# Improving Program Slicing with Dynamic Points-To Data

Markus Mock
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350
mock@cs.washington.edu

Darren C. Atkinson
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566
atkinson@engr.scu.edu

Craig Chambers and Susan J. Eggers
Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350
{chambers,eggers}@cs.washington.edu

## ABSTRACT

Program slicing is a potentially useful analysis for aiding program understanding. However, slices of even small programs are often too large to be generally useful. Imprecise pointer analyses have been suggested as one cause of this problem. In this paper, we use dynamic points-to data, which represents optimal or optimistic pointer information, to obtain a bound on the best case slice size improvement that can be achieved with improved pointer precision. Our experiments show that slice size can be reduced significantly for programs that make frequent use of calls through function pointers because for them the dynamic pointer data results in a considerably smaller call graph, which leads to fewer data dependences. Programs without or with only few calls through function pointers, however, show only insignificant improvement. We identified Amdahl's law as the reason for this behavior: C programs appear to have a large fraction of direct data dependences so that reducing spurious dependences via pointers is only of limited benefit. Consequently, to make slicing useful in general for such programs, improvements beyond better pointer analyses will be necessary. On the other hand, since we show that collecting dynamic function pointer information can be performed with little overhead (average slowdown of 10% for our benchmarks), dynamic pointer information may be a practical approach to making slicing of programs with frequent function pointer use more successful in reality.

## Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.3.4 [**Programming Languages**]: Processors; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*

## General Terms

Languages, Measurement, Experimentation

## Keywords

Program Slicing, Points-To Analysis, Dynamic Analysis


*SIGSOFT 2002/FSE-10*, November 18–22, 2002, Charleston, SC, USA.


## 1. INTRODUCTION

Program slicing [11, 15, 33] has been proposed as an approach to aid program understanding tasks. For example, a backward program slicer computes the set of statements that may have affected the value of a given variable, which may aid programmers during debugging. Possible other applications of slicing are software maintenance, testing, and reverse engineering [7, 10, 11, 32, 33]. With such applications in mind, a variety of program slicing tools have been developed, many of these for the widely used C programming language [5, 7, 13, 14, 31].

Although in theory program slicers make several program understanding tasks easier, their usefulness in practice has been limited, because existing program slicers frequently produce slices that are quite large. Various reasons have been proposed to explain this phenomenon. For example, some slicers use context-insensitive data-flow analyses, which analyze procedures equally regardless of their call site. As a consequence, data-flow information for multiple calls to the same function is shared, resulting in more imprecise analysis results, which may lead to a larger slice size.

Another culprit may be the pointer analysis algorithms used in the slicing tool. Flow-sensitive and context-sensitive algorithms potentially produce the most precise results, but due to their complexity ($O(n^3)$ or worse, where $n$ is the number of lines of code) they generally do not scale well, limiting their applicability to relatively small programs. However, slicing would be most beneficial for large programs that cannot be easily understood without the use of tools. Therefore, existing slicers typically use pointer analyses that trade precision for better performance to enable slicing of complex programs. These analyses are typically not fully flow- or context-sensitive [9, 28, 30, 34].

The pointer imprecision problem is particularly severe for C programs, which use pointers extensively to simulate call-by-reference semantics, to emulate object-oriented dispatch via function pointers, to avoid the expensive copying of large objects, to implement list, tree, or other complex data structures, and as references to objects allocated dynamically on the heap. Therefore, imprecise pointer analyses will result in conservative assumptions about data dependences in a program, contributing to a larger slice size.

The pointer analysis used by existing slicers is usually implemented as a *points-to* analysis. Points-to analysis determines, for each variable in the program, the set of locations (i.e., variables, procedures, and heap locations) to which a given variable may point at a particular program point. The resulting *points-to sets* are used by the subsequent data-flow analysis to resolve pointer dereferences.

Traditional *static* points-to analyses compute an approximation of the set of objects to which a pointer may point. They are conser-

vative in the sense that their results must be correct for any input and execution path of the program. In addition, for the C programming language they have to make various conservative assumptions when analyzing a program, for instance, because of C's weak type system.

An alternative way of gathering points-to data is to perform a *dynamic* points-to analysis. A dynamic points-to analysis records the targets of program pointers during actual program execution, by instrumenting the program source with calls to an appropriate data-capturing routine. Since dynamic points-to sets only capture the targets of pointers during a particular program execution, they are in general unsound (i.e., optimistic). However, with a sufficient number of program inputs and execution runs, the dynamic sets are likely close to optimal. Moreover, any sound points-to analysis and a (generally infeasible) optimal points-to solution must *at least* contain the targets present in a dynamic points-to set. Therefore, dynamic points-to sets can be used to obtain an approximate lower bound on the points-to sets. A recent study [23] showed that the typically observed dynamic points-to sets are 10–100 times smaller than the points-to sets computed by Das's highly-scalable One-Level Flow algorithm [9], which generally produces results as precise as Andersen's well-known algorithm [2]. Andersen's algorithm, in turn, has been shown [17] to be of comparable precision as some other well-known pointer analysis algorithms [17, 24]. Mock et al. [23] additionally showed that the majority of program variables point to only a single logical location during execution with the SPEC-provided test inputs.[1] Although more expensive flow-sensitive algorithms [26] can obtain much better average points-to sets than the scalable analyses used in [23], they still do not in general yield points-to sets as small as the dynamic sets.

Since the dynamic sets are a lower bound for the results of any sound static analysis, we can use them to obtain an upper bound on the potential improvement of slice sizes that might be achieved by using more precise pointer analysis algorithms in slicing. To compute this bound, we modified the *Sprite* program slicing tool [5, 6] to accept dynamic points-to data from *Tumi*, which is a modification of the Calpa instrumentation tool [21, 22]. We chose to use applications from the SPEC 2000 benchmark suite to perform our experiments, since SPEC benchmarks are of considerable size, cover a wide range of tasks (e.g., simulations, group theoretic computations, graphics, databases, word processing) and they are actually used in practice.[2] In addition, we used several benchmarks that have been used by other researchers in the evaluation of their slicing studies [15, 17, 18]. We instrumented, executed, and collected data for our benchmarks and constructed a variety of slices on the source code using the dynamic points-to data.

Our results were bimodal, and depended on the degree of function pointer use in the program. Programs that contained many (static) call sites that call procedures through function pointers had significantly smaller slice sizes. Their improvement in slice size was due to the reduced call graph that resulted from using the dynamic points-to information for call sites. On the other hand, programs which made no or only infrequent use of calls through function pointers saw little improvement in slice size, typically just a few percent. At the root of this counter-intuitive behavior, we found that a large fraction of dependences in these programs were direct, i.e., not pointer-induced, and therefore their potential improvement from better pointer information is limited by Amdahl's Law. In more detail, our paper makes the following contributions:

- We compute an upper bound on the reduction in slice size that can be obtained by improved pointer information for a wide range of realistic applications. While other researchers have compared slice sizes obtained with pointer analyses of different precision [17], our study is the first to establish an upper bound of possible slice size reduction by improved pointer information.
- We show that programs with many call sites that make calls through function pointers experience a significant reduction in slice size, when dynamic pointer information is used. The smaller slices are due to the much more compact call graphs that dynamic pointer information produces. For the programs in our test suite and the inputs used to obtain their dynamic pointer data, we were able to verify that the reduced call graph is not simply a consequence of bad code coverage, but is a much better (though somewhat optimistic) approximation of the possible call relationships present in the programs. Since the precision of analyses of object-oriented programs is often limited by an imprecise call graph,[3] these analyses, as well as program slicing for object-oriented programs, such as those written in Java, may also benefit from our technique.
- Despite dramatically smaller points-to sets, programs without many calls through function pointers see little reduction in slice size. We found that the C programs in our test suite contain a large fraction of direct data dependences. Therefore, the effectiveness of removing (even a very large number of) spurious dependences that arise from imprecise pointer information, is limited. While we can claim that this is the case only for the benchmarks in our study, we believe that it is likely to apply to many other C programs as well, based on the variety of applications that the programs in our benchmark suite represent.
- Finally, since reductions in slice size only occur on programs for which the dynamic points-to data results in a more compact call graph, program instrumentation can be restricted to collect dynamic points-to data only for function pointers. We show that this restricted form of profiling results in only minimal slowdowns ranging from 2–30%. This makes the construction of a better (though optimistic) call graph a practical technique, that can be harnessed by all software tools that would improve from better call graph information, including, but not limited to, program slicers.

The rest of this paper is organized as follows. Section 2 and Section 3 describe our program slicing tool, *Sprite*, and the instrumentation framework with which we obtained the dynamic points-to data. We present our experimental setup in Section 4 and discuss the results in Section 5. Section 6 contains related work, and we conclude in Section 7.

## 2. PROGRAM SLICING

We used a modification of version 3.0 of the *Sprite* program slicing tool [3, 5], a research prototype developed for slicing C programs, to compute program slices that are based on both the static

[1] A logical location is either a program variable or a heap allocation site. There may be multiple instantiations of a single logical variable in the case of local variables and multiple distinct objects allocated at the same memory allocation (heap) site.

[2] Applications are submitted to the SPEC consortium and selected based on their relevance and representativity of actual computing practice.

[3] In general, the possible targets of object-oriented method calls can only be approximated very imprecisely by static analysis, resulting in an imprecise call graph.

and dynamic points-to data. Currently, Sprite computes only backward program slices.

Sprite first constructs the control-flow graph (CFG) of the program. The CFG consists of basic blocks of three-address statements, each of which represents a single computation such as a simple addition or pointer dereference. Steensgaard's static points-to analysis [30] is then performed over the CFG to compute equivalence classes of memory locations that are used as points-to sets during slice computation. Although Sprite can perform a slight modification of the points-to analysis that distinguishes the fields of structures, this modification was not used since previous results found it yielded little improvement in program slices and negatively impacted performance during slicing [5]. To compute a program slice, Sprite computes a maximum-fixed-point solution to the data-flow equations given in [5] using an iterative, convergence algorithm. After slice computation is complete, Sprite reports the number of source code lines included in the slice and can highlight the included lines within a user interface.

Most default options to Sprite were used during the experiments. The sole exception was that the names of functions that performed custom memory allocation (e.g., `xmalloc` in `find`) were specified to increase the precision of the points-to analysis. We generally computed context-insensitive slices in our experiments, because previous work had shown that context-sensitive slices are not much smaller and require significantly more time to compute [4, 8]. However, to rule out context-sensitivity as a factor influencing our results, in Section 5.3 we also computed some context-sensitive slices.

# 3. DYNAMIC POINTS-TO DATA

We used a slightly modified version of the instrumentation tool *Tumi* [23] to generate the points-to data used in this study. The dynamic points-to sets are obtained in three steps. First, a static points-to analysis is run on the application source code. For each pointer and dereference point, it computes a conservative approximation of the set of logical locations (variables, procedures, or memory allocation sites) a pointer may point to. Then the application is instrumented, inserting code that associates the run-time addresses of pointers with the run-time addresses of potential pointer targets (identified by the static points-to analysis of the first step). Finally, the instrumented application is compiled, and executed on some representative input. Upon termination, the instrumentation code will save the set of logical locations that were referenced at each instrumented pointer use, thereby producing a dynamic points-to set for each pointer use. In this process the address matching step is essential: since distinct run-time addresses may refer to the same logical location, simply recording the pointer addresses is not sufficient to construct the set of logical locations pointed-to at run time. More details can be found in [23] and in [20].

The dynamic points-to data is flow-sensitive, since it is collected per pointer dereference point in the program. For the experiments reported in Section 5, we also produced flow-insensitive dynamic points-to sets as follows. For each pointer variable `p`, the dynamic points-to sets of all program points that dereferenced `p` were combined (using set union), producing the set of objects `p` pointed to during execution, regardless of where the pointer was dereferenced.

## 3.1 Generating Points-To Data

To obtain the dynamic points-to sets for the applications in this study, we used the SPEC-provided test inputs, which are meant to exercise the programs' functionality. We chose to use the test inputs, since they allow us to gather the points-to sets faster. We also found that running the applications on the larger reference data sets produced virtually unchanged points-to data, possibly because the reference inputs execute the same parts of the application only more often. For the non-SPEC programs, we either used the examples and test suites provided with the applications or performed representative tasks, such as searching through all files in a directory hierarchy or scanning a large volume of text. Instrumentation slowed down the applications by 1 to 2 orders of magnitude, causing the test inputs to finish within minutes or hours, well within the time scale of computing actual slices. Moreover, the points-to data can be reused across different slices of the same application, thereby amortizing the cost of generating the dynamic points-to data.

## 3.2 Instrumentation for Function Pointers

Since our results in Section 5.4 show that slices of programs that include many calls through function pointers can be considerably improved by using dynamic points-to data exclusively for the call sites, we would like to be able to gather the function addresses with minimal slowdown. Fortunately, unlike run-time addresses of variables, procedure addresses do not change at run-time. Therefore the expensive mapping from run-time addresses to compile-time names (performed while the program is executing) is not necessary. Instead, we have to capture only the addresses of the functions that are invoked at call sites that use function pointers.

To instrument these call sites, we created a lightweight instrumentation version of Tumi, which collects only the run-time addresses of function pointers. During execution, this lightweight instrumentation stores those addresses in a per-call-site hash-table. When the program finishes, the contents of the table are saved to disk, and later translated to procedure names (using, for instance, the Unix tool `nm`) to obtain the points-to sets for the executed call sites. This lightweight instrumentation resulted in much smaller slowdowns, ranging from 0.6% for `mesa` to 30% for `gap`, with a geometric mean of 10.3%. These degradations are comparable to the slowdowns imposed by standard profiling tools such as `gprof` or `pixie`. With this technique, therefore, function pointer data can be collected efficiently with minimal run-time overhead.

# 4. EXPERIMENTS

This section describes our workload, the choice of slicing criteria, and how we generated the actual slicing results.

## 4.1 Workload

For our experiments, we chose to instrument and slice programs in the SPEC 2000 benchmark suite, along with programs used by other researchers in their slicing experiments. We chose to use programs from the SPEC benchmark suite because they are of considerable size, perform a variety of different computations (from graphics, compression, spell checking, mathematical computations to simulation), and are actually used in practice (they are submitted to the SPEC consortium). Unfortunately, there is no consensus on what a typical C application looks like, so we had to use what we believe to cover a good range of actual computing practice. We chose to use in addition some applications that have been used in previous slicing work for comparison purposes. Unfortunately, many of the programs used in previous slicing experiments, were either no longer available, too small, or not interesting with respect to points-to information because even the flow-insensitive static points-to analyses we compare against in our paper were able to produce very precise points-to data because of the simplicity of the programs.

Table 1 shows the programs used with their sizes and the number of executable lines (i.e., lines that actually perform some computation). The slices computed by Sprite include only executable

| | Source Lines | Executable Lines | Reachable Functions | Executed Functions | Slicing Criteria | Origin | Description |
|---|---|---|---|---|---|---|---|
| art | 1,270 | 545 | 22 | 18 | 837 | SPEC 2000 | image recognition, neural networks |
| equake | 1,513 | 670 | 24 | 19 | 1,111 | SPEC 2000 | seismic wave propagation simulator |
| mcf | 1,909 | 635 | 24 | 21 | 880 | SPEC 2000 | combinatorial optimization |
| bzip2 | 4,639 | 1,246 | 63 | 21 | 1,579 | SPEC 2000 | compression |
| gzip | 7,757 | 1,864 | 62 | 26 | 1,546 | SPEC 2000 | compression |
| ispell | 8,020 | 2,742 | 107 | 33 | 1,617 | GNU (v3.1.20) | spell checking |
| parser | 10,924 | 4,414 | 297 | 230 | 6,223 | SPEC 2000 | word processing |
| diff | 11,755 | 3,285 | 110 | 27 | 2,110 | GNU (v2.7) | file comparison |
| ammp | 13,263 | 5,614 | 161 | 46 | 5,146 | SPEC 2000 | molecular dynamics |
| vpr | 16,973 | 5,954 | 255 | 163 | 7,993 | SPEC 2000 | circuit placement and routing |
| less | 18,305 | 4,371 | 328 | 117 | 1,879 | GNU (v358) | text file viewing |
| twolf | 19,748 | 11,304 | 167 | 104 | 13,816 | SPEC 2000 | placement and global routing |
| vortex | 52,633 | 23,245 | 643 | 518 | 31,324 | SPEC 2000 | object-oriented database |
| *grep* | 13,084 | 3,674 | 108 | 39 | 3,520 | GNU (v2.4.2) | pattern matching |
| *find* | 13,122 | 3,004 | 96 | 37 | 740 | GNU (v4.1) | filesystem searching |
| *mesa* | 49,701 | 21,069 | 770 | 130 | 7,270 | SPEC 2000 | graphics |
| *burlap* | 49,845 | 16,608 | 189 | 123 | 5,293 | FELT (v3.05) | finite element solver |
| *gap* | 59,482 | 19,998 | 826 | 356 | 15,245 | SPEC 2000 | group theory interpreter |

**Table 1: Sizes and descriptions of the programs used in the experiments. An executable line is any line of source code that performs a computation during runtime. In particular, declarations, blank lines, and comments are excluded. Italicized programs use function pointers heavily and are therefore listed together.**

lines of code; therefore, the slice sizes reported in Section 5 refer to the number of executable lines included in the slice. The last five programs use function pointers heavily (discussed in detail in Section 5.4), and are therefore listed together. We gathered the dynamic-points to sets as described in Section 3.1.

## 4.2 Slicing Criteria

Ideally, slicing criteria, (i.e., pairs of the form `(statement, variable)`), would be chosen that might be used by a software engineer during debugging (since Sprite computes a backward program slice). However, since we are unfamiliar with the benchmarks, we instead elected to exhaustively generate slicing criteria for each program, i.e., we generated all possible `(variable, statement)` pairs for the program, with the only restriction that `variable` is referenced in `statement`. This ensures results that are not biased because of a particular choice of slicing criteria.[4] We then restricted the initial slicing criteria to come from only those functions that were actually executed during some points-to profiling run to ensure the availability of dynamic pointer information.

Table 1 shows the number of possibly reachable functions in each program and the number of functions executed during the instrumentation runs, along with the number of criteria used in the experiments. The percentage of executable functions of the total (statically) reachable functions varies widely for the programs, demonstrating that the code coverage of the inputs is sometimes quite poor (e.g., for `mesa` for which only about 17% of the reachable functions were executed). This means that some of the test cases provided with the applications need improvement. The set of reachable functions was constructed using a call graph extractor that uses Steensgaard's [30] points-to analysis to account for the effects of function pointers.

## 4.3 Experimental Procedure

The following steps were performed for each program in our test suite:

1. Instrument each program to record the pointer dereferences at run time using Tumi.
2. Execute the program on its provided test input, recording the raw dynamic points-to information. For each executed dereference point, its position in the program and the set of target logical locations is written out to disk.
3. For each pointer, compute the flow-insensitive data by merging the referenced objects across all points that dereferenced the pointer, as described in Section 3.
4. Using the static data, the flow-sensitive dynamic data, and the flow-insensitive data, perform program slices using Sprite on the generated criteria, recording information such as the final size of the slice in lines and the sizes of the incoming data-flow sets for each basic block.
5. Compute average sizes and reductions in slice size.

# 5. RESULTS

## 5.1 Data-Flow Analysis

We measured two quantities during slice computation: dereference size and data-flow set size. The dereference size is the size of a pointer's points-to set at the time of its dereference. When a statement of the form `*p = x` or `x = *p` is visited during slicing, the size of the dereference set of `p` is recorded by Sprite. (Since Sprite uses an iterative algorithm, a single program point may be visited many times; however, the points-to set and hence the dereference size do not change.) We used this quantity to measure the number of data-flow facts (e.g., variables) present at a program point. We also measured the size of the incoming data-flow set of a basic block. Upon visiting each block, the size of the set is recorded by Sprite. This quantity gives an indication of the amount of data-flow information being propagated during analysis.

We then computed averages over all slices of these two quantities for each program in our test suite. Figure 1 shows the improvements in each of these measurements from using the flow-insensitive dynamic points-to data (computed as explained in Sec-

[4]Other slicing work has typically glossed over this point; for instance, [17] and [27] do not describe in detail what criteria were used in their experiments.

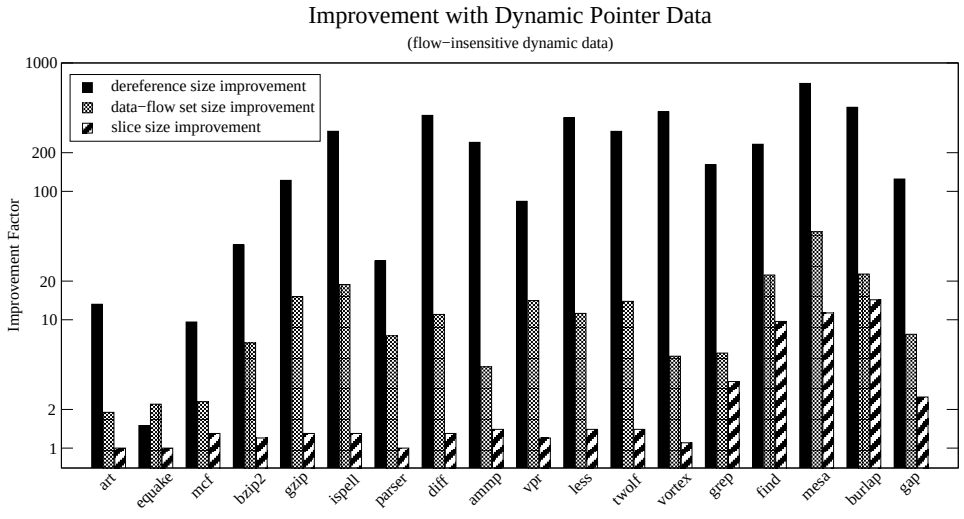


**Figure 1: Improvement in average dereference size, set size, and slice size for slices computed using dynamic data.**

tion 3). The improvements from using flow-sensitive points-to data are similar (within 1–5% for the dereference size and 1–2% for the data-flow set size) and are not shown.

The reduction in dereference size is typically an order of magnitude or more, ranging from a factor of only 1.4 for `equake` to a factor of close to 700 for `mesa`. However, the reduction in data-flow set size is less, sometimes substantially so. This drop implies that although we have introduced fewer data-flow facts into the analysis at statements involving pointers, that decrease does not yield an equal decrease in the amount of data-flow information being propagated during analysis.

## 5.2 Slice Size

Table 2 presents the average slice size for each program, including the results obtained using flow-sensitive dynamic points-to data. The data shows that using the flow-sensitive points-to data instead of the flow-insensitive data has virtually no effect on slice size. Since for the majority of the dereference points the flow-sensitive dynamic points-to sets were identical to the flow-insensitive sets, this was unsurprising. Furthermore, this indicates that (at least for the applications in our benchmark suite) there may be limited benefits of using flow-sensitive pointer analysis in general, which appears to be consistent with the way pointers are typically used in C programs (passing pointers to large structures, for instance).

Figure 1 also presents the improvement in average slice size when using dynamic pointer data. The data shows that our applications fall into two categories. For the first category, the improved pointer data results in only insignificant improvement. All of the applications in the second category, however, which comprises all the applications that use function pointers heavily (`grep`, `find`, `mesa`, `burlap`, and `gap`), showed a considerable reduction in slice size. Note that the figure shows improvement in *average* slice size, i.e., $(\frac{1}{n}\Sigma static_i)/(\frac{1}{n}\Sigma dynamic_i)$. Since a slice with dynamic points-to data is guaranteed to be never larger than the corresponding slice with static points-to data, comparing the static average slice size with the average slice size with dynamic data gives an indication of the overall improvement in slice size with dynamic points-to data. As an alternative measure, we also looked at the pairwise data and computed the average percent reduction in slice size, i.e., $\frac{1}{n}\Sigma[(static_i - dynamic_i)/static_i]$. The measurements showed the same general trends and are therefore omitted from the paper. The data is available in a technical report [20].

## 5.3 Slices of Programs With Little Function Pointer Use

In order to explain the lack of improvement for programs with little function pointer usage, we wanted to isolate and eliminate factors, such as context-sensitivity and control dependences, that might influence slice size. As discussed in Section 2, previous work indicated that increasing context-sensitivity yields only a small improvement in slice size. However, context-sensitivity and points-to set size are not orthogonal, and any such improvement could be magnified by using more precise points-to data. Therefore, we performed slices with increased context-sensitivity as practical.[5] Comparisons between the improvements in slice size for our original (context-insensitive) slices and the slices with increased sensitivity are shown in Figure 2. As the figure demonstrates, increasing context-sensitivity has little effect on the improvement in slice size. In fact, the slices with static data and slices with dynamic data generally improved about the same when context-sensitivity was enabled.

Another possible factor that might explain the general lack of improvement in slice size for the programs in the first category are control dependences. For example, more precise points-to data might eliminate a data dependence between two statements *A* and *B*, but *A* may still be included in the slice because *B* is control-dependent upon *A*. To assess the impact of control dependence on our slices, we modified Sprite to ignore intraprocedural control dependences when computing slices. The effects of ignoring control dependences on slice size are shown in Figure 2. The graph demonstrates that the slice size reduction resulting from slicing with dynamic pointer data was only slightly higher when ignoring control

[5]We were able to perform fully context-sensitive slices only for the smaller applications. For the medium-sized applications we were able to compute 2-CFA results (i.e., context-sensitive for call depths of up to 2 [29]). For `vortex` even computing 1-CFA results ran out of memory.

| | Static | Dynamic (flow-insensitive) | Dynamic (flow-sensitive) |
|---|---|---|---|
| art | 59.6 | 57.1 | 57.1 |
| equake | 168.4 | 164.8 | 164.8 |
| mcf | 56.8 | 45.3 | 45.3 |
| bzip2 | 73.0 | 58.5 | 58.5 |
| gzip | 54.0 | 42.0 | 42.0 |
| ispell | 242.2 | 185.5 | 185.5 |
| parser | 195.9 | 186.9 | 186.7 |
| diff | 228.3 | 171.2 | 171.2 |
| ammp | 339.0 | 247.0 | 247.0 |
| vpr | 117.0 | 100.5 | 100.3 |
| less | 536.9 | 394.3 | 393.8 |
| twolf | 335.6 | 237.9 | 237.9 |
| vortex | 3,449.3 | 3,240.3 | 3,240.3 |
| *grep* | 527.8 | 183.2 | 183.2 |
| *find* | 460.8 | 47.4 | 45.7 |
| *mesa* | 3,267.3 | 288.3 | 288.3 |
| *burlap* | 5,291.6 | 369.6 | 369.4 |
| *gap* | 7,758.1 | 3,133.5 | 3,006.7 |

**Table 2: Average number of lines in a slice for slices computed using the static and dynamic points-to sets.**

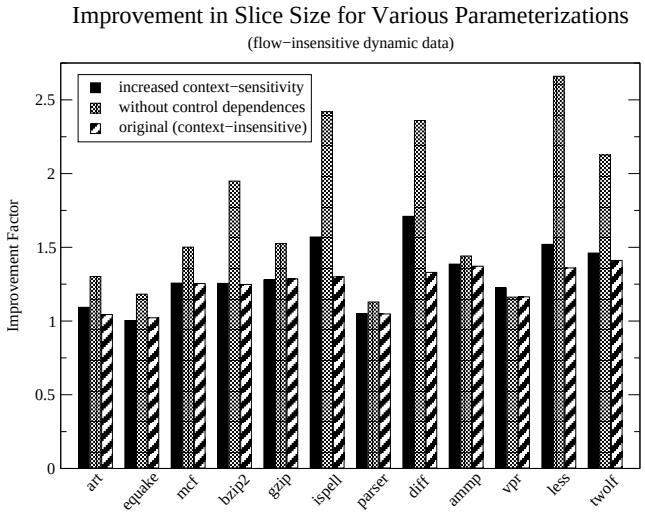


**Figure 2: Improvement in average slice size for slices with increased context-sensitivity, for slices computed without control dependences, and for context-insensitive slices. The bars labeled `original` refer to the context-insensitive slices with dynamic points-to data including both control dependences and data dependences.**

dependences. Five of the applications (`bzip2`, `ispell`, `diff`, `less`, and `twolf`) show significant improvement; however, the factor of improvement is still much less than what might be expected when using points-to data that is 10 to 100 times better. Therefore, although control-dependences have some effect on any improvement that can be gained from better pointer information, they cannot totally account for any lack of substantial improvement in all of the applications in the first category.

Since neither context-sensitivity nor the effects of control dependence explain the limited improvements gained by using dynamic points-to data for the applications with little function pointer use, we decided to look at the data dependences that are present regardless of the quality of pointer information. Therefore, we modified Sprite to construct a data-dependence graph that could be used to compute a program slice. (Ordinarily, Sprite uses an iterative algorithm to compute a maximum fixed point solution as discussed in Section 2.) In the data-dependence graph, an edge links a use of a program variable to definitions that reach that use. Once the graph is constructed, a program slice can be computed (ignoring control dependences) by simply performing graph reachability [15]. Smaller points-to sets should lead to fewer dependences between statements and therefore fewer edges in the graph. Figure 3 shows the number of edges in the data-dependence graph computed using both the static and dynamic points-to data.[6] The number of dependences decreases significantly for most programs. Only `art` and `equake` show little reduction, which is not unexpected given that they use pointers only to create and access arrays, not to create and manipulate complex structures.

Not all edges in the data-dependence graph are due to the effects of pointers. Direct dependences are those dependences that are *not* induced by pointer dereferences. These edges are always present regardless of the precision of the points-to sets. Figure 3 also shows the number of direct dependences between statements. Figure 4 shows this same data but with edges classified as direct edges, dynamic pointer edges, and edges present only when using the static data.

[6] `vortex` is not shown in the figure because the computation of the data dependence graph ran out of memory. It is also omitted from Figure 4 for the same reason.

As the figures indicate, for the smaller programs such as `gzip`, the majority of the dependences are direct. For the medium-sized programs such as `vpr`, approximately 33% of the dependences are direct. Consequently, any benefits from the more precise points-to data are immediately diminished by Amdahl's Law [1]. Amdahl's Law states that regardless of how much a part of a program that accounts for a fraction $f$ of the execution time is improved, the overall speedup will never exceed a factor of $1/(1-f)$. Similarly, regardless of how much we improve pointer-induced data dependences, this improvement will never exceed the limit imposed by the fraction of direct dependences present in the program. For instance, for `vpr` the data dependence edge improvement could be at most a factor of 3.2 even though its average dynamic points-to set is 100 times smaller than the static points sets. Since the data dependence edges largely determine the final slice, slice size improvement through better pointer information is ultimately limited by the fraction of direct dependences present in the program. Our results show that for the C programs in our benchmark suite, direct dependences make up a large fraction of all data dependences. Consequently, even our optimal (or optimistic) pointer information improved slice size only insignificantly. Given the wide range of applications that our benchmark programs represent, we strongly believe that a large fraction of direct dependences is likely to be found in many C programs in general, similarly limiting the effectiveness of more precise pointer information for reductions in slice size in those cases.

## 5.4 Slices of Programs with Heavy Function Pointer Use

For the programs in our benchmark suite that use function pointers heavily, we found that slices with dynamic data decreased by a factor of 2.5 for `gap` to 14.3 for `burlap`. To ascertain that this improvement is in fact due to improved function pointer data, we applied dynamic pointer information selectively in the following

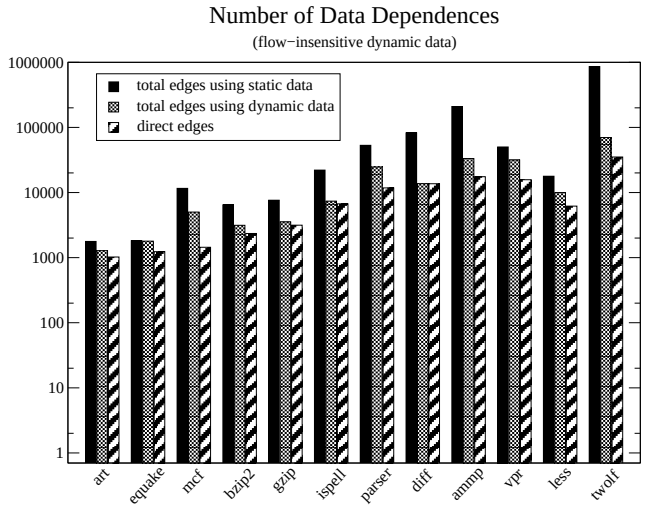


**Figure 3: Number of data dependences computed using static and dynamic points-to data. Also shown is the number of direct (non-pointer-induced) dependences.**

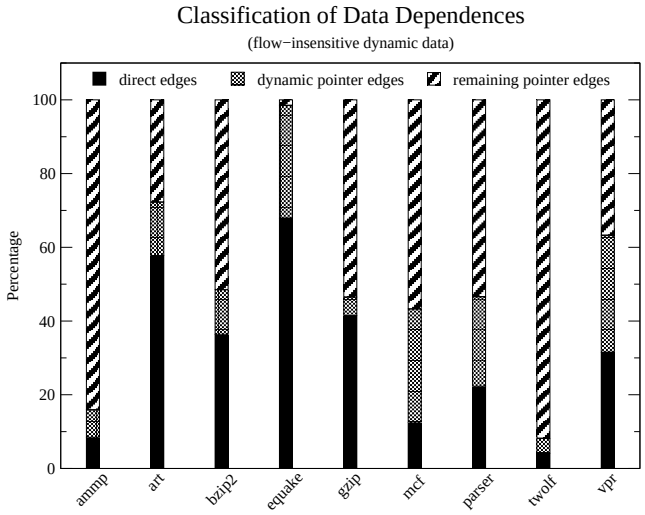


**Figure 4: Classification of data dependences showing the number of direct edges, dynamic pointer edges, and remaining pointer edges (additional pointer edges present using the static points-to data but not present using the dynamic points-to data).**

way. In addition to the slices with dynamic data for all pointers, we computed slices where we used dynamic data only for the function pointers and static data for the pointer variables, and slices where we used static data for the function pointers but dynamic data for variables. As shown in Figure 5, using the dynamic data only for the function pointers accounts for 48% to 91% of the improvement achieved by using dynamic data for all pointers. On the other hand, using the dynamic data only for the variables achieves only little reduction in slice size, demonstrating that most of the benefit derives from the improved call graph.

To estimate how much of the additional improvement is due to the optimistic nature of our dynamic pointer data, we used the following simple static techniques to obtain a better bound for the possible improvement due to better function pointer data. First, we enabled the filtering of the points-to sets for function pointers based on the their prototypes, i.e., all procedures in a points-to set whose prototype do not match the required prototype at the call site are eliminated from the static points-to sets.[7] For cases in which prototype filtering failed to reduce the points-to set size, we examined the source code of each application by hand to determine the approximate points-to sets for function pointers. To specify this information to Sprite, we specified a lexical pattern for filtering the static points-to data. For example, for `find`, we specified a pattern indicating that any call through a function pointer named "`parse_function`" resolved to any function whose name began with "`parse_`". Table 3 shows the resulting slice sizes using prototype or lexical filtering and the slice sizes for applying dynamic data universally, and selectively to only function pointers.

With the exception of `gap`, for which we were unable to come up with good lexical filters because of the complexity of and our unfamiliarity with the program, we found that the slice sizes obtained with dynamic function pointer data were generally closer to the sizes resulting from filtered static points-to data than to the much larger slices obtained by using the static points-to data alone. For instance, for `burlap` the average filtered slice size was 1,128, whereas the static slice size was 5,292, and the slice size with dynamic function pointer data was 461, i.e., the slices with static data were on average 4.7 times larger than the filtered slices, but the slices with dynamic function pointer data were only a factor of 2.4 too optimistic. For `grep` and `mesa` the dynamic slices were particularly close to the results obtained with filtering, indicating that the dynamic slices are not too optimistic. For `find` the slices with dynamic function pointer information turned out to be very optimistic. The reason is the poor code coverage of the test cases provided with the `find` tool (Table 1 shows that only 37 of the 96 reachable functions were executed), so that of the potentially called functions at call sites only a few were exercised when gathering the dynamic points-to data. Since good test cases that exercise all parts of a program should be part of any sound software development practice, we expect that using dynamic points-to data for function pointers will work well in practice, as long as good test cases for the pointer data generation are available.

[7]This is sound for programs obeying the ANSI C rules. In general, however, it may be unsound.

# 6. RELATED WORK

In work closest to ours, Liang and Harrold [18] describe a slicing algorithm that reuses slicing information computed in previous slices to speed up the slicing computation of subsequent slices. For their slicing algorithm they find that decreased points-to set size results in faster slicing times. In a separate study [17] they describe their pointer analysis algorithm (named *FICS*) in detail and apply it in their slicing tool. They show that the precision of FICS is comparable to Andersen's algorithm, and they perform slices with several pointer algorithms. However, while they also compare slice size for different pointer algorithms, their comparison of slice sizes compares only different static algorithms whose precision is fairly close, whereas our slices with dynamic data represent a true bound on any possible improvement from better pointer information.

The effects of context-sensitivity both on pointer-analysis and program slicing are examined by Horwitz et al. [16]. In practice, fully context-sensitive analyses are simply not feasible and there-

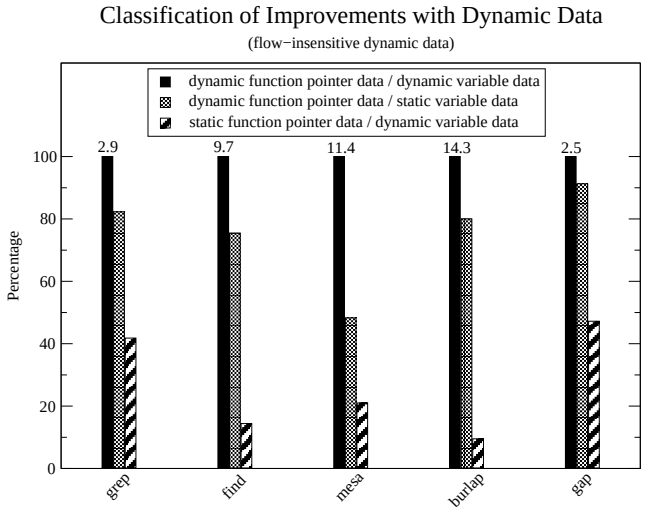


**Figure 5: Classification of improvements in slice size for slices using dynamic data. For each application, the factor of improvement using dynamic data for both function pointers and variables is shown.**

fore recent work has focused on providing analyses that are approximately context-sensitive [4] or on attempting to recover precision without adversely affecting performance [19]. Both Horwitz et al. and Liang and Harrold [19], however, find that context-sensitivity generally provides little improvement in slice size; in [19] most programs improve only by a few percent, and the best improvement is 23%. Our dynamic data is context-insensitive; however, since the majority of program variables point to only a single object, there is little motivation for increasing context-sensitivity of the points-to analysis, and in our context-sensitive slice experiments we found in fact only small improvement of slice size due to context-sensitivity.

In recent years, much work has been done to improve the precision and efficiency of pointer analyses. Horwitz [28] and Das [9] provide algorithms to improve the precision of points-to analyses, the latter achieving the precision of Andersen's [2] algorithm while running almost as fast as Steensgaard's algorithm [30]. Rountev and Chandra [25] use a technique called *variable substitution* to replace a set of program variables that are guaranteed to have the same points-to sets with a single variable, which reduces the size of the problem and greatly improves efficiency. Ryder et al. [26] present a flow-sensitive, interprocedural modification side-effects analysis, and show that it in many cases flow-sensitive pointer analysis is feasible with good results. However, none of the mentioned techniques reduce the size of the resulting points-to sets as much as using dynamic points-to data or even the combination of dynamic data supplemented with static data.

There has been much work done looking at poor points-to information as the source of imprecision in subsequent analyses. A recent experiment showed that distinguishing the individual fields of a structure, as opposed to treating the structure as a single object, yielded significantly better results for some compiler optimizations [12]. One study [27] examined the effects of improved points-to information on compiler analyses such as live variable analysis as well as program slicing. The final results improved only marginally in the case of program slicing. Bent et al. [8] demonstrated the importance of accurate library modeling on program

| | Static | Filtered | Dynamic (func. ptrs.) | Dynamic (all) |
|---|---|---|---|---|
| grep | 527 | 223 | 222 | 183 |
| find | 460 | 253 | 63 | 47 |
| mesa | 3,267 | 639 | 596 | 288 |
| burlap | 5,292 | 1,128 | 461 | 370 |
| gap | 7,758 | 7,747 | 3,433 | 3,133 |

**Table 3: Average slice sizes for slices with static data, prototype or lexically filtered static data for function pointers, dynamic data applied to function pointers only, and dynamic data applied to both function pointers and variables.**

slicing. However, in all cases, the sizes of the slices were generally too large to be useful in aiding program understanding.

# 7. CONCLUSIONS

Program slicing is a potentially useful analysis for aiding program understanding. Precise slices are most useful to the software engineer, since smaller slices mean less code to examine. In this paper, we looked at improving the precision of program slicing by using dynamic points-to data. Since dynamic points-to sets are a lower bound on the results of any sound static pointer analysis, we can use them to provide a lower bound on slice size. We found that more precise points-to information did indeed result in less propagated data-flow information during slice computation, as expected. However, the effects on slice size were bimodal.

First, for programs with many calls through function pointers, we found a significant improvement in slice size. Even though some fraction of this improvement was due to the optimistic nature of the pointer information, we were able to verify for our benchmarks that a large fraction of the improvement is in fact realizable in practice by combining the dynamic data with some simple inspection techniques. Moreover, for applications of slicing where soundness is not key, e.g., debugging, unmodified dynamic pointer data can be used directly. Since we show that dynamic function pointer data can be collected with little overhead, this may be a practical technique to improve slices for programs that use function pointers frequently. Moreover, this technique may be useful for slicing of object-oriented programs as well, since object-oriented dispatch shares some of the characteristics of function pointer calls in C. Since small points-to sets for function pointers resulted in considerable reductions slice size, exploring static pointer analyses that produce particularly precise results for function pointers is another interesting area of future research.

Second, for programs with few calls through function pointers, there was only little improvement in slice size. We found that this counter-intuitive result is due to Amdahl's law. C programs appear to contain many direct data dependences so that removing any spurious data dependences via function pointers is of generally little effect.

For this latter class of programs, our results suggest that advances other than improved pointer analysis are necessary in order to improve the quality of slices. Since algorithmic improvements like context-sensitive slicing or optimal points-to information showed little general improvement of slice sizes, requiring the user to make certain assertions about program properties, e.g., that the source code observes ANSI type rules, may turn out to be the best (maybe only) way towards practically useful program slicing in this case.

## 8. REFERENCES


[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS 1967 Joint Computer Conference*, pages 483–485, Atlantic City, NJ, Apr. 1967.

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

[3] D. C. Atkinson. *The Design and Implementation of Practical and Task-Oriented Whole-Program Analysis Tools*. Ph.D. dissertation, University of California, San Diego, Department of Computer Science & Engineering, Apr. 1999.

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

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

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

[7] L. Beltracchi, J. R. Lyle, and D. R. Wallace. Using a program slicing CASE tool for evaluating high integrity software systems. In *Proceedings of the 1996 American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies*, pages 1033–1039, University Park, PA, May 1996.

[8] L. Bent, D. C. Atkinson, and W. G. Griswold. A comparative study of two whole programs slicers for C. Computer Science Technical Report CS2001-0668, University of California, San Diego, Department of Computer Science & Engineering, Apr. 2001.

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

[10] M. A. Francel and S. Rugaber. The value of slicing while debugging. In *Proceedings of the 7th International Workshop on Program Comprehension*, pages 151–169, Pittsburgh, PA, May 2001.

[11] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, Aug. 1991.

[12] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–58, Snowbird, UT, June 2001.

[13] GrammaTech, Inc. Codesurfer user guide and reference manual.

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

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, Jan. 1990.

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

[17] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceeedings of the 7th European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, pages 199–215, Toulouse, France, Sept. 1999.

[18] D. Liang and M. J. Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursion. In *Proceedings of the 1999 International Conference on Software Maintenance*, pages 421–432, Oxford, England, Aug. 1999.

[19] D. Liang and M. J. Harrold. Light-weight context recovery for efficient and accurate program analyses. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 366–375, Limerick, Ireland, June 2000.

[20] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Gathering dynamic points-data and its incorporation in a program slicing tool for C programs. School of Engineering Technical Report COEN-2002-03-16, Santa Clara University, Department of Computer Engineering, Mar. 2002.

[21] M. Mock, M. Berryman, C. Chambers, and S. J. Eggers. Calpa: A tool for automating dynamic compilation. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, pages 100–109, Haifa, Israel, Nov. 1999.

[22] M. Mock, C. Chambers, and S. J. Eggers. Calpa: A tool for automating selective dynamic compilation. In *Proceedings of the 33rd Annual Symposium on Microarchitecture*, pages 291–302, Monterey, CA, Dec. 2000.

[23] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 66–72, Snowbird, UT, June 2001.

[24] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. Softw. Eng.*, 20(5):385–403, May 1994.

[25] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analyis. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 47–56, Vancouver, BC, June 2000.

[26] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. *ACM Trans. Prog. Lang. Syst.*, 23(2):105–186, Mar. 2001.

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

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

[29] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. dissertation, Carnegie Mellon University, School of Computer Science, May 1991.

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

[31] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3):121–189, Sept. 1995.

[32] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Meth.*, 10(1):5–55, Jan. 2001.

[33] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4):352–357, July 1984.

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