Better Debugging via Output Tracing and Callstack-Sensitive Slicing

Susan Horwitz, Ben Liblit, and Marina Polishchuk

Abstract—Debugging often involves 1) finding the point of failure (the first statement that produces bad output) and 2) finding and fixing the actual bug. Print statements and debugger break points can help with step 1. Slicing the program back from values used at the point of failure can help with step 2. However, neither approach is ideal: Debuggers and print statements can be clumsy and time-consuming and backward slices can be almost as large as the original program. This paper addresses both problems. We present *callstack-sensitive slicing*, which reduces slice sizes by leveraging the series of calls active when a program fails. We also show how slice intersections may further reduce slice sizes. We then describe a set of tools that identifies points of failure for programs that produce bad output. Finally, we apply our point-of-failure tools to a suite of buggy programs and evaluate callstack-sensitive slicing and slice intersection as applied to debugging. Callstack-sensitive slicing is very effective: On average, a callstack-sensitive slice is about 0.31 time the size of the corresponding full slice, down to just 0.06 time in the best case. Slice intersection is less impressive, on average, but may sometimes prove useful in practice.

Index Terms—Static program slicing, callstack-sensitive analysis, points of failure, output tracing and attribution.

1 Introduction

An erroneous program can fail in two different ways: 1) It can crash by causing a segmentation fault, throwing an uncaught exception, etc., or 2) it can produce bad output, but continue to execute. In case 1, information about the point of failure is usually provided, often including the line of code that caused the crash and a stack trace. In case 2, although the user might be able to identify the first instance of bad output, it can be a nontrivial task to find the actual point of failure (the statement that produced that output) and to determine what the callstack configuration was at the time of failure.

For example, consider the C program shown in Fig. 1a, which is supposed to compute and print either the product or the sum of the numbers from 1 to 10, depending on the command line argument. Function `getChoice` is implemented incorrectly, which causes the product to be printed when the sum is expected, and vice versa.

For a run with command line argument `sum`, the program prints `val: 3628800`. The point of failure is line 25, but that is not obvious; in fact, the person testing the program may assume that the point of failure is line 19, where the value of the sum is printed.

The first contribution of this paper is the description of a suite of tools that we have designed and implemented to help provide point-of-failure information for programs that fail by producing bad output. These tools allow a programmer to associate each byte of output produced by a C program (running under Linux) with the code that wrote that output. For example, for the program in Fig. 1a, our tools allow a programmer to pinpoint line 25 as the point of failure when the command line argument is `sum`. If the program is compiled with debug information, the sequence of active function calls at the time the output was written is also provided. These tools are discussed in Section 2.

Having point-of-failure information is a useful first step in debugging, but it is rare that the point of failure itself is the code that needs to be modified to correct the problem. In our example, although the point of failure is line 25, the actual bug is the incorrect implementation of function `getChoice` on lines 1-4.

Weiser [1] suggests that when programmers debug a program, they often use *backward slicing*: They start from the point of failure and work backward, following data and control dependences, examining all codes that might affect the behavior at the point of failure until they find the bug. Because a backward slice excludes code irrelevant to the point of failure, this process allows programmers to focus only on relevant code and thus speeds up the task of locating the actual bug.

In Fig. 1a, stars indicate the program components that are in the backward slice from line 25. Excluding code that has no effect on the value printed at line 25 would probably help the programmer find the error in function `getChoice` more quickly. However, in this example, the whole program is so small that slicing is not really needed for debugging. For large programs, slicing can be of limited value because backward slices can be very large. For example, the *Wisconsin Program-Slicing Tool 1.1 Reference Manual* [2] reports that, in six C programs, the average size of a backward slice ranged from 19 to 97 percent of the size of the whole program, and Binkley and Harman [3] report that, in 43 C programs, the average size of a backward slice ranged from 7.5 to 62 percent of the size of the whole program. (In the *Wisconsin Program-Slicing Tool 1.1 Reference*

- *S. Horwitz and B. Liblit are with the Department of Computer Sciences, University of Wisconsin–Madison, 1210 W Dayton St., Madison, WI 53706-1685. E-mail: {horwitz, liblit}@cs.wisc.edu.*
- *M. Polishchuk is with Microsoft Corporation, 1 Microsoft Way, Redmond, WA 98052. E-mail: marinapo@microsoft.com.*

Manuscript received 15 Feb. 2009; revised 28 July 2009; accepted 30 July 2009; published online 1 Oct. 2009.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-02-0035.
Digital Object Identifier no. 10.1109/TSE.2009.66.

```
** [ 1] int getChoice(char* ch) {
** [ 2]   if (strcmp(ch, "sum") == 0)
** [ 3]     return 1;
** [ 4]   else return 0;
     5  }
     6
** [ 7] void main(
     8          int argc,
** [ 9]         char* argv[]) {
    10    int sum = 0;
**  11    int prod = 1;
** [12]   int k = 1;
** [13]   int ch = getChoice(argv[1]);
** [14]   if (ch == 0) {
    15      while (k<11) {
    16        sum += k;
    17        k++;
    18      }
    19      printf("val: %d\n", sum);
    20    } else {
**  21      while (k<11) {
**  22        prod *= k;
**  23        k++;
    24      }
**  25      printf("val: %d\n", prod);
    26    }
    27  }
```

(a)

```
[ 1] void print(
[ 2]          char *msg,
[ 3]          int val) {
[ 4]   printf("%s %d\n", msg, val);
  5  }
  6
[ 7] int getChoice(char* ch) {
[ 8]   if (strcmp(ch, "sum") == 0)
[ 9]     return 1;
[10]   else return 0;
 11  }
 12
[13] void main(
 14          int argc,
[15]         char* argv[]) {
 16    int sum = 0;
 17    int prod = 1;
[18]   int k = 1;
[19]   int ch = getChoice(argv[1]);
[20]   if (ch == 0) {
 21      while (k<11) {
 22        sum += k;
 23        k++;
 24      }
 25      print("val: ", sum);
 26    } else {
 27      while (k<11) {
 28        prod *= k;
 29        k++;
 30      }
 31      print("val: ", prod);
 32    }
 33  }
```

(b)

Fig. 1. (a) Example of an erroneous program: function `getChoice` returns the wrong value, which causes the product to be printed when the sum is expected, and vice versa. For a run with command line argument `sum`, the point of failure is line 25, and for a run with command line argument `prod`, the point of failure is line 19. Stars indicate the lines that are in the slice back from line 25. Boxed line numbers indicate the lines that are in the intersection of the slices from the two failure points (lines 19 and 25). (b) Similar example but with a separate `print` function. Boxed line numbers indicate the lines that are in the intersection of two callstack-sensitive slices from line 4: one with an active call at line 25 and the other with an active call at line 31.

Manual [2], slices were taken starting from each call to a library function, while Binkley and Harman [3] took slices starting from every line of source code.)

One reason that backward slices can be so large is that the slice back from a statement S in a function f takes into account all possible calling sequences from `main` to f. For example, consider the program shown in Fig. 1b, which is the same as the original example in Fig. 1a except that the two calls to `printf` have been replaced by a call to a user-defined `print` function. In this version, the point of failure for input `sum` is line 4. Unfortunately, in this case, the backward slice from that line is the entire program because function `print` is called to print both the sum and the product.

A desirable backward slice for this example is one that isolates the execution of line 4 that results from the call to `print` at line 31, and not the one at line 25. One possibility is to use a *dynamic slice*, first defined by Korel and Laski [4]. However, as discussed in Section 4, the program instrumentation required to support dynamic slicing can introduce slowdowns of 1-3 orders of magnitude [5], which may be unacceptable for large programs.

An alternative, more efficient kind of slice that takes into account a given sequence of active function calls was first introduced by Binkley [6] (where it was called a *calling-context slice*). Krinke [7] identified a shortcoming of Binkley's algorithm and provided a more precise version (where it was called a *context-restricted slice*). We refer to these slices in this paper as *callstack-sensitive slices*.

While the full slice of the program in Fig. 1b from line 4 is the whole program, the callstack-sensitive slice from that line with one active call at line 31 excludes the loop on lines 21-24, making it clear that the error is unrelated to the way the sum is computed and helping the programmer find the bug in function `getChoice`.

Detailed algorithms for full and callstack-sensitive slicing are provided by Horwitz et al. [8] and Krinke [7], respectively, and are outside the scope of this paper. For both techniques, slicing back from a program component C involves a backward traversal of the edges of the System Dependence Graph representation of the program. The important difference arises when component C is in a procedure `P` other than `main`. In that case, the full slice follows edges back to all call sites that call `P`, while the callstack-sensitive slice only follows edges back to the call site that is at the top of the given callstack. If `P` was reached by a sequence of calls from `main`, e.g., call `P1`, call `P2`,..., call `Pn`, call `P`, the full slice continues to follow edges back to all call sites that call `Pn`, `Pn-1`, etc., while the callstack-sensitive slice only follows edges back to the call sites in the callstack. For the example program in Fig. 1b, the full slice back from line 4 includes both lines 25 and 31 (the two call sites that call function `print`). The callstack-sensitive slice, given the callstack that contains the call to `print` made on line 25, only follows edges back to the call site on line 25, and thus, does not include the call to `print` made on line 31.

An obvious question is whether callstack-sensitive slicing yields enough of a reduction in slice size to produce substantially more useful backward slices during debugging. Binkley provided no evaluation of callstack-sensitive slicing. Krinke did two evaluations. The first [7] evaluated slicing using two moderate-sized test programs (`ctags` and `patch`). For each program, a "characteristic" execution was performed, and both full and callstack-sensitive slices were computed for each function call. For these experiments, the average full-to-callstack-sensitive slice size ratios were just 1.10 and 1.01 for the two programs. The second evaluation [9] was performed on 14 moderate-sized programs, including `ctags` and `patch`. In this case, the programs were not actually run. Instead, callstacks of depth 1-12 were generated using the programs' static callgraphs. This study found greater size reductions for callstack-sensitive slices.

However, there are several problems with Krinke's studies. The first study was clearly too limited since it used just two programs and just one execution of each. The second study used callstacks based on the static callgraph; there is no guarantee that those callstacks could even arise in an actual execution. Furthermore, in the context of debugging, we are not interested in slices from all function calls, only in slices from points of failure. These may lead to very different results, as Krinke [9] himself warns:

> callstacks and criteria in practice will have different properties than artificially generated callstacks and criteria.

The second contribution of this paper is a series of experiments that addresses these shortcomings of Krinke's work. Our experiments, discussed in Section 3, were carried out on a total of 285 buggy versions of 11 programs obtained from the Software Artifact Infrastructure Repository [10], [11]. For each point of failure, we compared the sizes of the whole program, the full slice, and the callstack-sensitive slice.

For nontoy programs, we found that the full slice ranged from 3 to 87 percent of the size of the whole program, with mean 73 percent and median 80 percent. The size of a callstack-sensitive slice ranged from 3 to 80 percent of the size of the whole program, with mean 35 percent and median 28 percent. In nearly all cases (354 out of 401 different failure points across multiple nontoy applications), the callstack-sensitive slice was strictly smaller than the corresponding full slice. In the remaining cases, the callstack-sensitive and full slices were the same size. On average, for these programs, a callstack-sensitive slice was about 0.31 time the size of the corresponding full slice. In the best case, stack sensitivity yielded a slice just 0.06 time the size of the corresponding full slice.

Callstack-sensitive slicing can be even more helpful when a single bug causes different test inputs to trigger failures at different points or at the same point but with different sequences of active function calls. For example, if the program in Fig. 1a is tested with the command line argument `sum` and then `prod`, bad output will be produced at line 25 and then at line 19. If both errors are due to the same bug, the erroneous code must be in the slices back from both lines and thus we can make it even easier for the programmer to locate the bug by displaying the intersection of those two slices: the boxed lines of code in Fig. 1a.

If the program in Fig. 1b is tested with the command line argument `sum` and then `prod`, bad output will be produced at line 4 in both runs. In the first case, the active function call when the wrong output is printed is the one at line 31 and, in the second case, it is the one at line 25. There is only one full backward slice from line 4, but there are two (different) callstack-sensitive slices. Boxed lines in Fig. 1b indicate the intersection of those two slices.

Of course it is not usually known a priori that a program has just one bug. However, previously developed techniques [12], [13], [14] can be used to cluster runs so that failures in each cluster are likely to be due to the same bug. Slice intersections can then be computed for all failures in the same cluster.

Section 3 includes data about the efficacy of slice intersection. Although these results are less dramatic than the comparison of full and callstack-sensitive slicing, we found that slice intersection was occasionally very beneficial.

2 Mapping Bytes of Output to Source Code

This section describes our tools for C programs that allow a programmer to pinpoint code that wrote each byte of output produced by the program and to use that information for debugging.

In the following descriptions, the *source code information* associated with a byte of output is the name of the source file that caused the output to be written, the line number of the write, and the file names and line numbers of the function calls active at the time the output was written. We have implemented the following tools:

The *Trace Tool* records output information in a trace file while the program runs.

The *Interactive Mapping Tool* allows a programmer to browse the program's output, and for each selected output character, to obtain the corresponding source code information from the trace file.

The *What-Wrote Tool* finds and displays the associated source code information for an output stream and byte number of interest.

The *Compare-Traces Tool* takes two programmer-specified traces as input (e.g., a reference trace representing correct output and a buggy trace containing bad output),

TABLE 1
Nonstandard Tools and C Extensions Used by the Trace Tool

Task	Mechanism
apply tracer to existing code	shared library and `$LD_PRELOAD`
find standard file open functions	`dlsym(RTLD_NEXT, ...)`
create file proxy with custom handlers	`glibc`'s `fopencookie` function
pre-`main` trace initialization	`gcc`'s `constructor` attribute
post-`main` trace finalization	`gcc`'s `destructor` attribute
find saved PC addresses on call stack	`glibc`'s `backtrace` function
resolve raw PC addresses to source locations	`addr2line` command

finds the first byte of output at which the traces differ, and prints the source code information associated with that byte in the buggy trace.

In the following sections, we describe the Trace Tool, outline the format of the trace files, describe the three tools that make use of the trace files, and finish with a discussion of the tools' performance.

2.1 Output Trace Collection

The Trace Tool redefines the standard C library functions that manipulate files. These redefinitions are provided in a shared library that can be preloaded into any program's code space. Thus, the Trace Tool can be applied to arbitrary existing code without recompilation. The redefined functions create a trace file (formatted using XML) while the program runs. The trace file records each output event: opening an output file, writing to an output file or to `stdout` or `stderr`, and closing an output file. The record of each write event includes the actual bytes of output written, a representation of the current callstack, and information about the code that caused the write. When the Interactive Mapping, What-Wrote, or Compare-Traces Tools are run, they use the information in the trace file to allow the programmer to select individual bytes of output and to identify the corresponding source code and stack trace.

The Trace Tool takes advantage of several features and documented extensions of the GNU C compiler (`gcc`) and C library (`glibc`) running under Linux. Table 1 summarizes the extensions used; refer to appropriate platform documentation for further details [15], [16], [17].

2.1.1 Opens

Our shared library redefines the standard C file-opening calls: `fopen`, `fopen64`, and `fdopen`. Our implementations call the corresponding standard implementations from `glibc` to actually open the file. We then create a file proxy object with customized handlers for write, seek, and close operations. This file proxy is used to trap subsequent output operations and augment the trace file. The file proxy is set to be nonbuffered so that our custom handlers will be called immediately by each output routine without waiting for a buffer flush. (Otherwise, we would misidentify the code responsible for output between flushes.)

The file proxy includes a callback state structure as follows:

1. the real file (`FILE *`) that we are proxying,
2. the real file's current byte offset (initially zero unless appending),
3. an integer ID number unique to this opened-file instance.

After creating the file proxy, we append a "file opened" record to the trace file. This record gives the unique ID number for this file, the associated numeric file descriptor, and the name of the file opened (omitted in the case of `fdopen`). We then return the file proxy to the caller, which treats it as though it were a standard C `FILE *`.

Two important files are not opened in this manner: `stdout` and `stderr`. The standard output and error streams are already set up by the C runtime library before our tracer is active. Instead, we create a small fragment of initialization time code that wraps these two files in proxies as we would for any `fdopen` call. This initialization code runs before `main`, and therefore, no application output is lost.

Note that we proxy `FILE *` file objects, not raw file descriptors. Output sent directly to a file descriptor (e.g., via `write` instead of `fwrite`) is not traced. Writes via memory-mapped files are likewise omitted from the trace.

2.1.2 Writes

The file proxy's custom write handler performs the bulk of trace recording work. When the program sends output to a file proxy, such as by calling `fprintf` or `fwrite`, our custom write handler is called to consume the data. The handler receives the callback state structure described above plus a buffer containing the raw data to be written. Note that any output formatting, such as via `fprintf`, occurs before our handler is called. This saves us from having to reimplement that nontrivial formatting behavior ourselves.

In our custom write handler, we first perform the actual data write to the real file and then record information about this write in the trace file. That information includes the unique ID number for this file, the beginning and ending byte offsets of the write, and the raw data written. We also collect and record the vector of saved program counter addresses on the callstack, thereby identifying the code location making the output call as well as the entire call chain from program entry (`main`) down to that point. Finally, the current byte offset in the callback state structure is updated. We track offsets ourselves rather than relying on `ftell` because the latter is unreliable on nonseekable streams such as sockets and pipes.

2.1.3 Seeks and Closes

Our custom seek handler performs the actual seek on the real file and also updates the current byte offset as recorded in the callback state structure. The seek is not explicitly noted as an event in the trace, but will affect the offsets reported for the next write event.

Our custom close handler notes the close event in the trace, performs the actual close on the real file, and releases resources associated with the file proxy object and its callback state structure.

2.2 Output Trace File Format

Each run of a program records its output in a single trace file. For multithreaded programs, this file becomes a serialization point for all output. We designed the trace file format to be detailed, flexible, and easy to process. Our implementation is not tuned for performance, and alternate formats that emphasize compactness and efficiency are certainly possible.

Our traces are formatted as XML documents, with the top-level structure consisting of a sequence of `open`, `write`, and `close` event records. During trace collection, we open the trace file and print its initial XML declarations and opening `<output-trace>` tag as part of pre-`main` initialization. Post-`main` termination code prints the ending `</output-trace>` tag and closes the trace; signal handlers ensure that this finalization takes place even in the event of a crash.

Each stack frame in the output trace is initially of the form `<frame pc = "0x ..."/>`. That is, it gives a hexadecimal program counter address but no source file, line, or function information. A stand-alone frame resolver tool traverses the trace and adds `file`, `line`, and `function` attributes by examining the symbol table and debug information in the traced binary. If the program was compiled without debug information, function names are usually still available. If the program has been stripped (so that neither function names nor source locations are available), only the numeric `pc` attribute is retained.

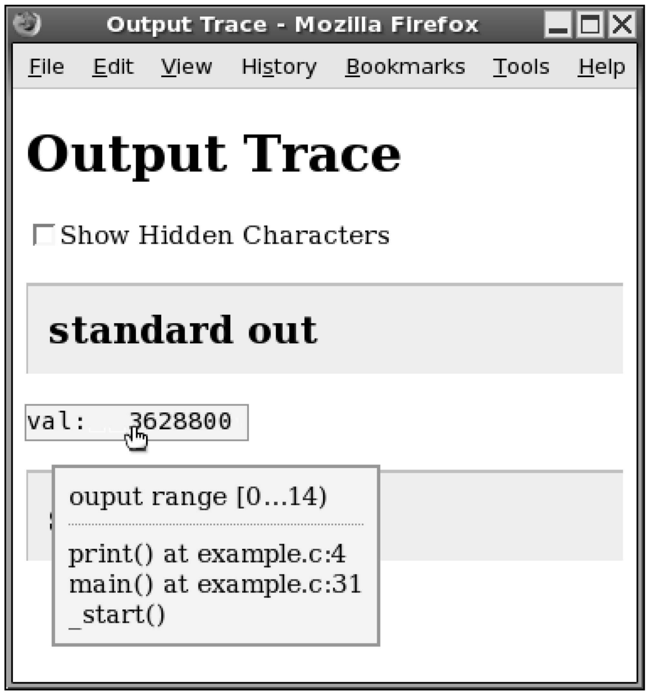
2.3 Output Trace Exploration and Analysis

We now describe a suite of trace processing tools. The Interactive Mapping, What-Wrote, and Compare-Traces Tools help a programmer explore and analyze output traces produced by the Trace Tool.

Fig. 2 shows the **Interactive Mapping Tool**, hosted in any full-featured Web browser, applied to the trace produced by running the program in Fig. 1b with input `sum`. The hand-shaped mouse pointer is hovering over the first output digit. The highlighted block of text surrounding this character indicates that the entire "`val: 3628800`" line was printed as a single operation. If the program had performed multiple output operations, only those bytes written at the same time as the selected character would have been highlighted.

The box immediately below the mouse pointer gives the range of bytes in the selected block and prints the source code information associated with those bytes, as well as the same information for each active function call. In this case, the 14 bytes that include the selected character were printed on line 4 of `print`, which was, in turn, called from line 31 of `main`, and both functions are in `example.c`. Standard error follows standard out, but is obscured in this view by the stack trace. If the program wrote to any other files, these would also appear in the trace in the order that they were opened. The "Show Hidden Characters" check box, if selected, uses special typographic symbols to reveal nonprinting characters such as spaces, tabs, and newlines.

Given an output file or stream (e.g., `stdout`) and a byte of interest (e.g., 6), the **What-Wrote Tool** prints the source code information associated with that byte. For example, to


Output Trace - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Output Trace
Show Hidden Characters
standard out

```
val:   3628800
```

ouput range [0...14)

print() at example.c:4
main() at example.c:31
_start()

Fig. 2. Interactive browsing of output trace for code in Fig. 1b run with input `sum`. The "`3`" character in standard output is selected.

find the code and surrounding stack context that printed the sixth byte (counting from zero) printed to standard output by the code in Fig. 1b:

```
% what-wrote  --stdout = 6   example.xml
example.c         4          print
example.c         31         main.
```

If the selected byte was written multiple times, such as via `fseek`, we provide the information for the last write, as this produced the final value present in the output after the program completed.

The **Compare-Traces Tool** finds the first point of divergence between a buggy run's output and a correct, reference run's output. It then displays the source code information for the buggy code that printed this bad byte. For example, to find the code from Fig. 1b that printed the first incorrect byte:

```
% compare-traces reference.xml  example.xml
example.c             4         print
example.c             31        main.
```

The Compare-Traces Tool ignores changes in output grouping: For example, if the reference trace prints "`val: 3628800`" in one operation while the buggy trace first prints "`val:`" and later prints "`3628800`," this is not considered a divergence. The tool also takes overwriting into account: An incorrect byte is only treated as a point of difference if that byte is not subsequently "corrected" by a later write at the same stream position.

At first, one might think that the functionality of the Compare-Traces Tool could be achieved by combining the Unix `diff` utility with the What-Wrote Tool. This is true except when the program writes to multiple output streams. In that case, `diff` has no way to know the order in which output bytes were written to the different streams. Because our Trace Tool produces a single trace file that keeps track of the bytes of output written to *all* streams, the Compare-Traces Tool is able to correctly identify the first point of difference, regardless of the output stream in which it occurs.

Although our present interest is in attributing erroneous output to the faulty code that produced it, these tools may be useful for a variety of other tasks as well, including the following:

- As an application evolves, one may wish to change the format of what was previously correct output; output tracing can identify the code that should be modified, and can prevent surprises by identifying other output also produced by the code being changed.
- Similarly, these tools can help debug internationalization errors that "leak" English text in what should be a non-English localization of a user interface.
- The tools support a limited form of learning by example: Given some output in a desired format, we can pinpoint the corresponding code and thereby reveal the (often obscure) format string that can then be used to produce similar output elsewhere.
- An output trace can help recognize similar output produced by distinct pieces of code, which may, in turn, guide refactoring to centralize common output functionality.

2.4 Tool Performance

Here, we discuss the efficiency of our tools. How much overhead is incurred when a program is run using our versions of the file manipulation functions and how long does it take to find the source code line that produced a given byte of output?

During trace collection, every byte of normal output is replicated in the trace, along with information about the current program point and callstack. For the test applications used in our experimental evaluation (Section 3), we find that output tracing roughly doubles execution time, with slowdowns ranging from 1.94 to 2.78. Note that the slowdown can vary widely, depending on the application: Output tracing imposes zero overhead for internal computations that do not perform output.

Our trace exploration and analysis tools use scripted sequences of XPath queries to resolve mapping requests. Although our verbose trace format is not tuned for speed, the core XPath implementation is provided by the fairly fast `libxml2` C library [18]. The What-Wrote Tool can resolve batch queries on a large (1.3 megabytes) trace in roughly 0.6 second. Interactive browsing for traces of moderate size is, for all practical purposes, instantaneous.

3 Experimental Results

The goal of our experiments was to assess how effective callstack-sensitive slicing and taking intersections of slices are likely to be in the context of program debugging. Therefore, our methodology was as follows:

1. Find buggy versions of C programs with test inputs that cause failures (either crashes or erroneous output).
2. Run the programs. Find the points of failure and the callstacks at those points.
3. Compute full and callstack-sensitive slices from the points of failure, and compare the slice sizes.
4. For each program with test inputs that cause failures at different points, compare the size of the smallest full slice from one of those points with the size of the intersection of the full slices from all of those points (and similarly for the callstack-sensitive slices).
5. For each program with test inputs that cause failures at the same point but with different callstacks, compare the size of the smallest callstack-sensitive slice from that point with the size of the intersection of all callstack-sensitive slices from that point.

To compute slices, we used the CodeSurfer slicing tool [19]. CodeSurfer builds a *System Dependence Graph* [8], [20] representation of a C program and uses the slicing algorithm of Horwitz et al. [8] to produce full slices. It also provides a Scheme API, which we used to implement callstack-sensitive slicing. The slices produced by CodeSurfer are sets of nodes of the underlying System Dependence Graph.

CodeSurfer provides a mapping from graph nodes to source code lines. We use this mapping to determine the sizes of programs and slices, which are reported in the following sections. For *program sizes*, we count the number of source code lines that correspond to at least one node in the program's System Dependence Graph. For *slice sizes*, we count the number of source code lines that correspond to at least one node in the slice. Some examples of source code lines that will not be counted in either program or slice size are lines that contain only comments, preprocessor directives, curly braces, and `typedefs`.

3.1 Test Programs

The buggy C code that we used comes from the Software-artifact Infrastructure Repository [10], [11]. Table 2 provides information about the programs. The first seven constitute the Siemens suite: a set of small, toy programs (all less than 1,000 lines of code) originally collected by Hutchins et al. [21]. For each of the programs in the Siemens suite, there are one bug-free "reference" version and from 7 to 41 additional versions with introduced bugs. The eighth program, `space`, is an application mentioned in work by Vokolos and Frankl [22] and Wong et al. [23]. For `space`, there are one reference version and 34 buggy versions. In this case, the bugs are real errors that were identified during the testing and operational use of the program. The final three programs are Unix utilities. For each of these programs, there are five reference versions that are actual field releases of that program. For each reference version, there are a number of additional versions with introduced bugs.

The information in Table 2 is as follows:

Number of code lines. Average number of lines of source code in each buggy version of the application. As discussed above, we count only lines that correspond to at least one node in the program's System Dependence Graph.

Number of buggy versions. Number of nonreference variants that produce at least one crash or bad output failure on the application's test suite. For the Unix utilities, this is the total number of buggy versions across all of the reference versions.

Total number of failures. For each program, this is the total number of failures across all buggy versions. Each run of a buggy version can have 0, 1, or 2 failures: A run that

TABLE 2
Information about the Test Programs Used in the Experiments

Program	Number of Code Lines	Number of Buggy Versions	Number of Failures	
			Total	Unique
`print_tokens`	364	7	484	14
`print_tokens2`	266	10	2,235	56
`replace`	393	31	3,533	74
`schedule`	223	9	799	16
`schedule2`	199	9	296	23
`tcas`	101	41	1,626	41
`tot_info`	184	23	1,924	32
`space`	4,331	34	71,770	99
`flex`	5,287	54	13,711	98
`grep`	4,544	20	4,382	101
`gzip`	2,718	47	7,532	103

produces the same output as the reference version and does not crash counts as 0 failures. A run that produces bad output but does not crash, or that crashes without having produced bad output, counts as 1 failure. A run that produces bad output then crashes counts as 2 failures.

Number of unique failures. For each buggy version of a program, there may be many inputs that cause the program to fail at the same point and with the same active callstack. For debugging purpose, all of those failures are associated with the same full slice and the same callstack-sensitive slice. Therefore, in the graphs of Figs. 3, 4, 5, and 6, all of those failures are represented by just one point on the x-axis. This column states how many distinct x-axis points there are for each program.

3.2 Comparisons of Callstack-Sensitive Slice Sizes and Full Slice Sizes

In this section, we consider whether using callstack-sensitive slices can significantly reduce the amount of code that a programmer has to examine during debugging. Fig. 3 shows how much larger full slices are than callstack-sensitive slices for our test programs. There is one graph for all of the Siemens applications combined, and one graph each for `space`, `flex`, `grep`, and `gzip`. As explained above, each point on the x-axis corresponds to a unique point-of-failure/callstack pair for one buggy program version. For example, if one buggy version fails at the same point and with the same callstack under 10 different inputs, then all 10 of those failing runs are represented by one tick mark in the graph. Each graph shows the ratio of the size of the full slice from the point of failure to the size of the callstack-sensitive slice from that point. In each graph, the points (ratios) are sorted by size. Note that the scale of the y-axis is different for each of the five graphs.

For the seven Siemens programs, there is generally little advantage to using callstack-sensitive slicing. There are 14 cases where the full slice is more than 1.3 times the size of the corresponding callstack-sensitive slice. On average, though, full slices are 1.03 times the size of the corresponding callstack-sensitive slice. In other words, on average, stack sensitivity shrinks a slice only very slightly. However, the Siemens suite consists entirely of small, toy programs, and thus, these results are not likely to predict the effects of stack sensitivity when slicing real code.

The results for the other test programs are more encouraging. The best results are for `grep` and `gzip`. In 58 out of 101 cases for `grep`, the ratio is over 4, and the maximum ratio is 17.24. In 46 out of 103 cases for `gzip`, the

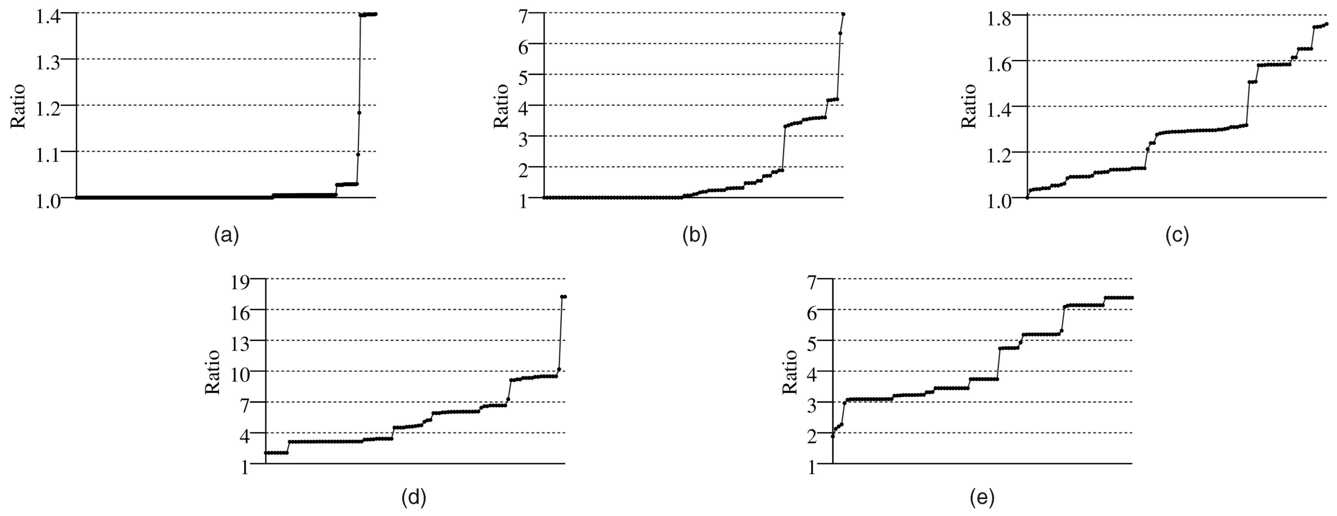


Fig. 3. Ratio of size of full slice to size of callstack-sensitive slice. Note that the scale of the y-axis is different for each graph. (a) All Siemens applications, (b) `space`, (c) `flex`, (d) `grep`, and (e) `gzip`.

```
1   void c() {
2     printf("in c\n");
3   }
4
5   void b() {
6     printf("in b\n");
7   }
8
9   void a() {
10    printf("in a\n");
11  }
12
13  void main() {
14    a();
15    b();
16    c();
17  }
```

(a)

```
1   void c() {
2     printf("in c\n");
3   }
4
5   void b() {
6     c();
7   }
8
9   void a() {
10    b();
11    c();
12  }
13
14  void main() {
15    a();
16    b();
17    c();
18  }
```

(b)

Fig. 4. Example programs that illustrate that the benefits of callstack sensitivity depend on the number of calls to the functions in the given callstack.

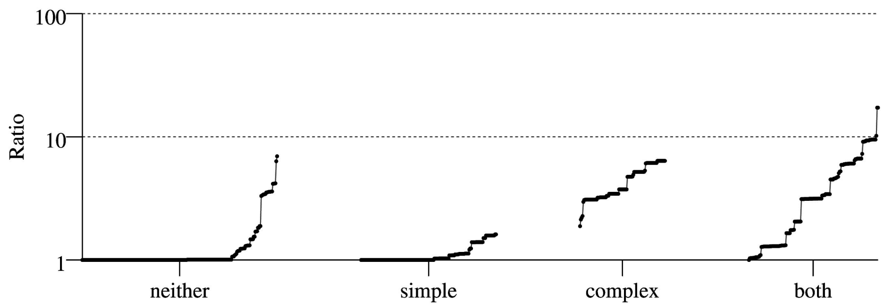


Fig. 5. Ratio of size of full slice to size of callstack-sensitive slice for all applications, grouped by types of recursion present.

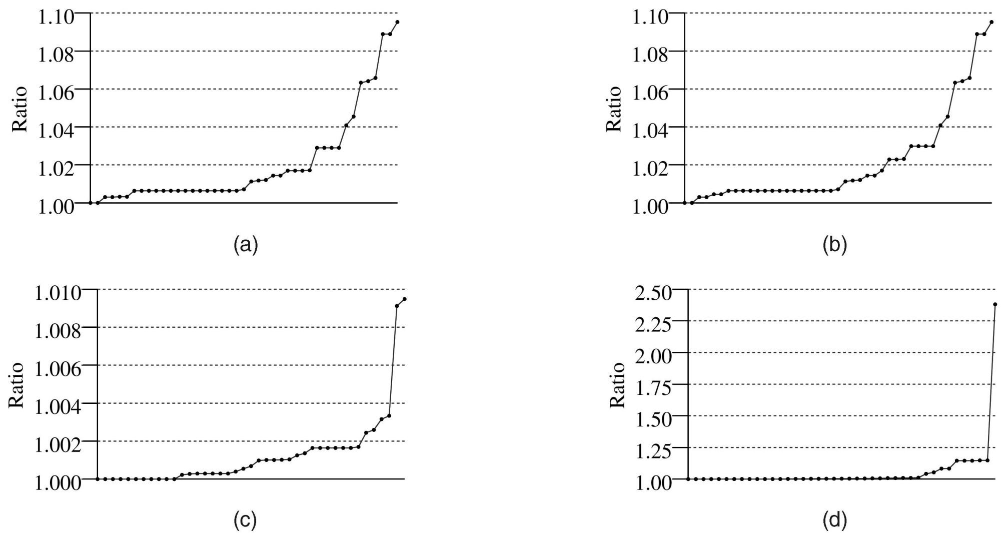


Fig. 6. Ratio of size of the smallest single slice to size of the intersection for applications with failures at multiple different points. (a) Siemens applications using full slices. (b) Siemens applications using callstack-sensitive slices. (c) Non-Siemens applications using full slices. (d) Non-Siemens applications using callstack-sensitive slices.

ratio is over 4 and the maximum ratio is 6.38. The average ratio across all non-Siemens tests is 3.22 (the average ratio is 1.72 for `space`, 1.30 for `flex`, 4.33 for `gzip`, and 5.42 for `grep`), and the median ratio across all non-Siemens tests is 3.09 (the median ratio is 1.10 for `space`, 1.29 for `flex`, 3.74 for `gzip`, and 4.65 for `grep`).

The factor that affects the power of callstack sensitivity is the total number of calls to the functions in the callstack. This is because the full slice will include all calls to those functions, while the callstack-sensitive slice will include only the calls that match the given callstack. In general, therefore, the deeper a program's callstacks can be and the more calls there are to the program's functions, the more helpful stack sensitivity will be (i.e., the smaller the callstack-sensitive slice will be relative to the full slice). To illustrate this, consider the two programs shown in Fig. 4. In the program in Fig. 4a, each function has only 1 call site, and the maximum depth of the callstack is 1. Therefore, for every line of this program, the full and callstack-sensitive slices are exactly the same. In the program in Fig. 4b, function `c` has three call sites, and function `b` has two call sites. The full slice from line 2 is the whole program, but the callstack-sensitive slice from line 2 with the callstack (lines 11 and 15) (of depth 2) includes only the lines whose line numbers are enclosed in boxes.

The effects illustrated in Fig. 4 suggest that the length of the longest acyclic path in a program's callgraph and the number of call sites for each function might be good predictors of the benefits of callstack-sensitive slicing. To investigate this hypothesis, we gathered data for the reference versions of all of our test programs. We found that in the Siemens suite, the longest acyclic callgraph path (i.e., the maximum callstack depth for a sequence of nonrecursive calls)[1] is 8, the average number of calls to a function ranges from 1.3 to 2.2, and no function is called from more than 18 call sites. In contrast, for the other test programs, maximum acyclic callgraph path length ranges from 13 to 20, the average number of calls to a function ranges from 3.0 to 6.4, and the maximum number of calls to a function ranges from 24 in `grep` to 198 in `flex`. Therefore, it is not surprising that callstack-sensitive slicing provides much greater benefit for the more complex programs than it does for the simpler programs of the Siemens suite.

These static metrics let developers predict whether callstack-sensitive slicing is likely to be beneficial before running a program. However, it is also worth considering whether such measures comport with runtime behavior. For the Siemens suite, callstacks at distinct points of failure average 2.4 calls deep (minimum 1, median 2, maximum 8) including the point of failure itself. Across all distinct points of failure in all non-Siemens applications, the average callstack depth roughly doubles to 4.9 calls (minimum 1, median 4, maximum 16). This is consistent with the maximal acyclic depths found via static callgraph analysis above.

Given the low overhead of our output tracing tools and the slice size reductions seen for the more complex test applications, we conclude that callstack-sensitive slicing is a sensible tool for programmers to employ when debugging real applications.

3.3 Impact of Recursion

Krinke [9] defines three types of recursions as follows:

Neither: no recursion. Present in some versions of Siemens applications and all versions of `space`.

Simple: exactly one recursive call per recursive component (the recursive component may include multiple functions, but there is only one call to the component from outside). Present in some versions of Siemens applications and some versions of `flex`.

Complex: more than one recursive call per recursive component. Present in all versions of `gzip`.

Both: simple and complex recursion in different recursive components of the same program. Present in some versions of `flex` and all versions of `grep`.

Krinke [9] reports increasing benefit from callstack sensitivity when moving from nonrecursive to simple-recursive to complex-recursive programs.

Our findings are less clear. Fig. 5 shows the same data points and ratios from all plots of Fig. 3, here on a logarithmic y-scale and grouped by the types of recursion present in the failing program. Based on this data, it seems that callstack-sensitive slicing is likely to be more beneficial for complex-recursive programs than for simple-recursive programs. This agrees with Krinke's findings. However, there are also cases where callstack-sensitive slicing provides more benefit for nonrecursive programs than for simple-recursive programs, and the largest ratios of all are for programs with both simple and complex recursion. Furthermore, ratios in complex-recursive `gzip` and nonrecursive `space` are fairly similar, while `flex`, whose variants have both simple and complex recursion, has the lowest ratios of all the non-Siemens applications.

As suggested before, the disparity between our results and Krinke's may be due in part to the differing pools of slices considered: We slice only from actual failure points, whereas Krinke's recursion study sliced from all possible call sites (up to a bounded stack depth) as determined by a static callgraph.

3.4 Comparisons of Slice Intersection Sizes and Individual Slice Sizes

In this section, we examine the benefits of intersecting slices when a program that is run on different inputs fails at different points or fails at the same point but with different callstacks. As mentioned in Section 1, intersecting slices only makes sense if all failures are caused by the same bug because only in that case will the slice intersection include the bug. When it is not known a priori that a program has just one bug, previously developed techniques [12], [13], [14] can be used to cluster runs so that failures in each cluster are likely to be due to the same bug. Slice intersections would then be computed only for failures in the same cluster. In our experiments, each buggy version of a program has just one bug, so it is reasonable to consider the benefits of slice intersection.

Intersecting slices is advantageous only if the size of the intersection is smaller than the size of the single smallest slice: If not, the programmer can simply look at the smallest slice. Therefore, we considered the ratios of the size of the smallest slice to the size of the intersection.

For programs that fail at the same point but with different callstacks, there is only one full slice, so we considered only the intersections of the callstack-sensitive slices from that point of failure. For the Siemens suite, slice intersection provided no benefit; in every case, the intersection was identical in size to the smallest individual callstack-sensitive slice. For the non-Siemens programs, on average, the smallest individual callstack-sensitive slice was 1.02 times the size of the intersection of slices. However, in 39 out of

1. Recursion can of course increase the depth of the callstack. Two of the Siemens-suite programs, `print_tokens` and `tot_info`, each have one recursive function; `space` has none; the reference versions of `flex` each have one or two; the reference versions of `grep` each have nine or 10; and the reference versions of `gzip` each have one.

54 cases, this ratio was above 1.05, and the maximum ratio 1.09 arose in 10 cases.

For programs that fail at different points, we compared the size of the intersection of all full slices from those points with the size of the smallest of the full slices, and also the size of the intersection of all callstack-sensitive slices from those points with the size of the smallest of the callstack-sensitive slices. Figs. 6a and 6b show the ratio of smallest single slice size to intersection size for Siemens applications. For the Siemens suite, the smallest full slice was, on average, 1.02 times the size of the intersection of full slices; in the best case, this ratio was 1.10. These ratios were very similar for callstack-sensitive slices; the average and maximum remain 1.02 and 1.10, respectively.

Figs. 6c and 6d show the same ratios (smallest single slice size to intersection size) for the non-Siemens programs. The benefits of intersecting full slices are smaller than those for the Siemens suite: The average ratio of the smallest full slice to intersection of full slices was 1.001 and the maximum ratio was 1.009. Intersecting callstack-sensitive slices for the non-Siemens programs was more beneficial: The average ratio was 1.06, and in the best case was 2.38.

Note that we have assumed that the smallest slice is readily available in our intersection experiments. This is true in a typical in-house testing scenario, with failures derived from a finite test suite under developer control. A developer can run the entire test suite first, then cherry-pick the single smallest slice while ignoring all others. If the entire test suite can be run in a reasonable amount of time, there is no reason not to wait for the one smallest slice.

However, postdeployment debugging is an alternative but equally important scenario. If failures are being collected from the field in an ongoing manner, then continuous intersection of slices as reports arrive may help narrow the search for a bug without having to wait for an ideal (small slice) failure to come along. Thus, in this context, slice intersection can still help developers by providing small slices in a more timely manner.

4 Related Work

This section first discusses how callstack-sensitive slicing fits in the slicing framework defined by Binkley et al. [24], [25], and then considers how it relates to techniques for reducing the sizes of slices based on criteria other than active callstacks.

4.1 Slicing Framework

Harman et al. [26] define a projection theory of program slicing. Binkley et al. [24], [25] use that theory to show the semantic relationships among eight different kinds of slicing. For example, it is shown that dynamic slicing subsumes static slicing because every semantically correct static slice is also a semantically correct dynamic slice.

Three orthogonal dimensions were considered, involving the following constraints:

- *input values*: does the slice preserve execution behavior for all possible inputs or only for a given input?
- *path sensitivity*: do the execution paths in the slice match those in the original program, or is it only the final values of the variables of interest that match?
- *iteration-count sensitivity*: does the slice preserve the values of the variables of interest at all execution instances of the point from which the slice is taken, or only for a particular, given instance?

Callstack-sensitive slicing involves a new kind of constraint on *calling context* and thus provides an interesting new fourth dimension along which to compare different kinds of slicing. The calling-context dimension would be similar to the iteration-count dimension: The latter allows one to specify, for code inside a loop, which iterations are of interest, while the former allows one to specify, for code in a called function, which active calling contexts are of interest.

It is worth noting, however, that the relationships defined by Binkley et al. [24], [25] provide no information about the relative sizes of the slices produced by algorithms that implement the different kinds of slicing. In the proposed new fourth dimension, callstack-sensitive static slicing would subsume "plain" (i.e., full) static slicing, but this says nothing about how much smaller callstack-sensitive slices are in practice. The investigation of that question is the subject of this paper.

4.2 Other Techniques for Reducing Slice Size

4.2.1 Dynamic Slicing

Dynamic slicing was originally defined by Korel and Laski [4]. The idea is to take a slice from a particular *execution instance* of a statement. A slice is specified by providing the statement of interest S, plus additional information to identify the instance of interest, such as the time t at which the statement executed. The dynamic slice from S at time t includes only the statements that actually influenced the execution of that instance of S, rather than the statements that might influence *some* instance of S.

The dynamic slice from an output statement at the time that the first erroneous value was written would, in general, be a smaller slice than the callstack-sensitive slices proposed here. However, dynamic slicing comes with a price: The program must be instrumented to gather an execution trace, and dynamic slicing itself can be very slow. Recent work [27], [28] has addressed the second problem (the time required for computing a dynamic slice), but the authors report program slowdowns of 1-3 orders of magnitude [5]. As discussed in Section 2.4, our tracing tool (the output of which provides the callstacks used to compute callstack-sensitive slices) has comparatively very low overhead, with slowdowns ranging from 1.94 to 2.78 in our experiments. Callstack-sensitive slicing is done using the program's System Dependence Graph (SDG) representation, which requires time polynomial in the size of the program to build [20]. Once the SDG is built, any number of callstack-sensitive slices can be computed. The worst-case time for a single callstack-sensitive slice is linear in the size of the slice times the depth of the callstack.

While future work may succeed in reducing the overhead of dynamic slicing, it is not clear how much better dynamic slices are than callstack-sensitive slices or whether the potential for reduced slice size will outweigh the disadvantage of program slowdown.

4.2.2 Parametric Slicing

Parametric slicing, as defined by Field et al. [29], is related to partial evaluation [30]. Both are static techniques that take into account some input values or some constraints on input

```
1  void print(
2           char *msg,
3           int val) {
4    printf("%s %d\n", msg, val);
5  }
6
7  int getChoice(char* ch) {
8    if (strcmp(ch, "sum") == 0)
9      return 1;
10   else return 0;
11 }
12
13 void main(
14            int argc,
15            char* argv[]) {
16   int result;
17   int k = 1;
18   int ch = getChoice(argv[1]);
19   if (ch == 0) {
20     result = 0;
21     while (k<11) {
22       result += k;
23       k++;
24     }
25   } else {
26     result = 1;
27     while (k<11) {
28       result *= k;
29       k++;
30     }
31   }
32   print("val: ", result);
33 }
```

Fig. 7. Example to illustrate a case where parametric slicing produces a smaller slice than callstack-sensitive slicing.

values (e.g., the value read into x is greater than zero). Field et al. [29] define parametric slicing as a framework, where slicing is carried out using a set of rewriting rules. The size of a parametric slice depends both on the particular rewriting rules that are used (with the usual trade-offs between precision and efficiency) and the particular constraints provided. For example, consider slicing the program in Fig. 1b. Given a reasonable set of rewriting rules and the constraint that the input value (the command line argument) is sum, the parametric slice from line 4 is the same as the callstack-sensitive slice from line 4 with an active call at line 31. However, the parametric slice from line 4 is worse (larger) than the callstack-sensitive slice if the constraint is that the input value is either sum or prod; in this case, the parametric slice is the same as the full slice. Finally, to see an example where a parametric slice is better than a callstack-sensitive one, consider the code in Fig. 7. This code is similar to the code in Fig. 1b, except that the sum and product are computed in variable result rather than in separate variables sum and prod, and there is just one call to function print to print the final value of result. Because there is only one call to print, the callstack-sensitive slice from line 4 (with an active call at line 32) is the same as the full slice; in particular, it includes lines 20-23. In contrast, the parametric slice from line 4 with the constraint that the input value is sum excludes those four lines.

4.2.3 *Conditioned Slicing*

Conditioned slicing [31], [32], [33] is very similar to parametric slicing in that the programmer is able to specify constraints on input values. The examples discussed above for which parametric slicing is better, worse, and the same as callstack-sensitive slicing apply to conditioned slicing as well. Fox et al. [33] discuss an extension to conditioned slicing that allows the specification of constraints (e.g., $x = y$) that must hold at the statement from which a slice is taken, but the details of the approach are not clear, and there is no discussion of an implementation or of experimental results.

4.2.4 *Hybrid Slicing*

Hybrid slicing [34] is a dynamic technique, used in conjunction with a debugger that allows the programmer to set break points. The system keeps track of the sequence of break points encountered as the program is run as well as the sequence of function calls and returns that are executed. When a slice is requested, the sequence of break points that have and have not been encountered plus the sequence of function calls/returns that have executed are used to compute a safe approximation to the paths that might have been taken (and those that could not have been taken). This approximation is used to exclude some program components from the slice.

If a complete history is maintained, a hybrid slice will always be at least as small as the corresponding callstack-sensitive slice. However, as suggested by Gupta et al. [34], the cost of hybrid slicing may be unacceptable if complete break point and call/return histories are maintained. Therefore, it is proposed that histories should be truncated to some predefined length. Clearly, when histories are truncated, it is possible to construct examples for which a hybrid slice is arbitrarily worse than the corresponding callstack-sensitive slice.

The experiments reported by Gupta et al. [34] are quite limited. They used very short programs (no more than 688 lines of code) and gave no timing data. Thus, it is difficult to predict how much history truncation will be required or how hybrid slice sizes will compare to the corresponding callstack-sensitive slice sizes in practice.

4.2.5 *Call-Mark Slicing*

Call-mark slicing, defined by Nishimatsu et al. [35], involves tracking of statements that execute at least once when a program is run. That information is used to reduce slice sizes by halting the backward traversal of the System Dependence Graph when a nonexecuted component is reached. The authors point out that it is not necessary to track every statement in the program: If statement $S1$ dominates statement $S2$ in the program's control-flow graph or if both statements occur in the same basic block, then (for a terminating program) if $S1$ does not execute, neither does $S2$. The technique is called *call-mark slicing* because they suggest limiting runtime tracking to the call statements in the program.

Fig. 8 contains a program to illustrate that for different examples, call-mark slicing can produce slices that are

```
1  void print(
2           char *msg,
3           int val) {
4    printf("%s %d\n", msg, val);
5  }
6
8  int dec( int val ) {
9    return val-1;
10 }
11
12 void main(
13           int argc,
14           char* argv[]) {
15   int k;
16   for (k=1; k<argc; k++) {
17     int sum = 0;
18     int prod = 0;
19     int N = atoi(argv[k]);
20     if (N < 20) {
21       while (N > 1) {
22         prod *= N;
23         N = dec(N);
24       }
25       print("prod", prod);
26     } else {
27       while (N > 1) {
28         sum += N;
29         N = dec(N);
30       }
31       print("sum", sum);
32     }
33   }
34 }
```

Fig. 8. Example to illustrate cases where call-mark slicing produces slices that are smaller than, the same size as, and larger than those produced by callstack-sensitive slicing.

smaller than, the same size as, or larger than those produced by callstack-sensitive slicing. The new example program computes and prints either the sum or the product of the numbers from 1 to N, for each value N supplied as a command line argument. If N is less than 20, the product is computed; otherwise, the sum is computed.

If the example program is run with the input `10`, line 4 (the call to `printf` in function `print`) is executed once, with an active call at line 25. The full slice from line 4 is the whole program. The callstack-sensitive slice and the call-mark slice are identical: Both omit lines 27-31.

If the example program is run with the input `10 100`, line 4 is executed twice: first with an active call at line 25 and then with an active call at line 31. In this case, the full and call-mark slices are identical: Both consist of the whole program. This is true whether only call statements or all statements are tracked because in this case, all statements in the program are executed. The callstack-sensitive slice with the first callstack omits lines 27-31, and with the second callstack, it omits lines 21-25.

If the example program is run with the input `1`, line 4 is executed once with an active call at line 25. The full slice is the whole program, the callstack-sensitive slice omits lines 27-31, and the call-mark slice additionally omits lines 22 and 23.

4.2.6 *Set Operations on Slices*

Ours is not the first work to contemplate forming intersections of slices. Gallagher and Lyle [36] suggest using slice intersections for software quality assurance auditing of safety-critical code. Intersections of backward slices identify possible interactions between critical components, with an empty intersection revealing two components to be mutually independent.

Agrawal et al. [37] propose a debugging paradigm based on dynamic slicing and backtracking. Their prototype implementation, SPYDER, allows user-directed slice differencing and intersection to "give the user the ability to obtain dynamic program dices (and more)." The particular role of intersections in debugging is not explored further, and no empirical evaluation is offered.

While slices shrink under intersection and differencing, operations that make slices larger can also be useful. Mulhern and Liblit [38] describe debugging strategies based on unions of slices. They show that many algorithms give slices whose unions do not behave as expected and offer a novel slicing algorithm that is both precise and well behaved under union.

5 Conclusions

We have described techniques that have the potential to make a significant difference in how, and how effectively, programmers debug their code. Our output trace collection and analysis tools allow a programmer to find point-of-failure information for programs that produce bad output. Given a point of failure identified by those tools or by a crash, callstack-sensitive slicing helps a programmer find the problem more quickly by reducing the size of the backward slice from that point.

Experimental evaluation of our proposed approach is very promising: The overhead of the trace and mapping tools is minimal, and callstack-sensitive slicing can dramatically decrease slice sizes.

We have also investigated the use of slice intersection for debugging programs that fail in multiple ways. While the improvements gained this way are more modest than those gained via the use of callstack-sensitive slices, they are still likely to be worthwhile in many cases in practice.

Acknowledgments

This work was supported in part by AFOSR grant FA9550-07-1-0210; LLNL contract B580360; and the US National Science Foundation (NSF) grants CCF-0621487, CCF-0701957, CCR-0305387, and CNS-0720565. Any opinions, findings, and conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or other institutions. M. Polishchuk was with the University of Wisconsin–Madison when this work was performed.

References

[1] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352-357, July 1984.

[2] *Wisconsin Program-Slicing Tool 1.1 Reference Manual,* Wisconsin Alumni Research Foundation, http://www.cs.wisc.edu/wpis/slicing_tool/slicing-manual.ps, Nov. 2000.

[3] D. Binkley and M. Harman, "A Large-Scale Empirical Study of Forward and Backward Static Slice Size and Context Sensitivity," *Proc. 2003 Int'l Conf. Software Maintenance,* Sept. 2003.

[4] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters,* vol. 29, no. 3, pp. 155-163, 1988.

[5] R. Gupta, personal communication, 2006.

[6] D. Binkley, "Semantics Guided Regression Test Cost Reduction," *IEEE Trans. Software Eng.,* vol. 23, no. 8, pp. 498-516, Aug. 1997.

[7] J. Krinke, "Context-Sensitivity Matters, but Context Does Not," *Proc. Int'l Workshop Source Code Analysis and Manipulation,* pp. 29-35, 2004.

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Trans. Programmings Languages and Systems,* vol. 12, no. 1, pp. 26-60, Jan. 1990.

[9] J. Krinke, "Effects of Context on Program Slicing," *J. Systems and Software,* vol. 79, no. 9, pp. 1249-1260, 2006.

[10] G.R.H. Do and S. Elbaum, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact," *Empirical Software Eng.: An Int'l J.,* vol. 10, no. 4, pp. 405-435, 2005.

[11] G. Rothermel, S. Elbaum, A. Kinneer, and H. Do, "Software-Artifact Infrastructure Repository," http://sir.unl.edu/portal/, Sept. 2006.

[12] P. Francis, D. Leon, M. Minch, and A. Podgurski, "Tree-Based Methods for Classifying Software Failures," *Proc. 15th Int'l Symp. Software Reliability Eng.,* pp. 451-462, 2004.

[13] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying Classification Techniques to Remotely-Collected Program Execution Data," *Proc. 10th European Software Eng. Conf.,* pp. 146-155, 2005.

[14] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical Debugging: Simultaneous Identification of Multiple Bugs," *Proc. 23rd Int'l Conf. Machine Learning,* pp. 1105-1112, June 2006.

[15] *The GNU C Library,* 0th ed., The Free Software Foundation, July 2001.

[16] *GNU Binutils,* binutils 2.17 ed., The Free Software Foundation, June 2006.

[17] R.M. Stallman andthe GCC Developer Community, *Using the GNU Compiler Collection (GCC),* gcc 4.1.1 ed., The Free Software Foundation, May 2006.

[18] D. Veillard, *The XML C Parser and Toolkit of Gnome,* http://xmlsoft.org/, Sept. 2006.

[19] GrammaTech, *Codesurfer,* http://www.codesurfer.com, Sept. 2006.

[20] T. Reps, S. Horwitz, and G. Rosay, "Speeding up Slicing," *Proc. ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 11-20, Dec. 1994.

[21] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. Int'l Conf. Software Eng.,* pp. 191-200, May 1994.

[22] F.I. Vokolos and P.G. Frankl, "Empirical Evaluation of the Textual Differencing Regression Testing Technique," *Proc. Int'l Conf. Software Maintenance,* pp. 44-53, Nov. 1998.

[23] W.E. Wong, J.R. Horgan, A. Mathur, and A. Pasquini, "Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application," *Proc. 21st Int'l Computer Software and Applications Conf.,* pp. 522-528, Aug. 1997.

[24] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and B. Korel, "Theoretical Foundations of Dynamic Program Slicing," *Theoretical Computer Science,* vol. 360, no. 1, pp. 23-41, 2006.

[25] D. Binkley, S. Danicic, T. Gyimothy, M. Harman, A. Kiss, and B. Korel, "A Formalisation of the Relationship between Forms of Program Slicing," *Science of Computer Programming,* vol. 62, no. 3, pp. 228-252, 2006.

[26] M. Harman, D. Binkley, and S. Danicic, "Amorphous Program Slicing," *J. Systems and Software,* vol. 68, no. 1, pp. 45-64, 2003.

[27] X. Zhang and R. Gupta, "Cost Effective Dynamic Program Slicing," *Proc. ACM SIGPLAN 2004 Conf. Programming Language Design and Implementation,* pp. 94-106, June 2004.

[28] X. Zhang, R. Gupta, and Y. Zhang, "Cost and Precision Tradeoffs of Dynamic Data Slicing Algorithms," *ACM Trans. Programming Languages and Systems,* vol. 27, no. 4, pp. 631-661, July 2005.

[29] J. Field, G. Ramalingam, and F. Tip, "Parametric Program Slicing," *Proc. ACM Symp. Principles of Programming Languages,* pp. 379-392, Jan. 1995.

[30] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation.* Prentice Hall Int'l, 1993.

[31] S. Danicic, C. Fox, M. Harman, and R. Hierons, "ConSIT: A Conditioned Program Slicer," *Proc. Ninth IEEE Working Conf. Reverse Eng.,* pp. 216-226, Oct. 2000.

[32] D. Daoudi, S. Danicic, J. Howroyd, M. Harman, C. Fox, and M. Ward, "ConSUS: A Scalable Approach to Conditioned Slicing," *Proc. Ninth IEEE Working Conf. Reverse Eng.,* pp. 109-118, Oct. 2002.

[33] C. Fox, M. Harman, R. Hierons, and S. Danicic, "Backward Conditioning: A New Program Specialisation Technique and Its Application to Program Comprehension," *Proc. Ninth Int'l Workshop Program Comprehension,* pp. 89-97, May 2001.

[34] R. Gupta, M. Soffa, and J. Howard, "Hybrid Slicing: Integrating Dynamic Information with Static Analysis," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 4, pp. 370-397, Oct. 1997.

[35] A. Nishimatsu, M. Jihira, S. Kusumoto, and K. Inoue, "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice," *Proc. 21st Int'l Conf. Software Eng.,* pp. 422-431, 1999.

[36] K.B. Gallagher and J.R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Trans. Software Eng.,* vol. 17, no. 8, pp. 751-761, Aug. 1991.

[37] H. Agrawal, R.A. DeMillo, and E.H. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software—Practice & Experience,* vol. 23, no. 6, pp. 589-616, 1993.

[38] A. Mulhern and B. Liblit, "Effective Slicing: A Generalization of Full and Relevant Slicing," Technical Report 1639, Univ. of Wisconsin-Madison, June 2008.

Susan Horwitz received the PhD degree in computer science from Cornell University in 1985. She has been on the faculty of the Department of Computer Sciences at the University of Wisconsin–Madison since September 1985, serving as an associate chair from 2004 to 2007. Her research has mainly focused on the design and implementation of language-based programming tools, including the design of algorithms for program slicing, on-demand data-flow analysis, and the analysis of programs with pointers.

Ben Liblit received the PhD degree from the University of California Berkeley with advisor Alex Aiken in 2004. He is an assistant professor in the Department of Computer Sciences at the University of Wisconsin–Madison. His research combines static, dynamic, and statistical methods to create debugging tools that cope with the ugly complexities of real-world software development. He received the 2005 ACM Doctoral Dissertation Award for his work on postdeployment statistical debugging.

Marina Polishchuk received the MS degree in computer science from the University of Wisconsin–Madison in 2006. She has been a software development engineer in test at Microsoft since 2006. She currently works to advance testing at Microsoft as a whole in Microsoft's Engineering Excellence Team, where she trains, mentors, and collaborates with testers to establish the best practices. Her primary interests are debugging, program understanding, automated test generation, and compiler testing.