Experimental Results from Dynamic Slicing of C Programs

G. A. VENKATESH
Bellcore

Program slicing is a program analysis technique that has been studied in the context of several different applications in the construction, optimization, maintenance, testing, and debugging of programs. Algorithms are available for constructing slices for a particular execution of a program (dynamic slices), as well as to approximate a subset of the behavior over all possible executions of a program (static slices). However, these algorithms have been studied only in the context of small abstract languages. Program slicing is bound to remain an academic exercise unless one can not only demonstrate the feasibility of building a slicer for nontrivial programs written in a real programming language, but also verify that a type of slice is sufficiently thin, on the average, for the application for which it is chosen. In this article we present results from using *SLICE*, a dynamic program slicer for C programs, designed and implemented to experiment with several different kinds of program slices and to study them both qualitatively and quantitively. Several application programs, ranging in size (i.e., number of lines of code) over two orders of magnitude, were sliced exhaustively to obtain average worst-case metrics for the size of program slices.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments—*interactive*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimizations*

General Terms: Experimentation, Languages, Performance

Additional Key Words and Phrases: Program analysis, program slice

1. BACKGROUND

The notion of program slicing originated in the seminal paper by Weiser [1984], who defined a program slice as a set of program statements that directly or indirectly contribute to the values assumed by a set of variables at some program point over all possible executions of the program. For example, a program slice with respect to the variable **i** in statement (5) of the program fragment in Figure 1(a) is shown in Figure 1(b).

Weiser's algorithm was further refined and extended by Ottenstein and Ottenstein [1984] and by Horwitz et al. [1988], with the latter group using

Author's address: Bellcore, 445 South Street, Morristown, NJ 07962; email: venky@bellcore.com.

```
(1)   scanf("%d",& n);
(2)   i = 2; c = 0;
(3)   while  (i <= (n/2))  {
(4)            if ((n%i) = = 0)  {
(5)                printf("%d", i);
(6)                c + +;
(7)            }
(8)            i + +;
(9)   }
(10)  printf("%d", c);
```

(a)

```
scanf("%d", & n);
i = 2;
  while  (i <= (n / 2))  {
           if ((n%i)  = = 0)  {
             printf (%d", i);
           }
           i + +;
}
```

(b)

Fig. 1. Example program slice (b) with respect to variable **i** in statement (5) of the program fragment (a).

program slices for integrating noninterfering versions of programs. The program slices used in these studies are called *static* slices, since they are computed statically for all possible executions of the program.

Static slicing was extended to the dynamic case by Korel and Laski [1988], who defined a dynamic slice as a subprogram that computes the values of variables of interest in a specific execution of the program. A variant definition of a dynamic slice was provided by Agrawal and Horgan [1990] where a dynamic slice is not necessarily an executable subprogram but rather a collection of statements that affects the values of variables of interest in a specific execution of the program.

The earlier notions, as well as several new notions of program slices, were defined and categorized in a formal framework by Venkatesh [1991], with the assumptions that there is no single notion of a program slice that is optimal for all applications and that one must explore the characteristics of various types of slices to select the type of slice that is appropriate for a given application.

2. MOTIVATIONS

Although studies on slicing in the literature (including the ones mentioned in Section 1) have explained the notion of slicing and its applications and have provided algorithms, almost all of them have done so in the context of small abstract languages. We are aware of four slicer implementations that tackle programs in a realistic language:

(1) an experimental prototype to demonstrate the use of dynamic program slicing for debugging C programs [Agrawal 1991];

(2) an experimental prototype that uses a static slicer for C or Pascal programs in a software maintenance tool, *Ghinsu* [Livadas and Roy 1992];

(3) a prototype static C slicer based on value dependence graphs [Ernst 1994]; and

(4) a C program analysis tool incorporating a static C slicer [Jiang et al. 1991].

The first implementation can only accommodate trivial C programs, due to its severe limitations in handling procedure calls, varied data structures, etc. It is also severely limited by speed considerations, due to its dependence on a debugger to gather execution data. The second implementation handles only a subset of the ANSI C language (no pointer variables). The third implementation uses an intermediate representation that is an optimized version of the program, and hence, the implications of just the slicing algorithm on source code reduction are not clear. We have been unable to get any information on the extent of the language considered and on the details of the implementation of the slicer for the last work mentioned above. None of these works provide any information on the efficacy of program slicing in terms of the slice size and/or execution times for realistic programs.

The algorithms for various types of slices differ in the direction and the extent to which they perform closures on data and control dependences [Venkatesh 1991]. Extending these algorithms to slice programs in real languages introduces complexities in determining the data and control dependences for static slicing, and complexities in logging execution behavior for dynamic slicing. Although research in resolving aliases due to procedure calls, pointer variables, and array references [Cooper 1985; Cooper and Kennedy 1989; Horwitz et al. 1989; 1990; Landi et al. 1993; Weihl 1980] contributes to static slicing, static resolution of aliases that is sufficiently precise to result in reasonably thin program slices for a general programming language is yet to be demonstrated. On the other hand, aliasing can be easily resolved by observing execution behavior for dynamic slicing. However, the feasibility of building a dynamic slicer that is able to do so with reasonable time and space considerations for realistic programs had never been demonstrated.

Program slicing is bound to remain an academic exercise unless one can not only demonstrate the feasibility of building a slicer for nontrivial programs written in a real programming language, but also verify that a type of slice is sufficiently thin, on the average, for the application for which it is chosen. The *SLICE* prototype was built to study the feasibility of using dynamic slicing for practical applications.

3. OUTLINE OF THE ARTICLE

The next section provides an overview of the *SLICE* prototype and its use. Section 5 describes the experimental procedure for obtaining quantitative metrics for the size of program slices. Section 6 contains a summary of the results, which are analyzed in more detail in Section 7. Section 8 lists the limitations and work in progress, as well as possibilities for future work. Section 9 summarizes the article and states the conclusions of the study.

4. THE SLICE PROTOTYPE

Since the major goal of this article is to present the results of the experimentation with dynamic slicing, this section provides just a brief overview of the

design and functionality of *SLICE* that is sufficient to place the results in context.

4.1 Design of SLICE

Dynamic slicing consists of two activities: (1) the execution of a program to be sliced with a given input in order to obtain a trace about the execution and (2) the subsequent construction of slices for any variable in the program. To resolve all aliasing, it is necessary to log every use and definition of a memory location and to relate it back to the variables in the source code that are bound to this location.

4.1.1 *Execution Traces.* The solution used by Agrawal [1991] to gather execution information through a debugging tool was not considered for execution of large programs due to its serious speed limitations. There were two choices for the type of instrumentation: source level or object-code level. Source-level instrumentation was chosen over object-code-level instrumentation based on the following assumptions:

—ease of portability to different platforms,
—relative simplicity of instrumentation, and
—ease of mapping slice nodes to the original source.

With hindsight, the object-code level would have been a better choice because the first two assumptions turned out to be incorrect. We believe that object-code-level instrumentation would have resulted in significantly faster execution of the instrumented code. Moreover, object-code instrumentation would have been much cleaner and more complete in handling system and library calls. We expect any production-quality implementation of dynamic slicing to use object-code-level instrumentation.

The instrumented source code maintains the syntactic structure of the original program only up to the statement level. Each expression and some of the statements are transformed into operationally equivalent expressions and statements that log any use or definition of a memory location as they mimic the execution of the original program. This transformation is transparent to the user, since the slices are mapped back to the original source.

All system calls are handled in one of two ways. System calls that have simple input-output dependences on its parameters are listed in a table with a bitmap representation for the in/out characteristics of its parameters. This table is compiled along with the tool to generate traces of the appropriate dependences before and after the execution of such calls. All other system calls have individualized envelopes over them that call the corresponding system routine and log the dependence information.

Although the current implementation does not optimize the amount of tracing required, there are several obvious and some clever techniques to do so. These include the following:

—Logging just once for each binding of a memory location to a variable along with the log of an entry to each basic block. This method removes from the

trace (with no loss of information) entries that record every use or definition of variables whose bindings do not change in the block.

—Using the optimizations developed by Ball and Larus [1994] to place the instrumentation in a minimal number of places from which the entire trace can later be reconstructed.

These optimizations are expected to reduce the size of the trace by a factor of about 5, compared to exhaustive logging. However, note that there are significant trade-offs involved between the size of the trace, the speed of execution, and the speed for slice construction. In the design of *SLICE*, speed was given preference over space as long as the space requirements did not exceed practical limitations.

The trace is first compiled into a dynamic dependence graph that links each instance of a use of a variable to the corresponding occurrences of its definition. Each occurrence of a definition is linked to the corresponding occurrences of the variable uses (if any) in its defining expression, as well as to the corresponding occurrences of the variable uses (if any) in expressions on which the definition site is control dependent. This compilation makes the slicing phase fast enough to enable interactive use, albeit at the expense of memory.

4.1.2 *Types of Slices.* *SLICE* supports both forward and backward slicing. A backward slice consists of statements (or expressions) that affect the value of a variable of interest. The example at the beginning of this article is that of a backward slice. A forward slice, on the other hand, contains statements in the program whose computation is *affected* by the value of a variable of interest. The formal definitions can be found in Venkatesh [1991].

For each direction of slicing, *SLICE* provides four kinds of slices differing in the extent to which the closures on the dependences are performed:

(1) *Data dependence.* For backward slicing, this type of slice is a trivial slice that just contains the previous definition site of a variable of interest. For forward slicing, it is the set of sites where the current definition for a variable of interest will be used. We call the former a *Def site* slice and the latter a *Ref site* slice. These have obvious applications in debugging.

(2) *Data closure.* This slice is obtained by performing a closure over just the data dependences, starting from a variable of interest. For forward slicing, we call them *Ref closure* slices, and for backward slicing, we call them *Def closure* slices. In addition to debugging, this type of slice has applications in program-understanding and maintenance tools.

(3) *Data and control closure.* This slice is obtained by performing a closure over both data and control dependences, starting from a variable of interest. For forward slicing, we call them *Ref-Control closure* slices, and for backward slicing, we call them *Def-Control closure* slices. In addition to the applications mentioned above, this type of slice has applications in testing tools. The *Def-Control closure* slice corresponds to the notion of a dynamic slice in Agrawal and Horgan [1990].

(4) *Executable.* This slice is obtained by performing a closure over both data and control dependences, including additional statements that are required to make the slice an executable subprogram that has the same behavior on execution as the original program for all of the statements included in it. This type of slice in backward slicing corresponds to the notion of a dynamic slice in Korel and Laski [1988]. In addition to the applications mentioned above, this type of slice can be used for program specialization.

4.1.3 *Using SLICE.* There are two versions of *SLICE*: (1) an X windows-based version that can be used in an interactive mode to query for and to display various slices for a selected variable and (2) a batch-mode version that is used to slice programs exhaustively and to collect the results. An example display of the interactive version is shown in Figure 2. An executable backward slice for the variable **j** in line 21 has been highlighted.

5. THE EXPERIMENT

As mentioned earlier in the article, a major goal of this work was to obtain empirical data on the typical size of various dynamic slices in realistic programs. The feasibility of using program slicing in specific applications could then be determined based on the empirical data. To obtain such data, several application C programs were selected and executed with a suite of test inputs for each program. For each such execution, the batch version of *SLICE* was used to slice exhaustively with respect to every execution occurrence of every variable in the program. Metrics about each individual slice were output for analysis.

The selected application programs had the following characteristics:

—The larger programs were all locally developed, and the authors were easily accessible to ensure that reasonable test suites could be constructed and that the behavior of the instrumented execution could be verified for correctness, if not obvious from the output. The smaller programs are available in most UNIX[1] distributions and were selected based on the availability of a suite of test cases developed by a software-testing research group at Bellcore. The test cases were developed to exercise the program as much as possible and to cover every basic block and branch.[2]

—The larger programs were all by different authors to ensure that the slice size data were not skewed by programming styles, as well as to obtain a representative sample of various programming styles.

—They all varied in size and purpose. The smallest was a 49-line program, and the biggest was about 7500 lines. They varied in purpose so as to obtain a representative sample of different language structures and uses.

—The smaller programs were also selected to be useful for possible comparison with static slicing algorithms and implementations. The selected pro-

[1] UNIX is a registered trademark of Novell, Inc.

[2] The entire set of test suites for the UNIX programs used here can be obtained from the author.

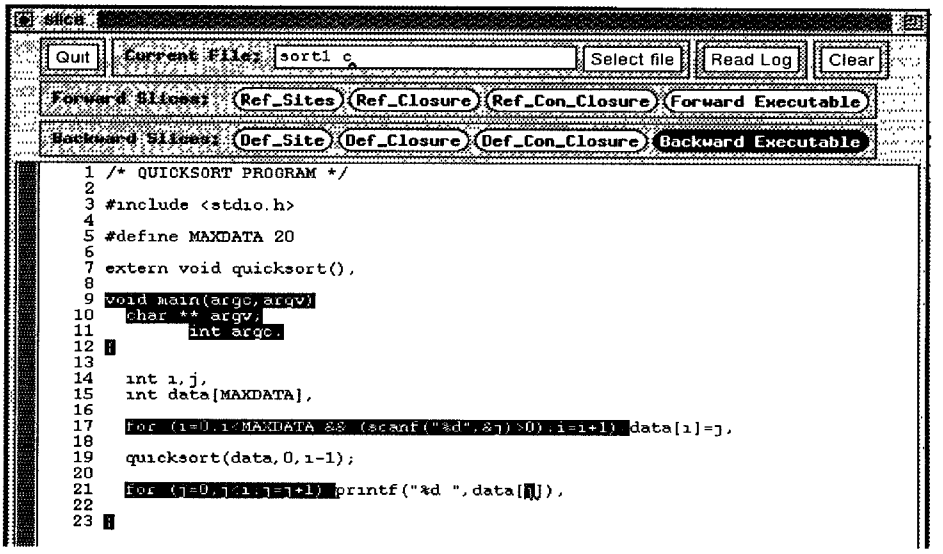

Fig. 2. *SLICE* user interface.

grams purposefully included language constructs such as arrays, pointer arithmetic, and procedures, to discourage comparison to static slicers that only worked on trivial programs.

There were at least three distinct test inputs for each possible functional mode (selected through switches in the command line or switches in the data) in which each application program could be executed. Each test input was designed to exercise as much of the code as possible. Since the purpose was to explore the upper bounds on slice sizes, trivial executions that would have resulted in small slices were not considered. For the same reason, none of the larger universally available programs could be included in the experiment. Test cases that fully exercise a program are difficult to construct without some automatic tools and/or some insight from people who are very familiar with the program. The test suites, if available, are usually designed for testing small components of the program at a time.

To keep the trace sizes reasonable, it was also necessary to select test cases that exercised a large number of different parts in the program while avoiding unusually long repeated executions of the same statements that contributed no additional information to the construction of slices. For example, if the coverage of the dependences in a sort program is independent of the number of elements sorted (accounting for all possible branch points), sorting on a smaller set of elements would be preferred. Such a selection of a test case should not, in any way, affect the generality of the results on slice sizes.

In this article, we present the data obtained from slicing nine C programs. All programs were run on DEC 5000/200s with 96MB of memory and NFS.

These programs are as follows:

(1) *SUM* is a program that computes a checksum for a file. The source code contains 49 lines. The program performs limited arithmetic and consists of a single main procedure.
(2) *UNIQ* is a program that removes or reports adjacent duplicate lines in a file. The source code contains 143 lines. The program performs mainly character manipulations and uses some pointer arithmetic.
(3) *COMM* is a program that displays lines in common (or lines not in common) between two sorted files. The source code contains 167 lines. The program performs mostly string comparisons and character operations and uses pointer arithmetic.
(4) *CAL* is a program that displays the calendar for a specified month or year. The source code contains 201 lines. The program performs some number manipulations and uses limited pointer arithmetic.
(5) *JOIN* is a relational database operator for text files. The source code contains 215 lines. The program mainly performs character operations and uses arrays and pointer arithmetic.
(6) *FLEX93* is a medical insurance costs estimation program that helps Bellcore employees to optimize the selection among the available medical options. The source code contains about 800 lines in a single file. It is a small program with limited functionality that does extensive number manipulations.
(7) *SPIFF* is a smart file comparison utility similar in function to *diff*, but uses some tolerances and structure information to compare files. The source code consists of about 4700 lines distributed in 16 files. The program uses arrays extensively with consequent proliferation of aliasing.
(8) *DQ* is the Bellcore personnel and services database query client program. The source code consists of about 2700 lines distributed in 3 files. The program uses extensive terminal I/O and network system calls.
(9) *ATAC* is a testing and coverage measurements tool that is being used in practice at Bellcore. The source code consists of approximately 7500 lines distributed in 24 files. The program has multiple functionalities that are selected by command line options. It performs extensive character manipulations and some arithmetic.

The first five programs are available on most UNIX platforms. The executions of all of the nine programs over all of the test cases and the exhaustive slicing for each run required over two months of total CPU time and about seven months of elapsed computing time on a Decstation 5000/200, primarily used as a cycle-server.

6. RESULTS OF THE EXPERIMENT

The summarized data for the execution phase of the test programs are shown in Table I. For comparison across programs, the number of executable nodes

Table I. Average Execution Data

Program	Executable Nodes (Total number)	Executed Nodes (Average Percent of Executable)	Average Execution Time		Log Size (Average MB)
			ms	overhead	
SUM	56	59.8	90.0	3.7	0.420
UNIQ	168	56.1	59.8	4.0	0.230
COMM	189	56.0	42.1	2.1	0.112
CAL	214	57.8	18.7	3.4	0.041
JOIN	368	51.9	48.9	2.0	0.148
FLEX93	997	59.9	153.6	9.8	0.574
SPIFF	3385	43.1	871.0	16.5	4.909
DQ	3722	35.9	136.7	2.0	0.573
ATAC	9080	21.9	2081.9	24.8	14.507

(i.e., program points where a memory location is used or defined) is more useful than the number of lines due to differences in documentation styles, formatting styles, equivalent languages constructs, etc. For example, the construct "**ptr" counts as three execution nodes for the three memory location accesses that it entails. This number for each program is shown in the second column. The third column provides the number of nodes executed per run averaged over all test runs as a percentage of the executable nodes. As expected, smaller programs are more likely to execute a larger fraction of the program per execution than larger programs that have multiple functionalities.

The Average Execution Time column provides the average time (sum of user and system times) for execution of the instrumented program, as well as the instrumentation overhead relative to the execution of the original program on the same inputs. The overhead ranges from a factor of 2 to a factor of about 25. DQ is a network communication-intensive program where the communication times and interactive usage times dominate the execution, and consequently, the overhead is low. ATAC, on the other hand, uses large data structures, each of whose uses and definitions must be logged.

The numbers for absolute execution time and size of the log are only meaningful in the context of this experiment for comparison between programs. First, the test inputs were chosen so as to execute as much of the program as possible while minimizing the repeated execution of statements that would not contribute additional information to slicing. Second, the instrumentation was designed to log, in a single execution, data from which all types of slices could be constructed. Without the need for executable slices (forward or backward), the size of the log will be linear in the size of the program (number of use/def nodes) [Agrawal and Horgan 1990].

The metrics from exhaustive slicing, averaged over all executions for each program, are shown in Figures 3 and 4 for quick comparison. The data are averaged over slices with respect to every execution instance of every variable in each execution of the program and are only included to provide a very rough estimate of the order of the metrics. The average slice size is not weighted by the number of different occurrences of any given variable, since

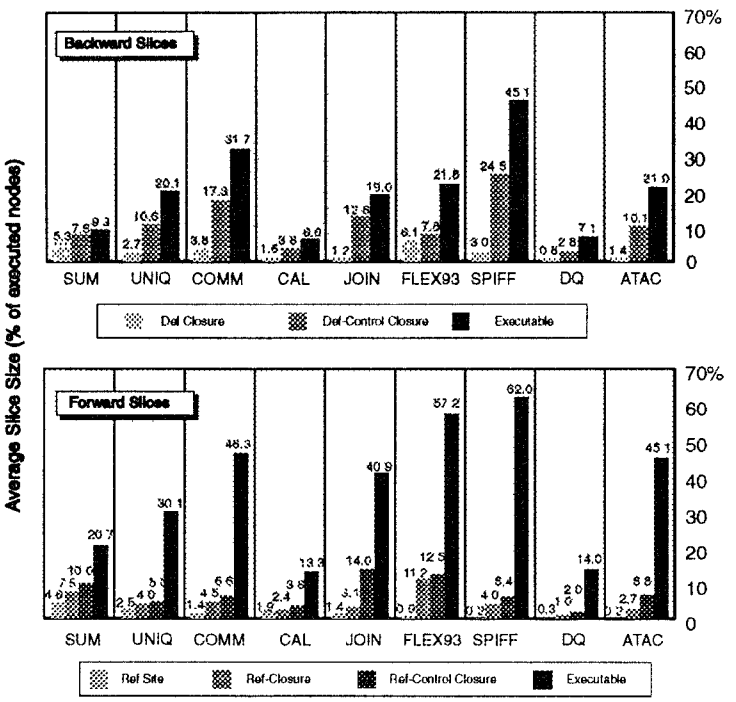

Fig. 3. Average slice size (percent of executable nodes).

the weighted average would be too dependent on the design of test inputs. The trivial backward *Def site* slice has been left out, since it always contains exactly one node and can be found in constant time from the dynamic dependence graph for any variable.

A more detailed analysis of the slice size for each program is provided in the next section. There are a few things that should be observed from Figures 3 and 4:

—As expected, the larger the number of closures performed in the slice, the larger the average size of the slice, and the higher the slicing time.

—The average slice size and average slice time are not correlated with the size of the program.

—Forward slicing is more expensive to compute and results in larger slices than do the corresponding backward slices.

—The average backward executable slice size is less than 50% of the *executed nodes* and, when combined with the average number of executed nodes in Table I, is well within 20% of the entire program. The corresponding numbers are even less for other types of backward slices.

—The average forward slice is less than 65% of the number of *executed nodes* and less than 25% of the entire program.

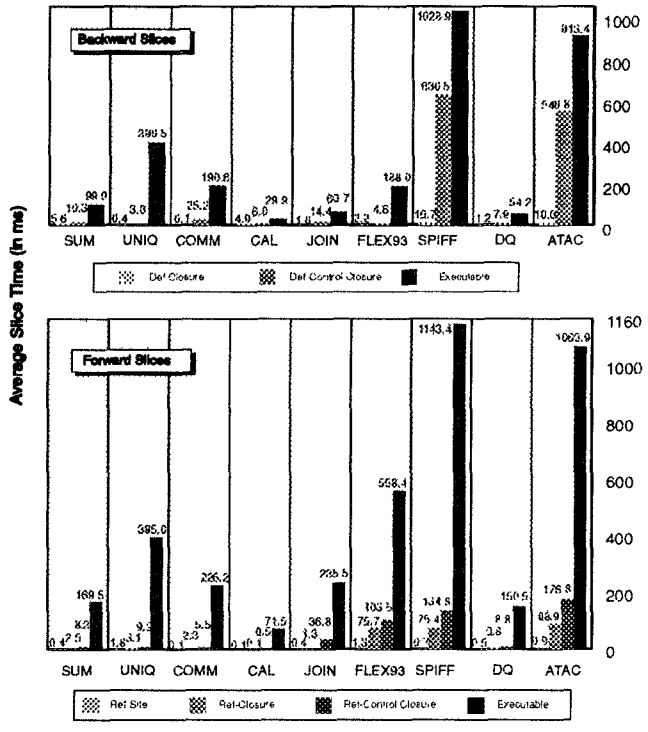

Fig. 4. Average slice time (in ms).

—Making a slice executable after considering all of the dependency closures increases the size of the slice by a factor of 2–3 for backward slicing and a factor of 5–10 for forward slicing.

7. ANALYSIS OF RESULTS

While the data pictured in Figure 3 provide a broad picture of the relative slice sizes, we need to examine the data in more detail to evaluate their applicability in practice. There are two questions, in particular, that are very relevant:

(1) What is the size distribution of the slices?
(2) How do the slice sizes relate to the kind of variable being used as the slicing criterion?

The answer to the first question provides a better indication of what to expect in an application where variables are randomly selected for slicing. The average data provided earlier may be skewed by either extremes. The second question attempts to correlate the size of slices with the intended use of the variables used as the slicing criterion.

The size distribution of slices are pictured in Figures 5 and 6 for backward and forward slicing, respectively. The distribution was computed for each test run and then averaged over all executions.

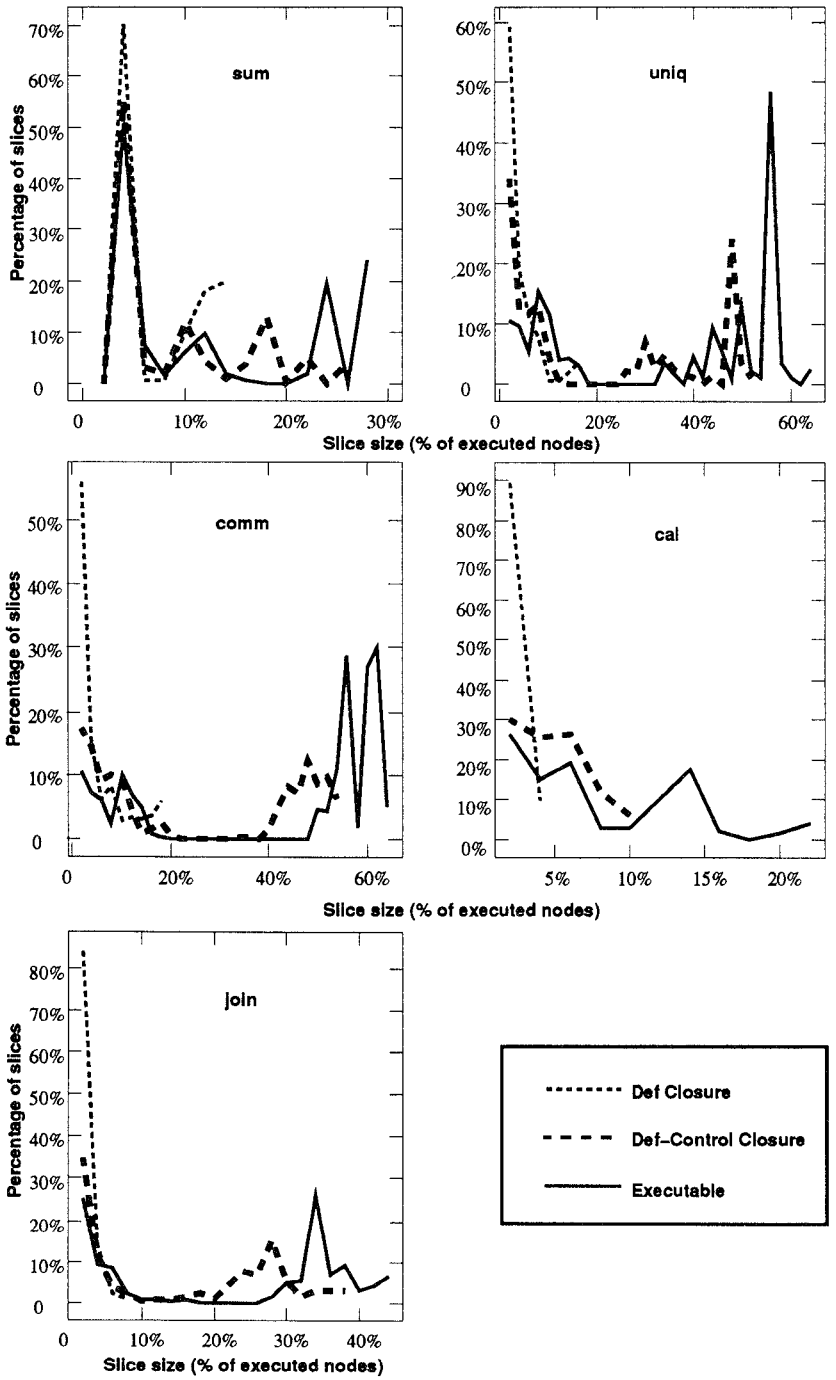

Fig. 5. Backward slice distribution.

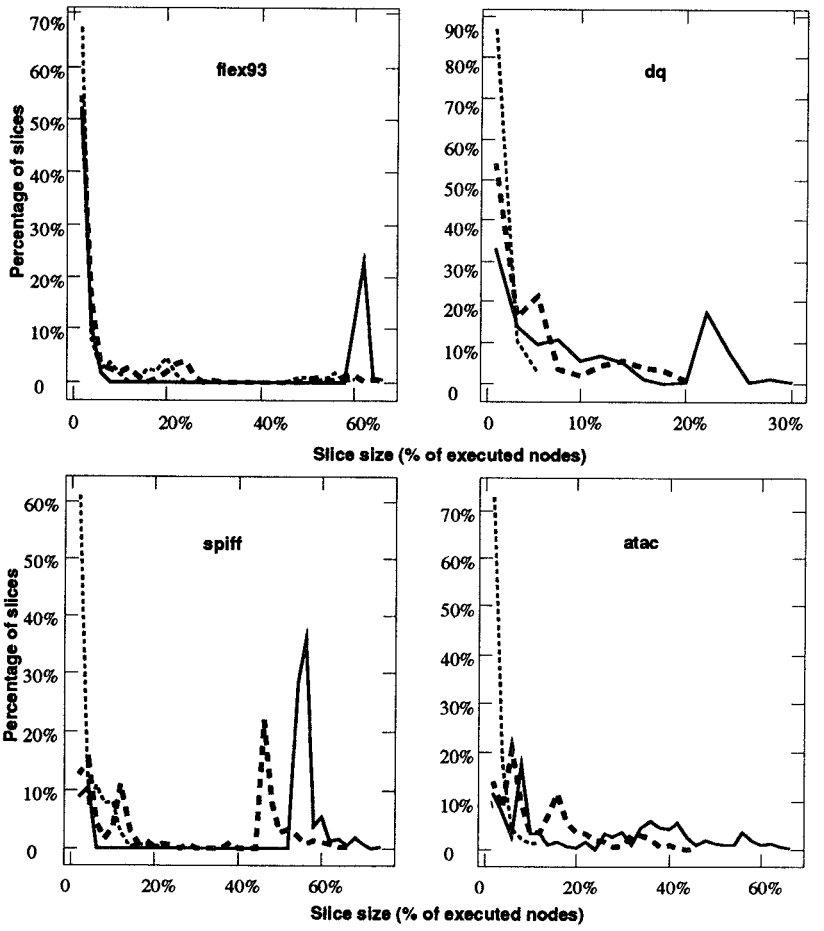

Fig. 5. *Continued*

An important observation is the clustering of slices at certain slice sizes, especially for executable slices and for closures of both data and control dependences. With a few exceptions, slices show a bimodal distribution clustering at the two ends of the range of slice sizes. A study to correlate the executable slice sizes and the functionalities of the source code captured by the corresponding slices yielded the following observations:

(1) The peak at the lower end of the slice size distribution corresponds to "trivial" slices. These are slices that include none of the basic functionalities of the program, but rather constitute executable programs involving variables whose values are independent of the basic computations of the programs. A typical example is the executable slice shown in Figure 2 for variable j. The quicksort routine that forms the crux of the program is irrelevant to the computation of the value of the variable j.

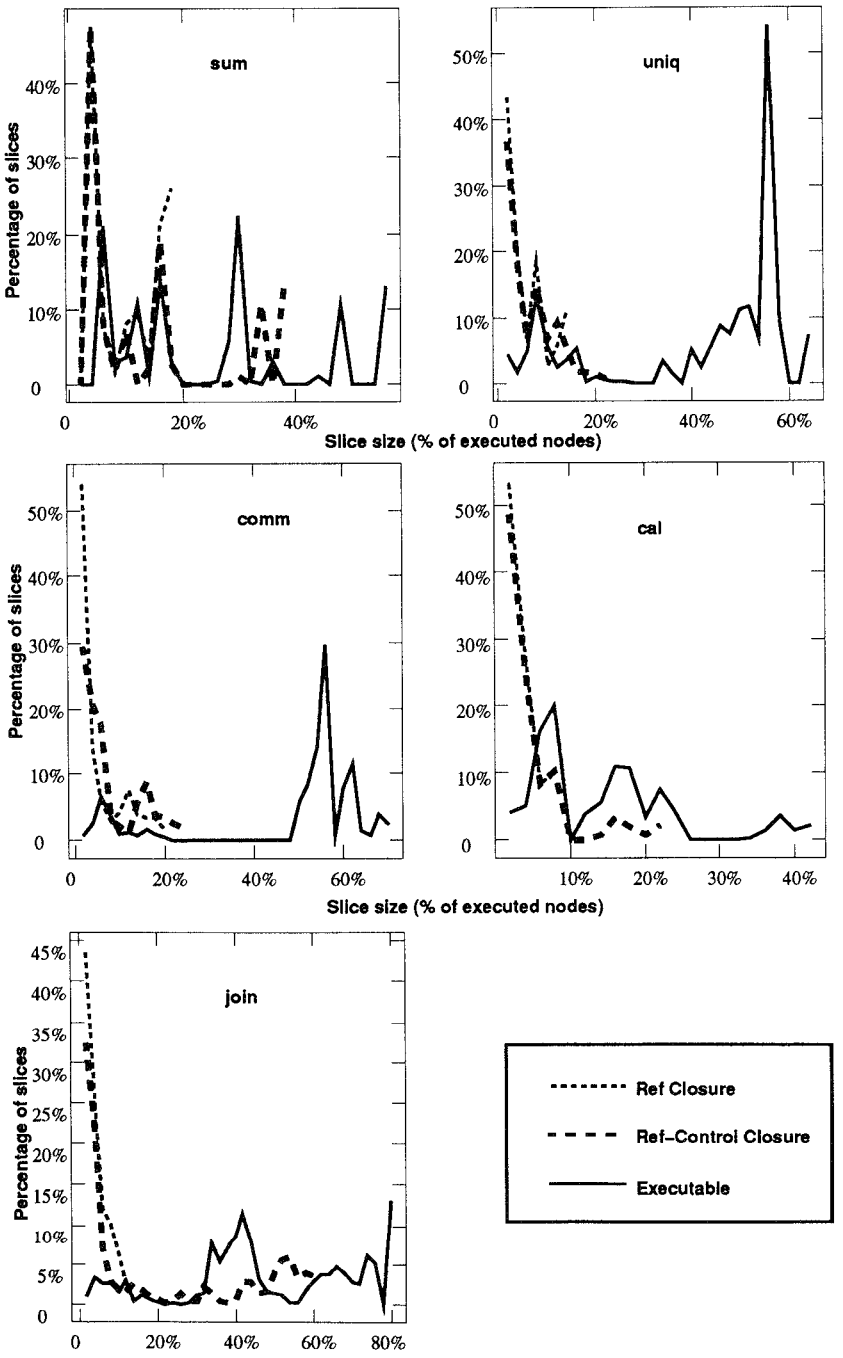

Fig 6. Forward slice distribution.

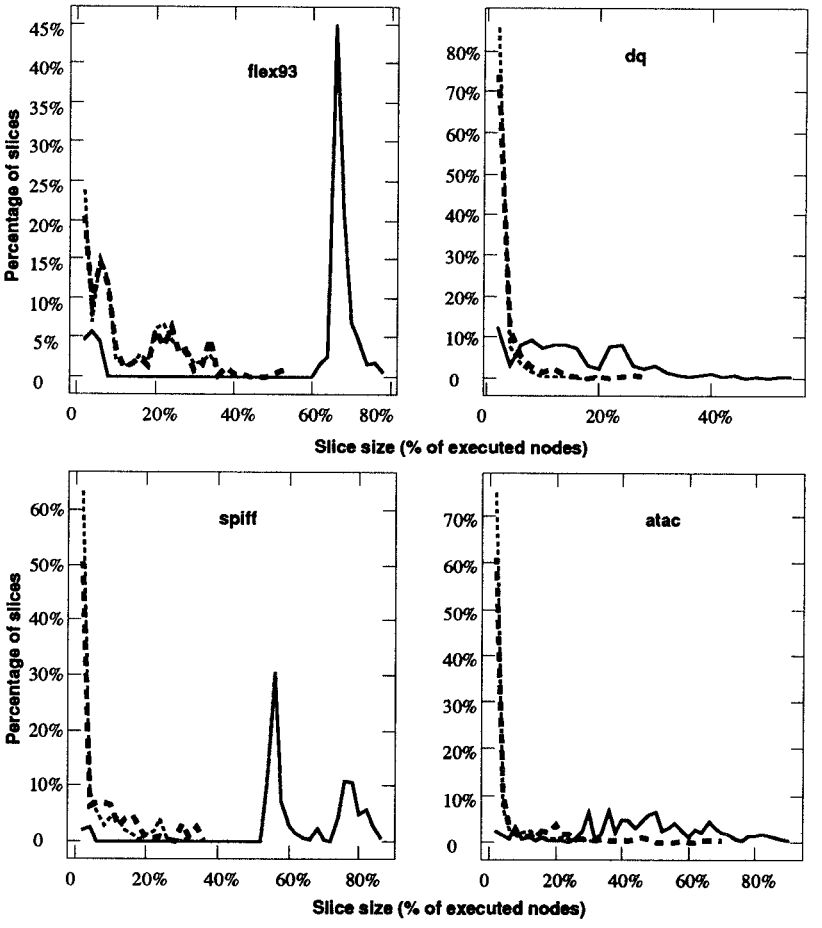

Fig. 6. *Continued*

(2) The peaks at the higher end correspond to slices that capture one or more of the functionalities of the program. In programs that have a major portion of the code involved in every functionality of the program (FLEX93 and SPIFF), there is a rather pronounced peak at the size that includes the relevant portion of the program (excluding some error-handling and exception-handling code). There are smaller peaks further at higher slice sizes for computations that use the results of the core computation. In a program where there are distinct portions of the program that are executed for different executions (ATAC), there are multiple peaks for slices that include the corresponding portions of the program. (The program DQ falls within these two extremes.)

These observations suggest a way (albeit an expensive way) to identify and isolate portions of the program that perform reasonably self-contained subcomputations. Studying the distribution of code corresponding to such slices

in relation to the textual/syntactic divisions of the program may provide useful information about the organization of the code and may be helpful in program maintenance or reengineering efforts.

For suggested applications of slicing, the bimodal distribution actually makes it easier to evaluate the suitability of a type of slicing. If the success of an application depends on more than just trivial slices, then the application *must* provide satisfactory results for slices with sizes at the upper end of the range. Applications such as debugging and testing that can use the smaller closures can expect the slices to be within 10% of the total program size and, hence, can benefit from the use of slicing. Program understanding/specialization tools that require the larger closures will only deal with program fragments that are less than 25% of the size of the entire program, making slicing a feasible technique for such applications. Designers of future applications may refer to the results of this experiment to ensure that their assumptions of slice sizes are reasonable for the intended application.

To answer the second question asked at the beginning of this section, variables in the programs were labeled using *Output*, *Loop Index*, *Array Index*, *Array*, *Local*, *Global*, *Function Argument*, and *Flag*. A variable was assigned all of the categories that were applicable in the local context of its use. The categorization was done manually by reading the program code, and variables did not inherit categories through possible aliasing. FLEX93 was chosen as the biggest program that was manageable for such an exhaustive classification of variables. Note that the categories are not precisely defined and that subjective judgement was used in labeling the variables. Input variables were not considered, since all input to FLEX93 is given through command line arguments and such variables constituted a major portion of the category of flag variables.

For each use of a variable in each category, the slice size was averaged over the slice data from all test runs. The distribution of slices were then computed as in the earlier case for all variables. However, the number of slices for each slice size was normalized as a percentage of the total number of slices in the corresponding category. This distribution was computed for each of the different kinds of slices considered in this study. The distribution of slice sizes with respect to variables in each category was then qualitatively compared with the distribution of slice sizes with respect to all variables.

Only the cases in which the slice distribution for a specific category was *noticeably different* from the overall distribution are displayed in Figure 7. The distribution for the variables that are not displayed in the graph was essentially the same as the overall distribution. The graphs in Figure 7 show similar data as the graphs in Figures 5 and 6, but with some changes for clarity and space efficiency:

—Each rectangle consists of a set of graphs with a common x-axis and the y-axes stacked vertically.

—The slice size distribution in the bottom stack is for all variables and is the same as the distribution in Figures 5 and 6 for the corresponding slice

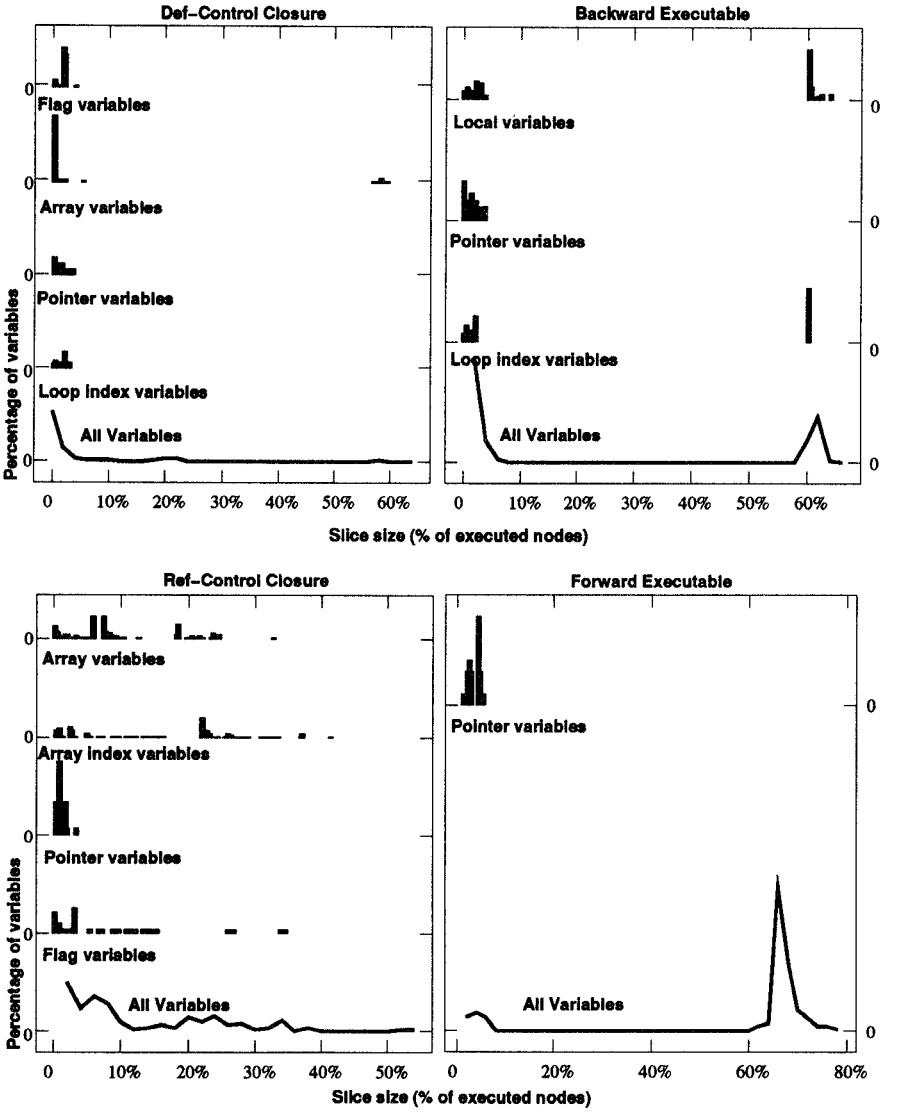

Fig. 7. Slice distribution and variable categories.

type. Since all variables are considered, the percentage of variables on the y-axes is equivalent to the percentage of all slices.

—A bar graph, rather than a line graph, is used for the distribution of slice sizes for each type of variable, since the distribution (unlike that for all variables) is rather sparse with no slices corresponding to most of the slice sizes. A continuous line graph connecting the tops of the bars would present a misleading picture.

The differences were noticeable only for the two data and control closures, for the two executable slices, and only in certain variable categories for each type of slice.

The slice size distribution for variable categories pictured in Figure 7 differs from the overall distribution in two major ways: (1) an absence of slices in specific variable categories for certain slice size ranges that are populated in the overall distribution (e.g., pointer variable distribution in executable slices) and (2) a dominance of slices in size ranges relative to the rest of the distribution that is at variance with the overall distribution (e.g., loop index and local variables have a higher fraction of their backward executable slices at the high end, as opposed to the overall distribution where a higher fraction of the variables have small slices).

None of the pointer variables in this program had large slices. Pointer variables do not appear to have many data dependences (although the objects referred to by the pointers may have a large number of dependences). The program had very little pointer arithmetic. Isolating the role of pointers may prove to be a useful application of dynamic slicing.

8. LIMITATIONS AND FUTURE WORK

The following limitations apply to the prototype implementation:

—Only dependences at the byte level (the lowest addressable memory size) can be accounted for. Programs that map variables to individual bits cannot be sliced.

—Dependence through system calls must be simple and through assignments to memory locations. The semantics of a system call that has complicated side effects cannot be captured by the slicer, and the constructed slices will be smaller than what the correct slices should be. System calls that have dependences through user interaction (interactive programs) or external databases cannot be handled by the prototype.

—**Goto** statements result in incorrect *executable* slices. This limitation is an artifact of the current implementation and not an intrinsic problem for dynamic slicing.

One of the currently running experiments is executing a set of standard UNIX programs over a large number of test cases and constructing a union of program slices for each variable in the program over all test runs to obtain a lower bound on the size of *static slices*. If such a lower bound is significantly small compared to the size of the program, then there is some hope that static analysis techniques may improve enough to provide meaningful slices. If the lower bound happens to be quite large, then static slices may turn out to be useless even if they could be constructed with great precision.

There are a large number of improvements that can be made to the implementation for possible use in production-quality tools. Better logging/tracing techniques, especially at the object-code level, would be highly beneficial and would remove some of the limitations with system calls. The prototype was designed to construct multiple types of slices, and hence, several optimizations that would benefit only some of the slices could not be made. For example, the logging overhead can be considerably reduced if executable slices were not required.

9. SUMMARY AND CONCLUSIONS

This article has reported on the experimental results of using a dynamic slicer for slicing nontrivial applications written in C. The implementation demonstrated the feasibility of building such a tool for various types of slicing. The implementation also indicated that source-level instrumentation for obtaining trace data is not the most satisfactory solution for dynamic slicing. Object-code instrumentation may yield better performance. Empirical data on the size of program slices obtained by using the slicer on several kinds of programs indicate that dynamic program slices for realistic applications are, indeed, reasonably small compared to the entire program. The data demonstrate that it is worthwhile to consider slicing techniques in the applications for which they have been proposed.

Slice sizes vary considerably within a program as well as across programs. The slices tend to have clustered slice sizes with code corresponding to distinct functionalities in the program. The kind of variable that is used as a slicing criterion also has some effect on the size of a slice. Although most categories of variables tend to have a distribution that is similar to the overall distribution of slice sizes, some types of variables (possibly influenced by programming style) seem to deviate noticeably from the overall distribution.

ACKNOWLEDGMENTS

Suggestions, criticisms, and comments from Hiralal Agrawal, Michael Ernst, and the reviewers resulted in significant improvements in the content and presentation style of the article.

REFERENCES

AGRAWAL, H. 1991. Towards automatic debugging of computer programs. Tech. Rep. SERC-TR-103-P, SERC, Purdue Univ., West Lafayette, Ind. Aug.

AGRAWAL, H. AND HORGAN, B. 1990. Dynamic program slicing. In *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 25*, 6 (June), 246–256.

BALL, T. AND LARUS, J. R. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst. 16*, 4 (July), 1319–1360.

COOPER, K. 1985. Analyzing aliases of reference formal parameters. In *Proceedings 12th ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 14–16). ACM, New York, 281–290.

COOPER, K. AND KENNEDY, K. 1989. Fast interprocedural alias analysis. In *Proceedings 16th ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan. 11–13). ACM, New York, 49–59.

ERNST, M. 1994. Practical fine-grained static slicing of optimized code. Tech. Rep. MSR-TR-94-14, Microsoft Research, Redmond, Wash.

HORWITZ, S., PFEIFFER, P., AND REPS, T. 1989. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 24*, 7 (June), 28–40.

HORWITZ, S., PRINS, J., AND REPS, T. 1988. Integrating non-interfering versions of programs. In *Proceedings 15th ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 13–15). ACM, New York, 133–145.

HORWITZ, S., REPS, T., AND BINKLEY, D 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12*, 1 (Jan.), 26–60.

JIANG, J., ZHOU, X., AND ROBSON, D. J. 1991. Program slicing for C—The problems in implementation. In *Proceedings Conference on Software Maintenance* (Sorrento, Italy, Oct. 15–17). IEEE, New York, 182–190.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inf. Process. Lett. 29*, 3 (Oct.), 155–163.

LANDI, W., RYDER, B. G., AND ZHANG, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 28*, 6 (June), 56–67.

LIVADAS, P. E. AND ROY, P. K. 1992. Program dependence analysis. In *Proceedings Conference on Software Maintenance* (Orlando, Fla., Nov. 9–12). IEEE, New York, 356–365.

OTTENSTEIN, K. J. AND OTTENSTEIN, L. M. 1984. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical SDEs. ACM SIGPLAN Not. 19*, 5 (May), 177–184.

VENKATESH, G. A. 1990. The semantic approach to program slicing. In *Proceedings SIGPLAN 91 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 26*, 6 (June), 107–119.

WEIHL, W. E. 1980. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Proceedings 7th ACM Symposium on Principles of Programming Languages* (Las Vegas, Nev., Jan. 28–30). ACM, New York, 83–94.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. SE-10*, 4 (July), 352–357.

Received October 1993; revised June 1994; accepted January 1995