve the results. The slices of CHCS show that as the context-depth increases, at first there is an appreciable decrease (1%) in the number of statements in the slice. However, an additional increase of the context-depth yields little improvement. These slices suggest that a high context-depth may be unnecessary to obtain a precise slice. They also support our hypothesis that a low context-depth slice is usually several times less costly than a higher one, suggesting that there is little extra cost to the tool user in performing a low context-depth slice first, on the hope that the result will adequately answer the tool user's question.

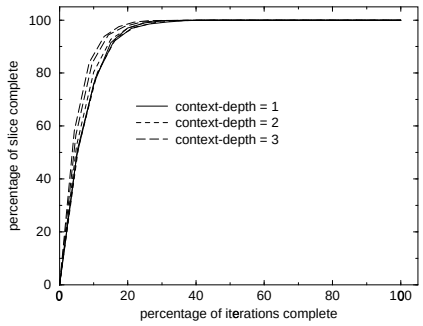
Table 8.2 presents statistics for one slice of BURLAP and one slice of GCC, at different context-depths. These slices exhibit the same behavior as the slices of the MUMPS programs—increasing the context-depth yields a small decrease in the number of statements in the slice and a large increase in the time and space required. For all

	criteria	depth	time	space	size of slice
BURLAP	(*arith.c:145,*{*type_error*})	1	0.20	2.0	8,849
		2	1.24	5.4	8,824
		3	3.56	17.5	8,680
GCC	(*unroll.c:3085,*{*const0_rtx*})	1	0.46	2.6	9,702
		2	4.20	12.9	9,574
		3	29.98	71.0	9,556

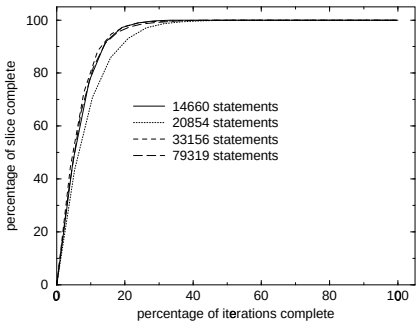
Table 8.2: Statistics at different context-depths for two C programs. Time is given in minutes and space in megabytes. The points-to analysis was performed using the *strong prototypes* option (Chapter 5).

but the smallest slices (i.e., less than 1,000 statements), we were unable to compute a slice at unbounded context-depth because of the high time and space requirements. For example, several slices of GCC failed to complete the first iteration after 12 hours. For other slices performed at a moderate context-depth (e.g., 2 or 3), the space requirements did not exceed the virtual memory capacity of the machine, but did exceed the main memory capacity. As a result, the program slicer spent most of its time moving data within the virtual memory hierarchy. These slices illustrate a disadvantage—poor reference locality—of the iterative search algorithm that was used (Chapter 7). However, without the use of this algorithm, data-flow set reclamation is not possible, and without reclamation, the space requirements would exceed the virtual memory capacity.

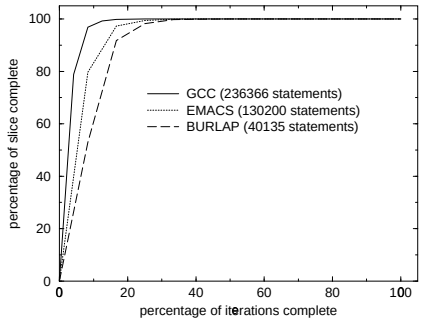
For most slices, we were unable to determine the context-depth that would result in a program slice equivalent to that obtained using an unbounded context-depth, as we were with COMPLY, since the space required exceeds the virtual memory capacity of the machine—computing a slice at a context-depth of four of CHCS exceeded the 600 MB available. The inability to compute slices at high context-depths of large programs is not unexpected. If the call-graph has n nodes, then the depth-1 context-graph has $O(n)$ nodes. In the worst case, each increase in context-depth can increase the number of nodes in the context-graph by a factor of n (i.e., a depth-2 context-graph can have $O(n^2)$ nodes, a depth-3 context-graph can have $O(n^3)$ nodes, etc.).



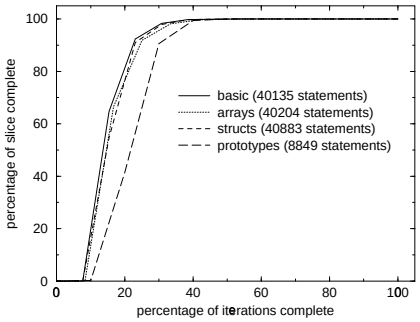
(a) CHCS: varying context-depth



(b) CHCS: varying size of slice



(c) C: varying programs



(d) BURLAP: varying parameterizations

Figure 8.2: Algorithmic convergence: (a) slices of CHCS at different context-depths, (b) several different slices of CHCS at a single context-depth, (c) a large slice of each C program, and (d) a single slice of BURLAP at different parameterizations of the points-to analysis.

8.4 Algorithmic Convergence

Even if a slice is computed at a single context-depth, the running time may still be unacceptable. For example, some program slices of CHCS in Figure 8.1 require more than two hours to complete, which for some uses may be unacceptable. It is our belief that the tool user should be able to interrupt the program slicer and examine the intermediate results (Chapter 4).

Our results show that the majority of statements in a slice are obtained after the first few iterations. Figure 8.2 depicts the convergence properties of our program

slicing algorithm by plotting the size of the slice at each iteration.[1] These figures show that approximately 80% of the statements in the slice are obtained within the first 20% of the iterations. Figure 8.2a presents data for two slices of CHCS at different context-depths, illustrating that the rate of convergence appears to be independent of the context-depth. Figure 8.2b presents data for several sizes of slices of CHCS, illustrating that the convergence rate also appears to be independent of the slice size. Figure 8.2c shows data for three large slices of the C programs. Finally, Figure 8.2c shows data for a single slice of BURLAP with different parameterizations of the points-to analysis. For these figures, the convergence rate also appears to be independent of both the program sliced and the choice of parameterization of the points-to analysis. This data seems to confirm our belief that the tool user can use customized termination of the slicer to substantially reduce the number of iterations of an analysis, independent of the context-depth and slice size.

8.5 Points-To Analysis Parameterization

8.5.1 Construction of points-to sets

In Chapter 5, we demonstrated that the ways in which pointers are used in large systems can hinder the points-to analysis's ability to perform an analysis that is sufficiently precise for the subsequent data-flow analysis. Our solution to overcoming this problem, without adding algorithmic complexity, is to allow the tool user to parameterize the points-to analysis.

Figure 8.3 presents statistics for performing a points-to analysis of our three example programs. When describing the parameterizations of the points-to analysis, the following titles are used: *none*—no parameterization, *mallocs*—private memory allocators are specified, *arrays*—array operator is not commutative, *structs*—structure members are distinguished, and *ideal structs*—structure members are distinguished, but structures are never collapsed. The last parameterization is extremely optimistic, but

[1] The data are normalized to percentages so that slices of different sizes and iterations can be compared.

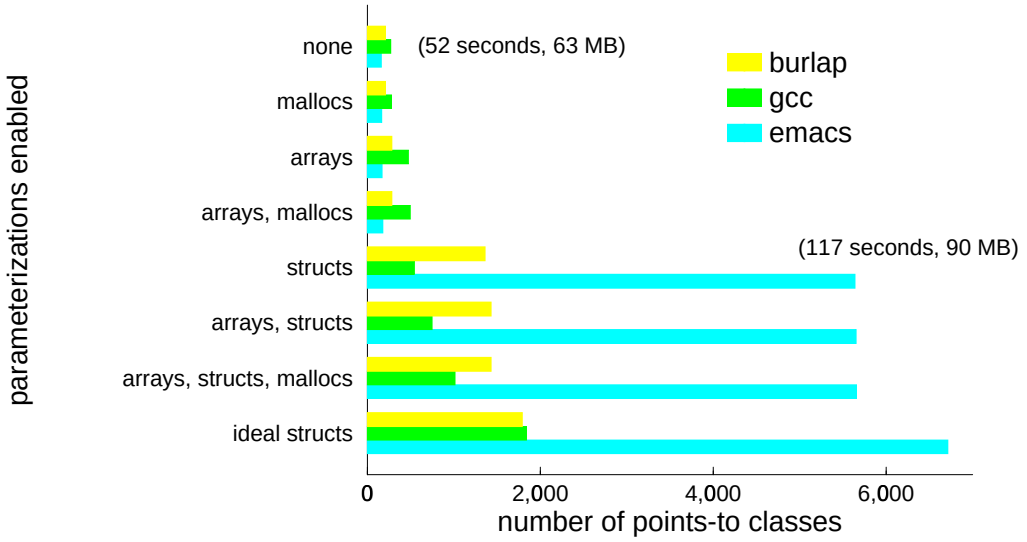


Figure 8.3: Effect of parameterization on points-to classes. The statistics shown in the graph are for GCC. The complete measurements are given in Table A.5.

---

provides a best-case upper bound on the number of points-to classes. Use of the *prototype* option (shown as *weak* or *strong*) is not included in this table, since the option does not change how the points-to classes are computed, but rather filters the classes once they are computed.

When comparing the various options, it is important to realize that they affect the points-to analysis in different ways. For example, the *structs* and *mallocs* options actually introduce additional locations to the points-to analysis, and with the exception of collapsing structures, do not affect how the points-to classes are merged. In contrast, the *arrays* option does not add any locations, but merely prevents unnecessary merging of the classes. However, the options are not necessarily orthogonal. Should two or more options be combined, the number of resulting additional classes may in fact be less than the sum of the number of additional classes introduced by using the options separately.

The results of the *structs* and *ideal structs* options cannot be compared directly with those of the other options. When structure fields are distinguished by the points-to analysis, an extended version of the analysis is actually being performed over

a *different* representation of the program. In particular, unless structure members are being distinguished by the points-to analysis and slicing algorithm, references to structure members are excluded from the CFG for performance reasons. Also, the *ideal structs* option is extremely optimistic, which explains how the number of classes can actually decrease when it is combined with other options. Finally, the ordering of statements in the program can greatly impact the performance of the points-to analysis if members are distinguished. For example, if all members of a structure are first referenced, and later pointer arithmetic is performed on the structure, then the structure must be collapsed and the points-to classes for all members must be recursively merged. In contrast, if the pointer arithmetic is done first, then the structure is first collapsed, requiring little time, and all later references to the structure members refer to the same points-to class, since the structure is now treated as a whole. However, the *structs* and *ideal structs* options do provide insight into whether distinguishing structure members has an beneficial effect on the analysis, given the more complex implementation and additional running time required.

As expected, enabling the *mallocs* and *arrays* options do not have an appreciable effect on the performance of the analysis. Yet, these options do increase the number of points-to classes, making them generally beneficial options. However, the degree of effectiveness varies significantly. The *arrays* option improves the results significantly for GCC (by 76%) and BURLAP (by 35%), but only slightly for EMACS (by 6%), probably because it has few occurrences of the array operator. In contrast, the *mallocs* option improves the results slightly for both GCC (by 4%) and EMACS (by 2%), but not at all for BURLAP, because it contains few references to private memory allocators.

Using the *structs* option increases both the number of points-to classes and the running time, making it a good choice for some programs, but not for others such as GCC. For both EMACS and BURLAP, the number of points-to classes increased by an order of magnitude while the running time increased by only 30%. For GCC, use of the *structs* option did not yield a significant increase in the number of points-to classes and doubled the running time of the analysis. As we can see from Figure 8.3, this is due to

the fact that a greater number of structures were collapsed in GCC than in EMACS and BURLAP. The greater number of collapses also accounts for the increased running time.

Figure 8.3 also shows the results for combining various parameterizations of the points-to analysis. For BURLAP, use of the *mallocs* option does not increase the number of points-to classes, and therefore combining the option with the other options is not worthwhile. For GCC, however, the *mallocs* option does increase the number of points-to classes, and combining the *mallocs* and *arrays* options shows a magnification in the number of points-to classes. By themselves, the *mallocs* option introduces an additional 10 classes and the *arrays* option introduces an additional 205 classes. When the options are combined, the result is an increase of 226 classes. The results are magnified even more when combined with additional parameters. A similar result holds true for EMACS. However, the results are not always magnified when options are combined. In GCC, for example, the combination of the *structs* and *arrays* options yields fewer classes than their sum. This failure to magnify can be explained by recalling that the options are not orthogonal to one another.

8.5.2 Function calls through pointers

Figure 8.4 shows the number of functions called at each call site using a function pointer—an *indirect call*. Use of the *arrays* option reduces the average number of functions called for both GCC and BURLAP, but not for EMACS, since EMACS has few arrays of function pointers. However, the *prototypes* option works well for all three programs. Strong prototype filtering works extremely well, so much so that the results do not improve further even if the *arrays* option is enabled. Since one of the co-authors is the author of BURLAP [Gobat and Atkinson, 1994], we verified the results from using strong prototype filtering by hand and found them to be near-perfect. The only cases where the strong prototype filter fails to separate functions of differing intent are when one of the two merged functions requires a pointer to some type and the other requires a pointer to `void`. These cases are not surprising since pointers to `void` are explicitly defined by the language as generic pointers.

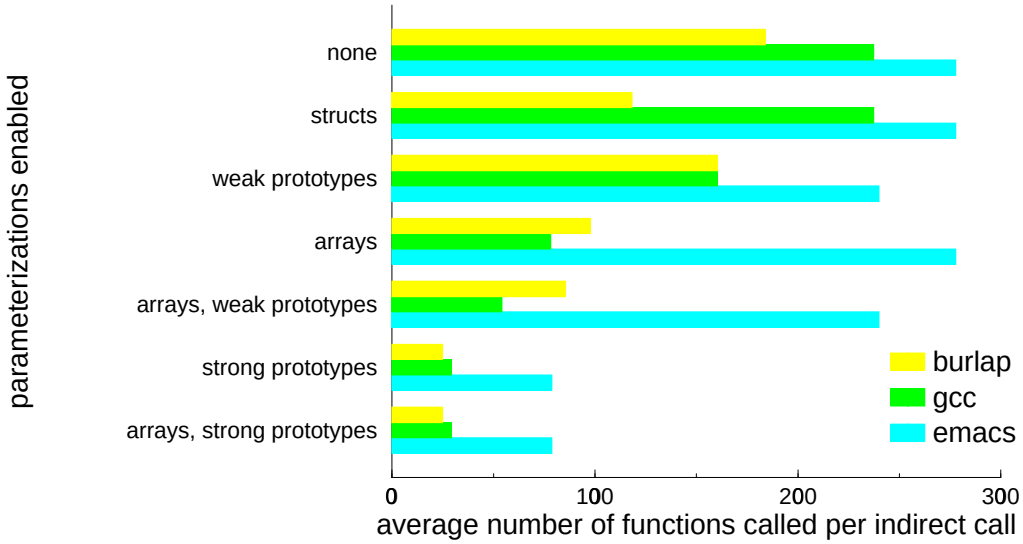


Figure 8.4: Effect of parameterization on function pointers. The graph shows the average number of functions called per call site using a function pointer (an *indirect call*) for various parameterizations of the points-to analysis. The complete measurements are given in Table A.6.

8.6 Effect of Parameterization on Program Slicing

Figure 8.5 shows the effects of user-parameterization of the points-to analysis on program slicing. The improvements due to the *arrays*, *structs*, and *mallocs* parameters are small (at most 3%), which suggests that the precision of the slicing algorithm is relatively insensitive to changes in the points-to classes. This is in contrast to our original expectations, since we saw significant improvements in the number of points-to classes using these parameters.

For the tool user, the quality of a slice is measured in terms of the number of statements included in the slice, as we have done, and not in terms of the number of data-flow facts or dependencies. The improvements of the points-to analysis can be "washed-out" by the transitive nature of the data-flow analysis being performed, as discussed in [Shapiro and Horwitz, 1997a]. For example, Figure 8.6 shows a program fragment containing calls to the private memory allocator `xmalloc()`. If the allocator

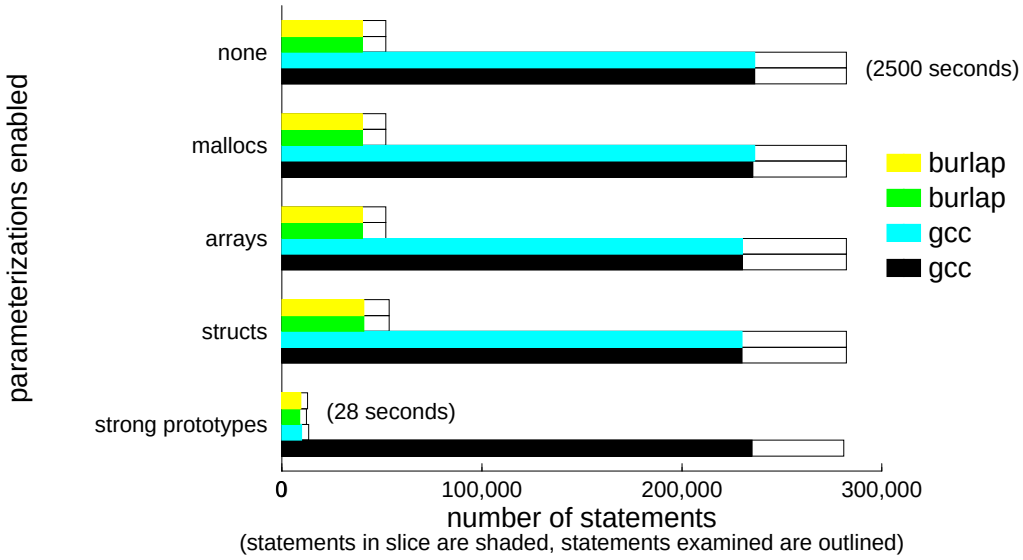


Figure 8.5: Effect of parameterization on program slicing. The statistics shown in the graph are for GCC. The complete measurements are given in Table A.7.

---

is not recognized by the points-to analysis, then a slice of either variable x or y will include the assignment statements to p and q. However, if the allocator is recognized, then a slice of variable x will not include the assignment to q, but the slice of variable y will still include both assignment statements to p and q.

The slices performed using the *prototypes* option show a dramatic improvement. The number of statements in the slice decreases several fold, as does the number of statements examined during slicing (shown as the outlined bar in Figure 8.5). This reduction is due to the more precise information for function pointers in both programs. Without use of the strong prototype filter, the computed call-graph is highly imprecise (i.e., containing a large number of false calls) resulting in a far greater number of statements being examined and subsequently included in the slice than is necessary. This implies that the filtering of function pointers based on their prototypes would be beneficial to many interprocedural data-flow analyses. We had expected to see similar results from use of the *arrays* option on GCC due to its dramatic reduction in the average number of calls made through a function pointer. However, this option fails to remove key

```
p = xmalloc (10);
q = xmalloc (10);

*p = 1;
*q = 2;

x = *p;
y = *p + *q;
```

Figure 8.6: A program fragment showing the negation of improvements in the points-to sets by program slicing. Even if `xmalloc()` is recognized as a private memory allocator, a slice of the variable `y` will still include both assignments to `p` and `q`.

---

functions from certain call sites, leading to false recursion between major subsystems (the code generator and declaration manager).

We were unable to find a parameterization of the points-to analysis that produced any substantial improvement for EMACS. We were disappointed that the use of the *prototypes* option did not yield an improvement as it had for GCC and BURLAP. After examining the source code for EMACS, we believe that the points-to sets for function pointers are fairly precise. Rather, it is the nature of EMACS that is the source of the problem. In particular, it appears that the subsystems of the EMACS interpreter are in fact recursively dependent due to the implementation of dynamically scoped error handling. For this situation, it may be useful for the tool user to provide other kinds of information, such as explicitly selecting which edges in the call-graph should be ignored and which should be traversed.

8.7 Conclusion

To test our hypotheses about demand-driven computation, discarding, flexibility, and parameterization of the points-to analysis, we computed several program slices of our example MUMPS and C programs. We were able to compute slices that were a function of the result's size, rather than of the entire program. We showed that increasing

context-depth yields little increase in precision, but increases the time and space requirements dramatically. Even if the machine has sufficient virtual memory to perform the subsequent analysis, the resulting paging overhead is detrimental.

By parameterizing the points-to analysis, the number of points-to classes increased and the number of functions called through a function pointer decreased, as expected. For some slices, parameterization substantially reduced the running time of the program slicer. However, the improvements were mainly due to the more precise information for function pointers, rather than to any increase in the number of points-to classes.

Examining our results, we believe that we have met our goal of developing practical (i.e., efficient) and task-oriented whole-program analysis tools. However, each aspect of our approach is critical to its success. For example, without customized termination, the larger slices of CHCS and GCC may require too much time to answer a tool user's question about the program.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998] and [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of these publications directed and supervised the research that forms the basis for this chapter.

Chapter 9

Conclusion

Because a large software system is difficult for its programmers and designers to understand, they could greatly benefit from automated support for program understanding. Tools supporting such analyses need to construct representations similar to those used by an optimizing compiler. However, unlike an compilation, program understanding tasks may require analyzing large portions of the program simultaneously. Consequently, to minimize time and space requirements, the tool infrastructure must adapt to the requirements of the analysis being performed, and the tool must provide flexible control of the analysis to the user.

Performing an effective whole-program analysis is difficult in the presence of pointers. Points-to analysis requires a global analysis over the program, making it difficult to integrate with demand-driven techniques, which are a necessity when analyzing large systems. The use of pointers in large programs, specifically function pointers and pointers to local variables, complicates performing an effective data-flow analysis.

To overcome these problems, we designed an event-driven software architecture for transparently demand-deriving and discarding program representations such as the AST and CFG. In our approach, we examine how a representation will be used, how long it takes to construct, and how much space it requires in order to determine how a representation should be constructed. Infrequently used representations such as the AST are discarded in order to save space and also time, by avoiding the use of the slower

portions of the virtual memory hierarchy. We also persistently retain representations that require a global analysis over the program, but which are inexpensive to store and reload, such as the program's call graph.

By allowing the user to parameterize the data-flow analysis—in our case backward program slicing—we have found that significant time and space could be saved by avoiding the computation of unnecessary data-flow information. Customizable context-depth allows the tool user to select the amount of context-sensitivity and thereby achieve a balance between good performance and precision. Our results show that increasing the context-depth results in a large increase in the time and space required by the data-flow analysis, but tends to improve precision only slightly. Allowing the user to control the termination of the analysis and increasing interactivity can avoid the computation of unnecessary data-flow information, saving time.

To effectively deal with the problems posed by pointer usage in large systems, we presented a solution for integrating points-to analysis with demand-driven analyses, along with techniques for parameterizing the analysis to achieve better points-to results. We also presented data-flow equations for computing an interprocedural backward program slice in the presence of both recursion and pointers to local variables, and described an efficient implementation of our equations.

To validate our techniques, we constructed tools for slicing MUMPS and C programs. We performed several slices of CHCS, a 1,000,000 line hospital management system written in MUMPS, and three large C programs—GCC, EMACS, and BURLAP. Using our program slicer for MUMPS, we were able to compute slices of CHCS in an economical amount of space, requiring just minutes or hours, rather than days. Using our program slicer for C, we were able to compute slices of BURLAP in a few seconds or minutes and of GCC in a few minutes or hours, without requiring use of the slower portions of the virtual memory hierarchy.

By parameterizing the points-to analysis, the number of points-to classes could be increased with little performance cost. Also, the number of functions called through a function pointer could be substantially decreased. Most notably, filtering the points-

to classes based on their prototypes greatly reduces the number of calls. For program slicing, the improved points-to information for functions resulted in more accurate and faster program slices, due to the increased precision of the computed call-graph, which suggests that our techniques would be applicable to many interprocedural data-flow analyses. For the largest slices that we performed, the time and space requirements are acceptable for today's desktop computers, indicating that practical whole-program analysis of large programs is feasible using our techniques.

9.1 Open Issues

Although we have discussed how to design practical and task-oriented whole-program analysis tools and have implemented such tools for two programming languages, a number of open questions remain.

- Can it be cost effective to share program representations among multiple tool users? How would sharing affect the way representations should be managed?
- Is it possible to design a heuristic for determining the progress of an analysis with respect to convergence? Since all of the convergence curves in Figure 8.2 are very similar, a curve-fitting heuristic might allow predicting the progress of a slice.
- Can the optimal context-depth be heuristically determined? Can the context-depth be increased or the context-graph modified during slicing to provide better precision where needed?
- How does the structure of the program being sliced affect the slicing algorithm? Do slices over well-structured modules of the system require fewer iterations and converge faster? Does the structure impact the recomputation time or the performance of the memory hierarchy?
- Is it beneficial to discard procedures of the CFG that have not yet contributed to the analysis? Is there a way to conservatively summarize information about a procedure to avoid re-examining it?

- How well do our techniques apply to data-flow analyses other than slicing?
- What do our results say about the viability of program slicing? Most slices that we performed are quite large and typically include 80% of the statements examined. Are these slices of any practical use to the programmer?

9.2 Extending Our Approach

Since prototype filtering works so well on improving the precision of the computed call-graph, one might wish to extend the technique to ordinary program variables. By using the type system of the language, similar filters can be constructed for variables. For example, if the points-to set for a variable p points to both an integer variable n and a real variable x, but p is declared to be a pointer to integer, then x can be removed from the points-to set. Since distinguishing structure members in the points-to analysis has poor time and space characteristics for some larger programs, filtering based on structure member types could be an inexpensive alternative. For languages like C, however, such filtering is almost certainly unsafe in the general case because the programmer can violate the type system. For languages with strong typing, on the other hand, type violations are not supported by the language and filtering can be safely applied. We have found that filtering based on function prototypes is worthwhile given its benefits and the infrequent violation of the type system for function pointers in C programs (i.e., casting function pointers is rare in C).

One question that remains is whether our techniques can be applied to other programming languages. Since points-to analysis requires a global analysis over the system for any programming language, our approach to integrating the analysis with demand-driven analyses applies generally. Although parameterizations of the points-to analysis such as array commutativity are specific to the C language, the general ideas can be applied to any programming language that is flexible and performance-oriented—any sufficiently powerful language will have pointer constructs whose use complicates effective points-to and data-flow analysis. For example, although private memory allocators

may not be used for performance reasons, they are often used for reasons of encapsulation and code reuse. The constructor and destructor functions in C++ [Stroustrup, 1991] are obvious examples. Distinguishing structure members and filtering points-to classes based on their computed prototypes are also necessary in C++, since its classes are little more than structures composed of many function pointers and some data. Finally, an aggressive implementation of the data-flow sets such as ours is necessary to achieve good performance when analyzing large programs, even if the generalization for pointers to locals is not required.

9.3 Contributions

9.3.1 Identification of key problems

We have identified several problems with designing and implementing efficient, task-oriented whole-program analysis tools. We addressed how program representations should be handled in order to provide space-efficient analyses, and we designed a software architecture to transparent implement our techniques. We examined how user parameterization of analyses can reduce time and space requirements and improve precision. We looked at how pointers can impact the accuracy and efficiency of a data-flow analysis and derived data-flow equations to overcome these problems. Finally, we implemented program slicing tools for two programming languages and used them to analyze several large, well-known systems.

9.3.2 Empirical evaluation

We computed several slices of our example systems. From our results we can infer some things about how programming style plays a role in program analysis. Programs that use pointers aggressively are hard to analyze because the style hinders the accuracy of the points-to analysis. Programs that use function pointers heavily are significantly impacted. Our results also reveal that program slicing may not be as useful

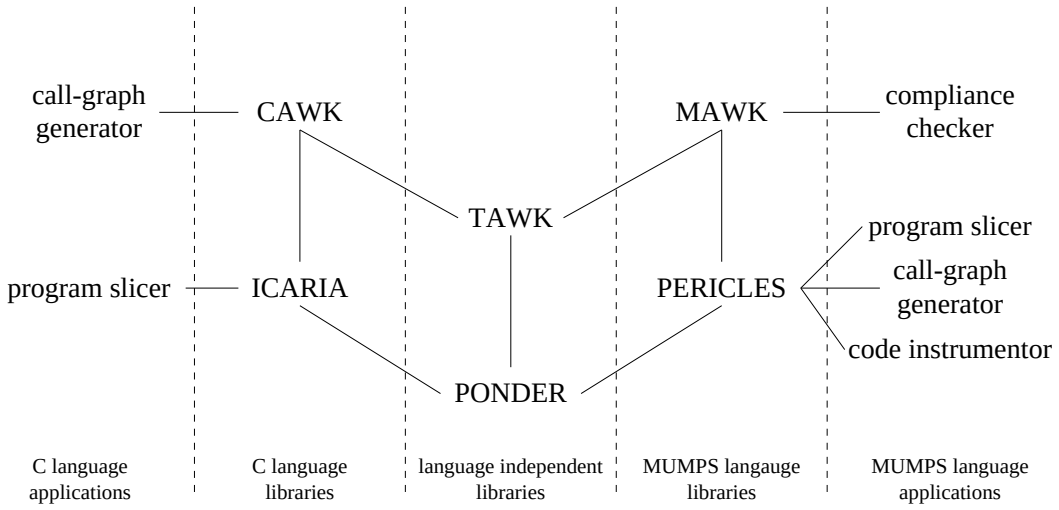


Figure 9.1: Family of tools developed for analyzing large programs.

as the program understanding community first thought. Our program slices are quite large, with typically 80% of the statements examined being included in the slice.

9.3.3 Family of program understanding tools

Because no single tool is appropriate for all tasks, we need support for developing new program understanding tools. Therefore, we have designed support for a family of tools, as shown in Figure 9.1. Our generic tool infrastructure is called PONDER. The PONDER library provides support for a generic AST along with high-performance memory allocators and flexible abstract data types useful for manipulating computed program information such as data-flow sets (Chapter 7) and index-tables (Chapter 2). An extension of the PONDER library is the TAWK library, which provides a high-level query language for matching ASTs [Griswold et al., 1996].

The PERICLES library provides the implementation of the AST for the MUMPS programming language. In addition to containing a parser and lexical analyzer for MUMPS, it provides an abstract interface for manipulating and searching the AST. The MAWK tool is high-level pattern matcher for MUMPS built from the TAWK and PERI-

CLES libraries. In addition to a program slicing tool, we have also built a compliance checker, code instrumentation tool, and call-graph generator for MUMPS. The ICARIA library provides the same services as the PERICLES library, but for the C programming language. Similarly, the CAWK tool is the equivalent of the MAWK tool, but for C programs. These tools and the supporting libraries are publicly available via the Internet at `http://www-cse.ucsd.edu/users/atkinson`.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998] and [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of these publications directed and supervised the research that forms the basis for this chapter.

Appendix A

Experimental Data

This appendix presents more experimental data than that presented in Chapter 8. The additional data is given for the sake of completeness. The data in the Chapter 8 is a representative subset of the data presented here.

For all of the tables in this appendix, the size of the slice is given as the number of three-address statements in the slice, space is given in megabytes, and the slicing criterion is a pair consisting of the statement and a set of variables. The statement is specified as a routine and line number for the MUMPS programs and as a filename and line number for the C programs. Unless otherwise stated, time is given in minutes. Also, each table contains a reference to the table or figure in the body chapter that contains a subset of its data.

The text of this chapter, in part, is a reprint of the material as it appears in [Atkinson and Griswold, 1998] and [Atkinson and Griswold, 1996]. The dissertation author was the primary researcher and author and the co-author of these publications directed and supervised the research that forms the basis for this chapter.

criteria	contexts	time	space	size of slice
(*NSUN:26,*{*NSORD*})	253	6.1	17.3	14,660
(*FHDRSTR:52,*{*FHQUIT*})	256	5.0	15.0	14,700
(*DGBED:84,*{*DGW*})	254	5.3	18.1	14,829
(*MSAKP:62,*{*MSAEFFDT*})	267	4.6	15.3	15,645
(*CHPLI:33,*{*LRPTN*})	394	9.0	17.3	20,852
(*PSNST:25,*{*IN*})	596	22.9	22.3	32,351
(*LRPRACT:10,*{*LRALL*})	611	24.1	22.6	33,156
(*CPENRFRM:21,*{*CPRBUF*})	647	29.8	23.8	33,935
(*ORSETN:42,*{*ORACTIN*})	1,255	141.9	60.5	73,707
(*ORSIGN:32,*{*ORLPKFG*})	1,321	154.9	71.8	73,907
(*ORENTRY:68,*{*LRORPTNZ*})	1,432	179.9	74.1	79,319

Table A.1: *Abbreviated in Figure 8.1 on Page 70.* Statistics for different slices of CHCS with a single context per procedure.

	criteria	time	space	size of slice
BURLAP	(*arithmetic.c:245,*{*type_error*})	0.20	2.0	8,849
	(*apply.c:646,*{*status*})	0.23	2.0	9,107
	(*matrixfunc.c:243,*{*status*})	0.22	2.1	9,332
	(*apply.c:243,*{*result*})	1.12	8.1	40,161
EMACS	(*alloc.c:1610,*{*gc_cons_threshold*})	5.38	31.4	129,292
	(*buffer.c:447,*{*buf*})	6.15	31.3	129,292
	(*frame.c:630,*{*frame*})	6.75	31.6	129,292
GCC	(*cse.c:8777,*{*in_libcall*})	0.02	0.2	824
	(*unroll.c:2085,*{*const0_rtx*})	0.46	2.6	9,702
	(*sched.c:4964,*{*reg_n_calls_crossed*})	40.04	75.4	235,030
	(*c-decl.c:2298,*{*b*})	40.99	76.1	235,037

Table A.2: *Abbreviated in Figure 8.1 on Page 70.* Statistics for different slices of the three C programs. The points-to analysis was performed using the *strong prototypes* option (Chapter 5).

	criteria	depth	contexts	time	space	size of slice
COMPLY	(*COMARG:15,*{*ERRTYP*})	1	26	0.017	0.29	488
		2	49	0.033	0.37	483
		∞	363	0.217	0.97	483
COMPLY	(*COMBLK:14,*{*ERRTYP*})	1	36	0.030	0.37	780
		2	74	0.058	0.43	774
		3	121	0.092	0.55	769
		∞	602	0.484	1.42	769
CHCS	(*PSPA:46,*{*PSDT*})	1	248	4.9	14.9	14,630
		2	488	11.1	18.5	14,448
		3	2,586	127.6	49.1	14,446
CHCS	(*DIC:38,*{*DUOUT*})	1	248	5.3	19.6	14,711
		2	488	10.3	23.1	14,532
		3	2,553	121.0	53.6	14,530

Table A.3: *Abbreviated in Table 8.1 on Page 71.* Statistics at different context-depths for the two MUMPS programs.

	criteria	depth	time	space	size of slice
BURLAP	(*arith.c:145,*{*type_error*})	1	0.20	2.0	8,849
		2	1.24	5.4	8,824
		3	3.56	17.5	8,680
BURLAP	(*apply.c:253,*{*result*})	1	1.12	8.1	40,161
		2	7.30	32.1	39,952
		3	30.94	101.3	39,937
BURLAP	(*apply.c:646,*{*status*})	1	0.23	2.0	9,107
		2	1.23	5.4	9,086
		3	3.76	17.5	8,942
BURLAP	(*matrixfunc.c:767,*{*status*})	1	0.22	2.1	9,332
		2	1.19	5.7	9,296
		3	3.90	18.3	9,152
GCC	(*cse.c:8777,*{*in_libcall*})	1	0.02	0.2	824
		2	0.04	0.6	824
		3	0.31	3.4	824
GCC	(*unroll.c:3085,*{*const0_rtx*})	1	0.46	2.6	9,702
		2	4.20	12.9	9,574
		3	29.98	71.0	9,556

Table A.4: *Abbreviated in Table 8.2 on Page 72.* Statistics at different context-depths for two C programs. The points-to analysis was performed using the *strong prototypes* option (Chapter 5).

	none			mallocs			arrays		
	time	space	classes	time	space	classes	time	space	classes
GCC	51.5	62.8	267	53.6	62.8	277	51.0	63.2	472
EMACS	26.9	38.8	159	26.4	38.8	162	26.5	38.8	169
BURLAP	15.4	22.7	207	15.3	22.7	207	15.2	22.7	279

	arrays, mallocs			structs			arrays, structs		
	time	space	classes	time	space	classes	time	space	classes
GCC	51.3	63.6	493	117.3	89.5	543	116.4	90.0	746
EMACS	26.6	38.8	174	40.2	51.5	5638	39.8	51.5	5648
BURLAP	15.2	22.7	279	21.3	29.0	1360	21.4	29.0	1430

	arrays, structs, mallocs			ideal structs			arrays, ideal structs		
	time	space	classes	time	space	classes	time	space	classes
GCC	125.0	90.1	1013	457.1	88.4	1835	192.7	89.0	1826
EMACS	40.4	51.5	5654	135.4	51.2	6710	97.9	51.1	6671
BURLAP	21.3	29.0	1430	24.8	28.4	1787	25.1	28.4	1805

Table A.5: *Abbreviated in Figure 8.3 on Page 75.* Effect of parameterization on points-to analysis. The table shows the performance of the points-to analysis for various parameterizations of the analysis. The space measurements include all necessary data structures including symbol tables. Also shown is the number of points-to classes. Time is given in seconds rather than minutes.

		basic	structs	arrays	weak	weak w/ arrays	strong	strong w/ arrays
GCC	(113)	237.3	237.3	78.1	160.1	54.1	29.2	29.2
EMACS	(70)	277.7	277.7	277.7	240.0	240.0	78.8	78.8
BURLAP	(16)	183.8	118.2	97.8	111.1	85.5	24.9	24.9

Table A.6: *Abbreviated in Figure 8.4 on Page 78.* Effect of parameterization on function pointers. The table shows the average number of functions called per call site using a function pointer (an *indirect call*) for various parameterizations of the points-to analysis. The number in parentheses indicates the number of indirect call sites in each program.

	points-to options	time	size of slice	statements examined
GCC	basic	49.44	236,366	282,192
(*c-decl.c:2298,*{*b*})	arrays	39.11	230,306	282,192
	prototypes	43.66	235,037	235,037
GCC	basic	42.32	236,354	282,192
(*unroll.c:3085,*{*const0_rtx*})	mallocs	48.20	236,351	282,192
	arrays	32.41	230,305	282,192
	prototypes	0.46	9,702	13,281
	combined	0.42	9,702	13,281
BURLAP	basic	1.29	40,135	51,863
(*arith.c:145,*{*type_error*})	arrays	1.04	40,204	51,863
	structs	1.72	40,883	53,535
	prototypes	0.20	8,849	12,195
	combined	0.23	8,661	12,319
BURLAP	basic	1.29	40,135	51,863
(*matrixfunc.c:767,*{*status*})	arrays	1.07	40,204	51,863
	structs	1.62	40,858	53,535
	prototypes	0.22	9,332	12,764
	combined	0.25	9,144	12,890

Table A.7: *Abbreviated in Figure 8.5 on Page 79.* Effect of parameterization on program slicing. The table shows the statistics for various slices of the example programs with different parameterizations (*basic*, *arrays*, *structs*, and strong prototypes) of the points-to analysis. (Some of the additional three-address statements generated for structure member accesses have been removed for comparison with the other parameterizations. However, not all additionally generated statements could be easily removed.) For GCC, the *arrays* and *prototypes* options were combined; for BURLAP, the *structs* and *prototypes* were combined.

Bibliography

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

Atkinson, D. C. and Griswold, W. G. (1996). The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, Berlin, Germany.

Atkinson, D. C. and Griswold, W. G. (1998). 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.

Barth, J. M. (1978). A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736.

Choi, J.-D., Burke, M., and Carini, P. (1993). Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC.

Choi, J.-D., Cytron, R., and Ferrante, J. (1991). Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, FL.

Choi, J.-D., Cytron, R., and Ferrante, J. (1994). On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490.

Duesterwald, E., Gupta, R., and Soffa, M. L. (1995). Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 37–48, San Francisco, CA.

Dwyer, M. B. and Clarke, L. A. (1994). Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM Symposium on the Foundations of Software Engineering*, pages 62–75, New Orleans, LA.

Emami, M., Ghiya, R., and Hendren, L. J. (1994). Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM '94 SIGPLAN Conference on Programming Language Design and Implementation*, pages 20–24, Orlando, FL.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

Gallagher, K. B. and Lyle, J. R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761.

Garlan, D. and Shaw, M. (1993). An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–39. World Scientific, Singapore.

Gobat, J. I. and Atkinson, D. C. (1994). The FElt system: User's guide and reference manual. Computer Science Technical Report CS94-376, University of California, San Diego, Department of Computer Science & Engineering.

Griswold, W. G. and Atkinson, D. C. (1995). Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116.

Griswold, W. G., Atkinson, D. C., and McCurdy, C. (1996). Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 144–153, Berlin, Germany.

Griswold, W. G. and Notkin, D. (1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269.

Griswold, W. G. and Notkin, D. (1995). Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE Transactions on Software Engineering*, 21(4):275–287.

Harrold, M. J. and Ci, N. (1998). Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, Kyoto, Japan.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60.

Horwitz, S., Reps, T., and Sagiv, M. (1995). Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 104–115, Washington, DC.

Jackson, D. (1991). ASPECT: An economical bug-detector. In *Proceedings of the 13th International Conference on Software Engineering*, pages 13–22, Austin, TX.

Jackson, D. and Rollins, E. J. (1994). A new model of program dependences for reverse engineering. In *Proceedings of the 2nd ACM Symposium on the Foundations of Software Engineering*, pages 2–10, New Orleans, LA.

Johnson, S. C. (1978). A portable compiler: Theory and practice. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 97–104, Tucson, AZ.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition.

Knoop, J. and Steffen, B. (1992). The interprocedural coincidence theorem. In *Proceedings of the 4th International Conference on Compiler Construction*, pages 125–140, Paderborn, Germany.

Koelbel, C. H., Zosel, M. E., and Steele, G. L. (1994). *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA.

Landi, W. and Ryder, B. G. (1992). A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM '92 SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA.

Landi, W. A., Ryder, B. G., and Zhang, S. (1993). Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, Albuquerque, NM.

Lehman, M. M. and Belady, L. A., editors (1985). *Program Evolution: Processes of Software Change*. Academic Press, Orlando, FL.

Lengauer, T. and Tarjan, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141.

Lewkowicz, J. M. (1989). *The Complete MUMPS: An Introduction and Reference Manual for the MUMPS Programming Language*. Prentice Hall, Englewood Cliffs, NJ.

Nye, A. (1990). *X Toolkit Intrinsics Programming Manual*. O'Reilly & Associates, Sebastopol, CA, 2nd edition.

Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 23–25, Pittsburgh, PA.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Perry, D. E. and Wolf, A. L. (1992). Foundations on the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52.

Reps, T. and Rosay, G. (1995). Precise interprocedural chopping. In *Proceedings of the 3rd ACM Symposium on the Foundations of Software Engineering*, pages 41–52, Washington, DC.

Ruf, E. (1995). Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA.

Shapiro, M. and Horwitz, S. (1997a). The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, pages 16–34, Paris, France.

Shapiro, M. and Horwitz, S. (1997b). 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.

Shivers, O. (1991). *Control-Flow Analysis of Higher-Order Languages*. Ph.D. dissertation, Carnegie Mellon University, School of Computer Science.

Stallman, R. M. (1991). *GCC Reference Manual*. Free Software Foundation, Cambridge, MA.

Stallman, R. M. (1993). *GNU EMACS Manual*. Free Software Foundation, Cambridge, MA.

Steensgaard, B. (1996a). Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 136–150, Linköping, Sweden.

Steensgaard, B. (1996b). 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.

Stroustrup, B. (1991). *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2nd edition.

Sullivan, K. J. (1994). *Mediators: Easing the Design and Evolution of Integrated Systems*. Ph.D. dissertation, University of Washington, Department of Computer Science & Engineering.

Sullivan, K. J. and Notkin, D. (1992). Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268.

Weise, D., Crew, R. F., Ernst, M., and Steensgaard, B. (1994). Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 297–310, Portland, OR.

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

Wilson, R. P. and Lam, M. S. (1995). 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.