SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2007; **37**:935–961
Published online 19 December 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.795

Locating faulty code by multiple points slicing

Xiangyu Zhang, Neelam Gupta and Rajiv Gupta*,†

Department of Computer Science, The University of Arizona, Tucson, AZ 85737, U.S.A.

SUMMARY

Dynamic slicing has long been considered as a useful tool for debugging programs as it effectively identifies a reduced fault candidate set which captures the faulty code in the program. Traditionally, a backward dynamic slice is computed starting from an incorrect value observed by the programmer during a failed program run. This incorrect value is either an incorrect output value or an incorrect address whose dereferencing causes the program to crash. Recently we proposed two additional types of dynamic slices, a forward dynamic slice of a minimal failure inducing input difference and a bidirectional dynamic slice of a critical predicate. We have built a dynamic slicing tool that computes dynamic slices by instrumenting program binaries and executing them to build dynamic dependence graphs. In this paper, through experiments, we demonstrate that supporting three different types of dynamic slices has the following advantages. First, we observe that for each type of dynamic slice there are distinct situations in which it is not applicable. Therefore, we should support multiple types of slices to handle a wide range of situations. Second, supporting multiple types of dynamic slices enables us to compute a multiple points dynamic slice which is the intersection of different type of available slices. Our experiments show that multiple points dynamic slices are significantly smaller than any of the three kinds of individual dynamic slices. Copyright © 2006 John Wiley & Sons, Ltd.

Received 13 February 2006; Revised 2 October 2006; Accepted 6 October 2006

KEY WORDS: fault location; dynamic dependence graph; dynamic backward slice; forward dynamic slice; bidirectional dynamic slice; multiple points dynamic slice

1. INTRODUCTION

The concept of program slicing was first introduced by Mark Weiser [1]. He introduced program slicing as a debugging aid and gave the first *static slicing* algorithm. The static slice of a reference to a variable

*Correspondence to: Rajiv Gupta, Department of Computer Science, The University of Arizona, 1040 East 4th Street, Gould Simpson Building, Room 746, Tucson, AZ 85721, U.S.A.
†E-mail: gupta@cs.arizona.edu

Contract/grant sponsor: Microsoft
Contract/grant sponsor: IBM
Contract/grant sponsor: NSF; contract/grant numbers: CNS-0614707, CCF-0541382, CCF-0324969 and EIA-0080123

at a program point is the set of program statements that *can influence* the value of the variable at the given program point under some execution. Therefore every statement in the program from which there is a chain of static data and/or control dependences leading to the variable reference belongs to the static slice of the variable reference. Let us consider a program containing faulty statements. Given a program point at which the value of a variable is output, if the static slice of this variable reference contains one or more faulty statements then under some executions incorrect results may be produced at the output statement. By studying the static slice of the output, a programmer may be able to detect the faulty statements. However, since static slices can be large due to the use of conservative dependence analysis, the effort required to locate the faulty statement is expected to be large.

To improve the effectiveness of slicing in program debugging, Korel and Laski proposed the idea of *dynamic slicing* [2]. The *backward dynamic slice* (BwS) of an incorrect value (either an incorrect output value or an incorrect address whose dereferencing caused a program crash) at an execution point includes all those executed statements which *actually effected* the value of the variable at that point. The BwS includes a subset of executed statements that frequently captures the faulty code in it. In the absence of dynamic slicing the programmer may have to examine all executed statements in search of faulty code, whereas in the presence of dynamic slicing the programmer needs to examine a smaller fault candidate set (FCS), which is the BwS. Results of our study of dynamic slices reported in [3] show that the number of distinct statements executed at least once during a program execution were 2.46–56.08 times more than the number of statements in a dynamic slice.

We recently proposed two new types of dynamic slices [4,5]. The first new type is the *forward dynamic slice* (FwS) of the *minimal failure inducing input difference*. Given an input on which the program fails, the *minimal failure inducing input difference* is the part of the input that is found to cause the failure. As a result, the forward slice starting from the failure inducing input produces yet another FCS that captures the faulty code. The second new type is the *bidirectional dynamic slice* (BiS) of a *critical predicate*. Given an input on which the program fails, the *critical predicate* is an execution instance of a conditional branch predicate such that if the outcome of the predicate's execution instance is reversed, the program terminates producing the correct output. Since the predicate outcome is related to the fault, we find that the BiS that includes both the BwSs and FwSs of the predicate execution instance produces yet another reduced FCS. Finally, now that we have three different FCSs corresponding to the BwS, FwS, and BiS, we can produce an even smaller FCS by intersecting two or more FCSs corresponding to the three different types of slices. We refer to such resulting slices as *multiple points dynamic slices* (MPSs) as they are the result of computing dynamic slices starting from multiple points (erroneous value, failure inducing input, and critical predicate). We have implemented a dynamic slicing tool that works on Intel x86 binaries and implements the above techniques. The key contribution of this paper is an experimental study carried out using the above tool. From the results of this study we make the following observations.

- Although, in general, it cannot be guaranteed that the different types of dynamic slices considered will capture the faulty code, in practice we observe that they are very likely to do so. In fact, for the set of faults considered in our experiments, faulty code was always captured by dynamic slices.
- We observe that for each type of dynamic slice there are distinct situations in which it is not applicable. Therefore we must support multiple types of slices in order to handle a wide range of situations.

- We also found that the FwS and BiS, the two new types of dynamic slices that we recently introduced, were significantly smaller than BwSs in many cases.
- Supporting multiple types of dynamic slices enables us to compute MPSs that are significantly smaller than all three kinds of individual dynamic slices in many cases.

The rest of the paper is organized as follows. In Section 2 we describe the different types of dynamic slices including BwS, FwS, BiS, and MPS. We illustrate the benefits of dynamic slices through examples and describe their limitations. In Section 3 we give an overview of our slicing tool for different types of slice. Section 4 presents the results of our experiments, analysis of data collected, and the conclusions that can be drawn from this study. Related work is presented in Section 5 and our contributions are summarized in Section 6.

2. DYNAMIC SLICING ALGORITHMS

The scenario for fault location considered in this paper is as follows. Given a failing run of a program, our goal is to identify a FCS, i.e. a small subset of program statements that includes the faulty code whose execution caused the program to fail. Thus, we assume that the fact that the program has failed is known either because the program crashed or because it has produced an output that the user has determined to be incorrect. Moreover, this failure is the result of the execution of faulty code and not the result of other reasons (e.g. faulty environment variable setting). The statements executed during a failing run can constitute a first conservative approximation of the FCS. However, since the user has to examine the FCS manually to locate faulty code, smaller FCSs are desirable. The goal of dynamic slicing is to produce a smaller FCS than the set that includes all executed statements.

In this section we consider three different kinds of dynamic slices collectively referred to as single point dynamic slices because the construction of the dynamic slice is started from a value that is known to be related to the cause of the program failure (i.e. the faulty code executed in the failing run). We will also discuss how MPS can be used to further reduce the sizes of FCSs. It is important to note that dynamic slices are not in general guaranteed to capture the faulty code. We will briefly indicate the reasons for this as we describe the various kinds of dynamic slices (a more detailed discussion of this topic can be found in [6]). However, as our experimental study shows, in practice the dynamic slices considered in this paper are very effective in capturing faulty code.

The various types of dynamic slices considered are computed from the dynamic dependence graph of the program execution. Therefore, before we discuss dynamic slices, let us briefly discuss how the dynamic dependence graph is constructed. Given a program run, in its dynamic dependence graph, each node represents an execution instance of a statement and edges correspond to dynamic data dependences (i.e. flow of data values from a node that computes a value to a node that uses that value) or dynamic control dependences (i.e. an indication that the execution of a node was performed directly as a result of a particular outcome of a conditional branch). Let us briefly discuss how dynamic dependences are identified so that the dynamic dependence graph needed during dynamic slicing can be constructed. To identify dynamic data dependences, for each memory address the execution instance of an instruction that was responsible for the latest write to the address is remembered. When an execution instance of an instruction uses the value at an address, a dynamic data dependence is established between the execution instance of the instruction that performed the latest write to the address and the execution instance of the instruction that used the value at the address. Dynamic control dependences are also identified. An execution instance of an instruction is dynamically control-dependent upon the execution instance of the predicate that caused the execution of the instruction. By first computing the static control predecessors of an instruction, and then detecting which one of these was the last to execute prior to a given execution of the instruction, dynamic control dependences are computed.

2.1. Single point dynamic slices

To construct a FCS, we need to identify a slicing criterion, i.e. a value in the dynamic dependence graph that is related to the execution of the faulty code. Once the slicing criterion is known, we traverse the dynamic dependence graph to identify the set of statements that are related to this value through chains of dynamic dependences. This set of statements forms a possible FCS. Since there is a *single point* at which the construction of the dynamic slice originates, we refer to such a dynamic slice as a single point dynamic slice.

2.1.1. BwS of an erroneous computed value

In prior work, BwSs have been used for debugging [2,7–9]. Consider a failing run which produces an incorrect output value or crashes owing to dereferencing of an illegal memory address. The incorrect output value or the illegal address value is now known to be related to faulty code executed during this failed run. It should be noted that the identification of an incorrect output value will require help from the user unless the correct output for the test input being considered is already available to us. The FCS is constructed by computing the BwS starting at the incorrect output value or illegal address value as shown in Figure 1. The BwS of a value at a point in the execution includes all those executed statements that effect the computation of that value. In other words, statements that directly or indirectly influence the computation of faulty value through chains of *dynamic data and/or control dependences* are included in the BwSs. Thus, the backward reachable subgraph forms the BwS and all statements that appear at least once in the reachable subgraph are contained in the backward slice. During debugging, both the statements in the slice and the dependence edges that connect them provide useful clues to the failure cause.

We illustrate the benefit of backward slicing by using a bug in *bc-1.06* as an example that causes a heap overflow error. In this program, a sufficiently large heap buffer is not allocated which causes an overflow. The code corresponding to the error is shown in Figure 2. The heap array *arrays* allocated at line number 167 overflows at line 177 causing the program to crash. Therefore, the dynamic slice is computed starting at the address of *arrays*[*indx*] that causes the segmentation fault. Since the computation of the address involves *arrays*[] and *indx*, both statements at lines 167 and 176 are included in the dynamic slice. By examining statements at lines 167 and 176, the cause of the failure becomes evident to the programmer. It is easy to see that although *a_count* entries have been allocated at line 167, *v_count* entries are accessed according to the loop bounds of the *for* statement at line 176. This is the cause of the heap overflow at line 177.

While in the above example the BwS clearly captures the faulty code, this is not true in general. In [10] a situation in which faulty code can be missed was first described. If the faulty code causes a predicate to evaluate incorrectly such that the execution of a statement that should have influenced the output is bypassed, then the predicate and hence the faulty code may not be included in the dynamic slice.

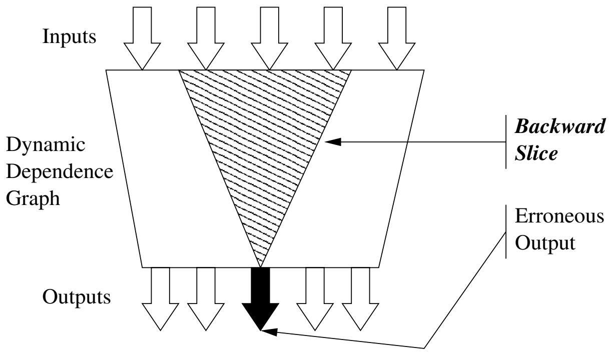
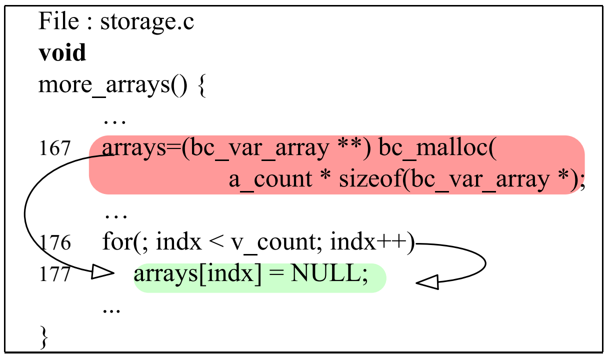


Figure 1. BwS.



```
File : storage.c
void
more_arrays() {
      …
      arrays=(bc_var_array **) bc_malloc(
                    a_count * sizeof(bc_var_array *));
      …
      for(; indx < v_count; indx++)
         arrays[indx] = NULL;
      ...
}
```

Figure 2. A heap overflow bug in *bc-1.06*.

Next we will describe two additional kinds of single point dynamic slices. We have recently discovered the potential of these two types of slices as a possible means for constructing FCSs. Interestingly, as we will see subsequently, while the above dynamic slice is a *backward slice*, the two new kinds of slices are constructed by *forward* and *bidirectional* traversals of the dynamic dependence graph and are thus called *forward dynamic slice* and *bidirectional dynamic slice*.

2.1.2. FwS of failure-inducing input difference

Zeller introduced the term delta debugging [11] for the process of determining the causes for program behavior by looking at the differences (the deltas) between the old and new configurations of the programs. Hildebrandt and Zeller [12,13] then applied the delta debugging approach to simplify and

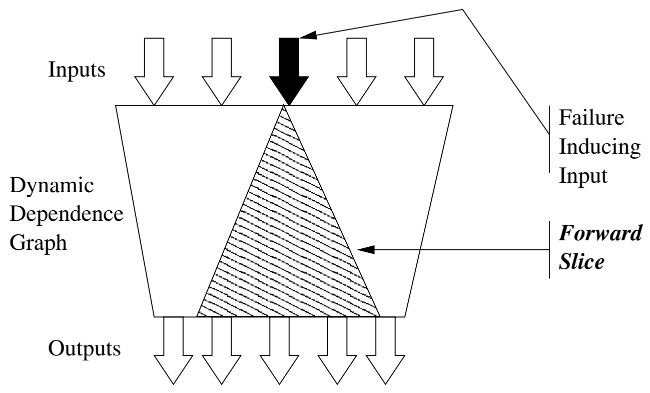


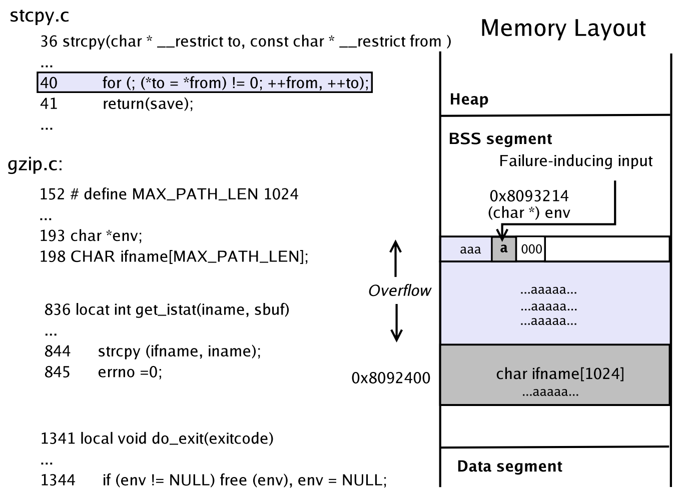
Figure 3. FwS.

isolate the failure-inducing input difference. The basic idea of delta debugging is as follows. Given two program runs r_s and r_f corresponding to the inputs I_s and I_f, respectively, such that the program fails in run r_f and completes execution successfully in run r_s, the delta debugging algorithm can be used to systematically produce a pair of inputs I'_s and I'_f with minimal difference such that program fails for I'_f and executes successfully for I'_s. The difference between these two inputs isolates the failure-inducing difference part of the input. These inputs are such that their values play a critical role in distinguishing a successful run from a failing run.

Since the minimal failure-inducing input difference leads to the execution of faulty code and hence causes the program to fail, we can identify a FCS by computing a FwS starting at this input as shown in Figure 3. In other words, all statements that are influenced by the input value directly or indirectly through a chain of data and/or control dependences are included in the FCS. Thus, now we have a means for producing a new type of dynamic slice that also represents a FCS. We recognized the role of FwSs in fault location for the first time in [4].

There is an additional cost to using the above technique. First, the user must provide a parser for the input such that the input can be separated into meaningful entities and new inputs generated from it. Second, the user must examine the outputs for the additional program runs corresponding to the generated inputs. Finally, we also note that the FwS is not guaranteed to contain the faulty code [6].

We applied the above approach to a well-known buffer overflow problem in *gzip-1.0.7*. Figure 4 illustrates the details of the problem. On the left-hand side of Figure 4, we show the relevant code segment for the problem. The problem occurs in the *strcpy* statement at line 844. Variable *ifname* is a global array defined at line 198. The size of the array is defined as 1024. Before the *strcpy* statement at line 844, there is no check on the length of string *iname*. If the length of string *iname* is longer than 1024, the buffer overflows and the program crashes. The memory layout of the *gzip* program is shown on the right-hand side of Figure 4. We can see from the figure that there is a global pointer *env* located in an address space above array *ifname*. The difference between *env* and *ifname* is 3604 bytes. If the length of string *iname* is larger than 3604, the value of *env* will be changed as a result of buffer overflow. When we look at function *do_exit* at line 1341, before the program quits, it tries to free the



```
stcpy.c
36 strcpy(char * __restrict to, const char * __restrict from )
...
40        for (; (*to = *from) != 0; ++from, ++to);
41        return(save);
...

gzip.c:
152 # define MAX_PATH_LEN 1024
...
193 char *env;
198 CHAR ifname[MAX_PATH_LEN];

836 locat int get_istat(iname, sbuf)
...
844     strcpy (ifname, iname);
845     errno =0;

1341 local void do_exit(exitcode)
...
1344      if (env != NULL) free (env), env = NULL;
```

Figure 4. Buffer overflow bug in *gzip*.

memory pointed to by *env*. If the value of *env* is an illegal memory address owing to buffer overflow, it causes a segmentation fault at line 1344.

To test the *gzip* program, we picked two inputs: the first input is a file name '*aaaaa*', which is a successful input, and the second input is a file name '*a < repeated 3610 times >*', which is a failure input because the length is larger than 3604. After applying *sddmin* algorithm [13] on them, we have two new inputs: the new successful input is a file name '*a < repeated 3604 times >*' and the new failed input is a file name '*a < repeated 3605 times >*'. The failure-inducing input difference between them is the last character '*a*' in the new failed input.

We used slicing to compute the forward slice of the failure-inducing input difference in the failed input. The size of the forward slice is 3 which includes the `for` statement at line 40 in *strcpy.c*. This is of course the place where the buffer overflow occurred. Our slicing implementations run on the binary code level and thus are able to check the memory space of a program and even check the code in the library.

2.1.3. BiS of a critical predicate

Given an erroneous run of the program, the objective of this method is to explicitly force the control flow of the program along alternate path at a critical branch predicate such that the program produces the correct output. The basic idea of this approach is inspired from the following observation. Given an input on which the execution of a program fails, a common approach to debugging is to run the program on this input again, interrupt the execution at certain points to make changes to the program state, and then see the impact of changes on continued execution. If we can discover the changes to the program state that cause the program to terminate correctly, we obtain a good idea of the error that otherwise

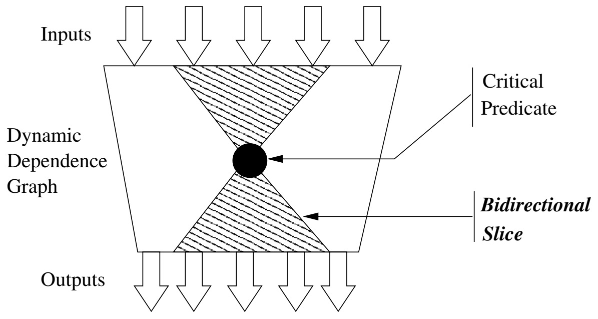


Figure 5. BiS.

was causing the program to fail. However, automating the search of state changes is prohibitively expensive and difficult to realize because the search space of potential state changes is extremely large (e.g. even possible changes for the value of a single variable are enormous if the type of the variable is integer or floating). On the other hand, changing the outcomes of predicate instances greatly reduces the state search space as a branch predicate has only two possible outcomes, true or false. Therefore, we note that through forced *switching* of the outcomes of some predicate instances at runtime, it may be possible to cause the program to produce correct results.

Having identified a critical predicate instance, we compute a FCS as the BiS of the critical predicate instance as shown in Figure 5. This bidirectional slice is essentially the union of the BwS and the FwS of the critical predicate instance. Intuitively, the reason why the slice must include both the BwS and FwS is as follows. Consider the situation in which the effect of executing faulty code is to cause the predicate to evaluate incorrectly. In this case, the BwS of the critical predicate instance will capture the faulty code. On the other hand, it is possible that by changing the outcome of the critical predicate instance we avoid the execution of faulty code and hence the program terminates normally. In this case, the forward slice of the critical predicate instance will capture the faulty code. Therefore, the faulty code will either be in the backward slice or the forward slice. We recognized the role of BiSs in fault location for the first time in [5].

Next we briefly describe how a critical predicate instance is found. To search for a critical predicate instance, the program is repeatedly re-executed and during each re-execution the outcome of a single predicate instance is forcibly changed by our dynamic-instrumentation-based slicing tool. Since, during the program run, a large number of predicate instances are present, a strategy is needed to order predicate instances such that they are tested for being a critical predicate instance in that order. We have identified a simple ordering heuristic that, in practice, rapidly locates a critical predicate instance. This heuristic orders the predicate instances that belong to the BwS of the erroneous output using their dependence distance from the erroneous output. In other words, predicate instances, in the BwS of the erroneous output, that are linked by a shorter chain of dynamic dependence edges to the erroneous output are examined before the predicate instances that are linked by a longer chain of dynamic dependences to the erroneous output. In general, there may be many critical predicate instances that suffice for computing a bidirectional slice. Our algorithm, however, finds the first critical predicate

```
970     base = …
        . . .
2565    base[…] = ...
        . . .
2667    for ( i = 0; i <= lastdfa; ++i )
2668          {
              . . .
2673          int offset = base[i+1];
              . . .
2677          chk[offset] = EOB_POSITION;
              . . .
2681          chk[offset - 1] = ACTION_POSITION;
              . . .
2683          }
2684
2685    for ( i = 0; i <= tblend; ++i )
2686          {
              . . .
2690          else if ( chk[i] == ACTION_POSITION )
                    printf("%7d, %5d,", 0 , …);
              . . .
2696          else    /* verify, transition */
                    printf("%7d, %5d," , chk[i], …);
              . . .
2699          }
```

Figure 6. Incorrect output bug in *flex*.

instance and then uses it to compute the BiS. While multiple critical predicate instances can be found in some cases, the search for these additional critical predicates will incur additional execution time cost.

It should be noted that, as stated earlier, dynamic slices are not guaranteed to capture the faulty code. In addition, this techniques has the limitation that a critical predicate may not exist. For example, if the fault is sufficiently complex, changing the outcome of a single predicate instance may not be able to produce the correct output.

We illustrate the notion of critical predicate with the faulty version of the *flex* (a fast lexical analyzer generator) program shown in Figure 6 which is taken from the Siemens suite [14,15]. The Siemens suite provides the associated test suites and faulty version for each program. The program in Figure 6 is derived from *flex-2.4.7* and augmented by the provider of the program with a bug that is circled in the figure: $base[i+1]$ should actually be $base[i]$. We took the first provided input that produced an erroneous output. We observed that the output is different from the expected output for the 538th character, a '1' is produced as output owing to the execution of *printf* in the *else* part (line 2696) of the *else if* statement at line 2690 instead of a '0' that should be output by execution of the *printf* in the *then* part of the *else if* statement. Under the correct execution at line 2673 *offset* would have been assigned the value of $base[0]$ which is 1. The variable $chk[0]$ at line 2681 would have been assigned *ACTION_POSITION* causing the predicate at line 2690 to evaluate to *true* for the loop iteration corresponding to $i = 0$. As a result of the error at line 2673, an incorrect value of $offset(= 3)$ causes $ch[0]$ to have an incorrect stale value $(= 1)$ which causes the predicate at line 2690 to incorrectly evaluate to its *false* outcome. Using our proposed method we looked for a predicate instance whose switching corrected the output. We found the appropriate instance of the *else if* predicate instance through this search. Once this predicate instance was found we could easily determine by following

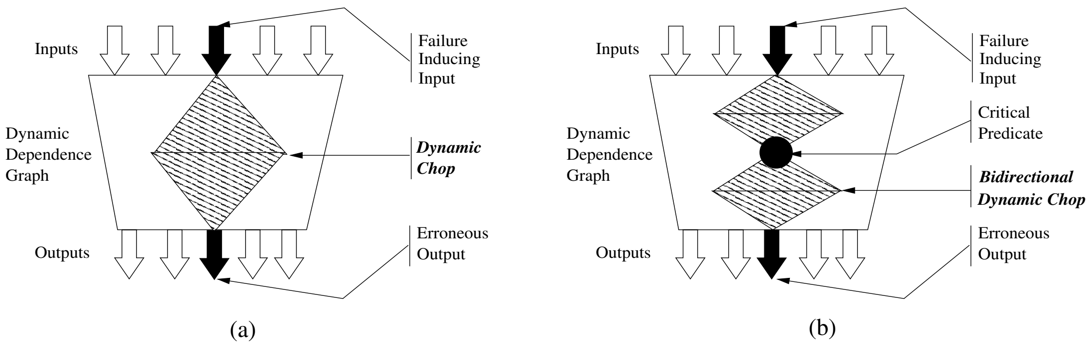


Figure 7. MPSs: (a) dynamic chop; and (b) bidirectional dynamic chop.

the data dependences backwards that the incorrect value of *ch*[*0*] was a stale value and that it did not come from the most recent execution of the for loop at line 2667. Thus, now it was clear that the error was in the statement at line 2673 which sets the *offset* value.

In the above example we demonstrated that by enforcing the outcome of a predicate we avoided searching for potential modifications of values for *chk*[ ], *offset*, or *base*[ ] which are *integer* variables and thus have a wide possible range of values. The above example also illustrates that it is important to alter the outcome of selected predicate instances as opposed to all execution instances of a given predicate. This is because the fault need not be in the predicate but elsewhere and thus all evaluations of the predicate need not be incorrect.

2.2. MPSs: dynamic chops

We have described three different ways of computing FCSs in the preceding section. Each FCS captures the faulty code. Therefore, it follows that if more than one kind of dynamic slice is available, we can further reduce the size of the FCS by intersecting the single point dynamic slices. In fact the idea of intersecting dynamic slices to obtain smaller slices has been explored by researchers in the development of dynamic chopping. First, Figure 7(a) shows that by intersecting the FwS and BwS we can obtain a *dynamic chop*. The dynamic chop captures the faulty code and is smaller than both the FwS and BwS. Next, if the BiS is available, it can be intersected with the dynamic chop to further reduce the size of the FCS. As shown in Figure 7(b) the result of this operation is a pair of dynamic chops, which we refer to as a *bidirectional dynamic chop*, one between the failure inducing input difference and a critical predicate and the other between a critical predicate and the erroneous output value. As our experiments that are presented in Section 4.3, this is a very promising technique for reducing the size of the FCS.

It should be noted that the above approach is only effective if the dynamic slices being intersected contain the faulty code. If even one of the dynamic slices fails to capture the faulty code, the intersection of dynamic slices surely will not contain the fault. However, in our experiments, the above problem did not arise.

3. OUR SLICING TOOL

We have developed a dynamic slicing tool that was used to conduct the experiments as described in the next section. In this section we briefly describe this tool, from the point of view of its usage and then we describe some additional implementation details.

3.1. Tool usage

Our tool executes a *gcc* compiler generated binary for Intel x86 under the control of the *Valgrind* [16] system. During the execution, dynamic dependences are identified and the dynamic dependence graph is constructed, as described in the beginning of Section 2. After the execution finishes, either because the program terminates or because the program crashes, the user is presented with a simple debugging interface that provides limited capabilities including the ability to request computation of a dynamic slice for an execution instance of an instruction that writes to a register, writes to a memory location, or represents the execution of a predicate. The slicing criteria used for a BwS is identified by the user and input into the system. The slicing criteria for forward dynamic slicing is computed separately and then input into the system by the user. The slicing criteria for computing a critical predicate is automatically determined by the tool. Once the slicing criteria are known to the tool, then through the traversals of the dynamic dependence graphs already available to the tool the computation of dynamic slices and their intersections are performed. Even though our tool works on the binary level, the slices are mapped back to source code level using the debugging information generated by *gcc*. For the library code, if debug information is not available, the slice is reported in terms of binary instructions. However, if the source code of the library is available, it can be recompiled with the debug option and then we can also report the portion of the slice from the library code at the source code level.

3.2. Implementation details

The main components of the tool carry out the following functions. The *static analysis* component of our tool computes static control dependence information required during slice computations from the binary. The static analysis was implemented using *Diablo* [17] retargetable link-time binary rewriting framework as this framework already has the capability of constructing the control flow graph from x86 binary. The *dynamic profiling* component of our system, which is based on *Valgrind* [16], that accepts the same *gcc* generated binary, dynamically instruments it by calling the *slicing instrumenter* and executes the instrumented code with the support of the *slicing runtime*. The slicing instrumenter and slicing runtime were developed by us to enable collection of dynamic information and computation of dynamic slices. Valgrind's kernel is a dynamic instrumenter that takes the binary and before executing any new (never instrumented) basic blocks it calls the instrumentation function, which is provided by the slicing instrumenter. The instrumentation function instruments the provided basic block and returns the new basic block to the Valgrind kernel. The kernel executes the instrumented basic block instead of the original. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The *slicing runtime* essentially consists of a set of call back functions for certain events (e.g. entering functions, accessing memory, binary operations, predicates, etc.). The static control dependence information computed by the static analysis component is represented based on the virtual addresses, which can be understood by Valgrind.

Table I. Faults used in the study.

Program	Fault description	Source
flex 2.5.31	(a) Some variable is not defined with option -l, which fails the compilation of xfree86	SourceForge [18]
	(b) String ']]' is not allowed in user's code	SourceForge [18]
	(c) The generated code contains extra #endif	SourceForge [18]
grep 2.5	Using -i -o together produces wrong output	Savannah [19]
grep 2.5.1	(a) Using -F -w together produces wrong output	Savannah [19]
	(b) Using -o -n together produces wrong output	Gmane [20]
	(c) "echo dofe — grep dofe" finds no match	Gmane [20]
make 3.80	(a) Backslashes in dependency names are not removed	Savannah [19]
	(b) Fail to recognize the updated file status while there are multiple target in the pattern rule	Savannah [19]
gzip-1.2.4	1024 byte long filename overflows into global variable	AccMon [21]
ncompress-4.2.4	1024 byte long filename corrupts stack return address	AccMon [21]
polymorph-0.4.0	2048 byte long filename corrupts stack return address	AccMon [21]
tar-1.13.25	Wrong loop bounds lead to heap object overflow	AccMon [21]
bc-1.06	Misuse of bounds variable corrupts heap objects	AccMon [21]
tidy-34132	Memory corruption problem	AccMon [21]
s-flex-v4	Assignment fault injected	Siemens Suite [15]
s-flex-v5	Assignment fault injected	Siemens Suite [15]
s-flex-v6	Assignment fault injected	Siemens Suite [15]
s-flex-v7	Assignment fault injected	Siemens Suite [15]
s-flex-v8	Predicate fault injected	Siemens Suite [15]
s-flex-v9	Predicate fault injected	Siemens Suite [15]
s-flex-v10	Assignment fault injected	Siemens Suite [15]
s-flex-v11	Assignment fault injected	Siemens Suite [15]

4. EXPERIMENTAL EVALUATION

In this section, we present results of our experiments. For these experiments we use faulty versions of commonly used programs (*flex*, *grep*, *make*, *gzip*, *ncompress*, *polymorph*, *tar*, *bc*, *tidy*, and *s-flex*). With the exception of faults in *s-flex*, all other faults are real faults reported by users of these programs. We obtained faulty versions of *s-flex* from the Siemens suite of programs [15]. These faulty versions were generated by injection of faults in the program. The description of the fault and the source of the faulty program version are given in Table I. The failing runs analyzed using slicing contained two types of situations. In some cases the programs produced incorrect output while for others the program simply crashed with a segmentation fault error.

The experiments we have conducted allowed us to study several issues. First, we report on the applicability of the three slicing techniques: BwS, FwS, and BiS. Second, we compare the dynamic slice sizes to see which technique is more effective in narrowing the size of the FCS. Finally, we give the results of an experiment conducted to study the synergy among these techniques, i.e. we identify the benefit of using *multiple points dynamic slicing*, which narrows the FCS by intersecting the resulting BwS, FwS, and BiS. In the remainder of this section we present the detailed results of the above experiments an analyze the data collected. We would like to mention that the conclusions drawn in the subsequent sections are based upon the failed runs that exercise 23 bugs. Further studies are required to determine to see if these conclusions hold for other types of applications and/or bugs.

4.1. Applicability

We first consider the applicability of the three types of dynamic slices. This information is provided in Table II where an entry of ✓ and × indicate whether the particular type of dynamic slice could be computed or could not be computed, respectively. In addition, the presence of ✓ indicates that the faulty code was indeed captured by the computed dynamic slice. In other words, although in general a dynamic slice may or may not capture the faulty code, for the failures studied the computed dynamic slices did contain the faulty code. Moreover, since the dynamic slices capture the faulty code, so do their intersections. Thus we can say that the slicing criteria used in computing the various types of dynamic slices are highly effective.

We also observe that while each dynamic slicing technique was applicable in well over half of the 23 failures studied, there were a few failures for which each of the techniques could not be applied. In particular, BwS could not be computed for five of the 23 failures, FwS could not be computed for three of the 23 failures, and BiS could not be computed for six of the 23 failures. In addition, we observe that while there was no failure for which all three techniques could not be applied, there was one failure for each respective technique where only one type of slicing technique was found to be applicable. We also observe that in many cases all three techniques were applicable. In conclusion, the results of our experiments show that all combinations of applicability and lack of applicability of the dynamic slicing techniques arise in this set of failed runs.

Let us briefly discuss the reasons for each of the dynamic slicing techniques not being applicable as observed in the failed runs studied.

- *No output*. Backward slicing was found to be not applicable for *grep*. The faults present in *grep* caused the program to terminate without the execution of any output producing statement (although the correct output is not no output). Since a backward slice is computed starting at the execution instance of a statement that produced an incorrect output value, for failing runs of *grep* no backward slicing criteria was available.
- *No failure inducing input*. A forward slice could not be computed for *s-flex* versions *v6*, *v8*, and *v11*. The inability to identify a failure inducing input is the reason why we could not compute the forward slice. We observed two situations in these programs. First, in some cases the fault was in form of an incorrect constant assignment (i.e. the constant value used in the constant assignment was incorrect). Therefore there was no failure inducing input. The second scenario we observed is described as follows. To identify the failure inducing input, according to the delta debugging algorithm by Zeller [13], we must begin with two know inputs: one on which the program fails and another on which the program runs successfully. Unfortunately the fault was such that there was no input in the given test suite for which the program did not fail. Therefore we could not apply the delta debugging algorithm successfully.

Table II. Applicability of dynamic slice types.

Program	BwS	FwS	BiS
flex 2.5.31 (a)	✓	✓	✓
(b)	✓	✓	×
(c)	✓	✓	×
grep 2.5	×	✓	✓
grep 2.5.1 (a)	×	✓	✓
(b)	×	✓	×
(c)	×	✓	✓
make 3.80 (a)	✓	✓	✓
(b)	✓	✓	✓
gzip-1.2.4	✓	✓	✓
ncompress-4.2.4	✓	✓	✓
polymorph-0.4.0	✓	✓	✓
tar-1.13.25	✓	✓	✓
bc-1.06	✓	✓	✓
tidy-34132	✓	✓	✓
s-flex-v4	✓	✓	✓
s-flex-v5	✓	✓	×
s-flex-v6	✓	×	×
s-flex-v7	✓	✓	✓
s-flex-v8	×	×	✓
s-flex-v9	✓	✓	✓
s-flex-v10	✓	✓	×
s-flex-v11	✓	×	✓

- *No critical predicate*. Bidirectional slice could not be computed for some versions of *flex 2.5.31*, *grep 2.5.1*, and *s-flex*. This is because we could not identify a critical predicate. Sometimes, a critical predicate cannot be found because none exists. For example, if the bug is complex, the simple switching of a single branch outcome may not result in the correct output being generated.

Thus, we see that there are different situations under which each of the above technique fails. Therefore, to enable the use of dynamic slicing in fault location across a broad range of programs and faults it is important to consider multiple types of dynamic slices.

4.2. Dynamic slice sizes

If dynamic slicing is not used, the programmer must search all statements executed during a failed run for faulty code. However, by employing dynamic slicing we reduce the size of the FCS from the set of executed statements. In this section we evaluate the degree to which this reduction is achieved by the

Table III. Comparison of dynamic slice sizes.

Program	LOC	EXEC (%LOC)	BwS (%EXEC)	FwS (%EXEC)	BiS (%EXEC)	MIN	(%LOC)
flex 2.5.31 (a)	26 754	1871 (6.99%)	695 (37.15%)	605 (32.36%)	225 (12.03%)	BiS:	225 (0.84%)
(b)	26 754	2198 (8.22%)	272 (12.37%)	257 (11.69%)	—	FwS:	257 (0.96%)
(c)	26 754	2053 (7.67%)	50 (2.44%)	1368 (66.63%)	—	BwS:	50 (0.19%)
grep 2.5	8581	1157 (13.48%)	—	731 (63.18%)	88 (7.61%)	BiS:	88 (1.03%)
grep 2.5.1 (a)	8587	509 (5.93%)	—	32 (6.29%)	111 (21.81%)	FwS:	32 (0.37%)
(b)	8587	1123 (13.08%)	—	599 (53.34%)	—	FwS:	599 (6.98%)
(c)	8587	1338 (15.58%)	—	12 (0.90%)	453 (33.86%)	FwS:	12 (0.14%)
make 3.80 (a)	29 978	2277 (7.60%)	981 (43.08%)	1239 (54.41%)	1372 (60.25%)	BwS:	981 (3.27%)
(b)	29 978	2740 (9.14%)	1290 (47.08%)	1646 (60.07%)	1436 (52.41%)	BwS:	1290 (4.30%)
gzip-1.2.4	8164	118 (1.45%)	34 (28.81%)	3 (2.54%)	39 (33.05%)	FwS:	3 (0.04%)
ncompress-4.2.4	1923	59 (3.07%)	18 (30.51%)	2 (3.39%)	30 (50.85%)	FwS:	2 (0.10%)
polymorph-0.4.0	716	45 (6.29%)	21 (46.67%)	3 (6.67%)	22 (48.89%)	FwS:	3 (0.42%)
tar-1.13.25	25 854	445 (1.72%)	105 (23.60%)	202 (45.39%)	117 (26.29%)	BwS:	105 (0.41%)
bc-1.06	8288	636 (7.67%)	204 (32.07%)	188 (29.56%)	267 (41.98%)	FwS:	188 (2.27%)
tidy-34132	31 132	1519 (4.88%)	554 (36.47%)	367 (24.16%)	541 (35.62%)	FwS:	367 (1.18%)
s-flex-v4	12 418	1631 (13.13%)	39 (2.39%)	877 (53.77%)	37 (2.27%)	BiS:	37 (0.30%)
s-flex-v5	12 418	1882 (15.16%)	692 (36.77%)	1187 (63.07%)	—	BwS:	692 (5.57%)
s-flex-v6	12 418	424 (3.41%)	156 (36.79%)	—	—	BwS:	156 (1.26%)
s-flex-v7	12 418	2045 (16.47%)	243 (11.88%)	910 (44.50%)	836 (40.88%)	BwS:	243 (1.96%)
s-flex-v8	12 418	610 (4.91%)	—	—	280 (45.90%)	BiS:	280 (2.25%)
s-flex-v9	12 418	1396 (11.24%)	236 (16.91%)	535 (38.32%)	230 (16.48%)	BiS:	230 (1.85%)
s-flex-v10	12 418	1683 (13.55%)	727 (43.20%)	970 (57.66%)	727 (43.20%)	BwS:	727 (5.85%)
s-flex-v11	12 418	1749 (14.08%)	102 (5.83%)	—	27 (1.54%)	BiS:	27 (0.22%)

three kinds of slices (BwS, FwS, and BiS). We also compare the sizes of these three types of slices to determine if one kind is preferable over the other types.

Let us consider the results presented in Table III. In this table LOC represents the lines of code in each program while EXEC is the LOC that are executed during the failing run being used to locate faulty code. The LOC belonging to each of the dynamic slices (BwS, FwS, and BiS) are also given. Finally, the MIN column gives the type and size of the smallest of the three kinds of slice.

First we observe that even though the executed lines of code as a percentage of total LOC (EXEC (%LOC)) ranges from only 1.45% to 16.47%, EXEC is still quite large (>1000 in 15 out of 23 cases). Therefore reduction in the size of the FCS is highly desirable. Second, we observe that the dynamic slicing techniques give significant reductions. The BwS, FwS, and BiS exceed 1000 in only 1, 4, and 2 cases, respectively. In parenthesis the slice sizes as a percentage of executed statements are also given. We observe that all three kinds of dynamic slices, when applicable, reduce the size of the FCS quite significantly. The size of BwS ranges from 2.39% to 47.08% of EXEC. The size of FwS ranges from 0.90% to 63.18% of EXEC. Finally, the size of BiS ranges from 1.54% to 60.25% of EXEC.

We define the *FCS* metric $FCS(xS)$ which indicates the fraction of LOC that are in the FCS across all the benchmarks when slice of the type xS is used. It is the ratio of the total LOC when a slice of type

xS is used ($\sum Size(xS)$) to the total LOC that are being executed ($\sum EXEC$). However, since we cannot compute all kinds of slices in all executions, we differentiate the executions into those where the slice can be computed ($App(xS)$) and those where it cannot be computed ($\overline{App(xS)}$). The full formula thus reads:

$$FCS(xS) = \frac{\sum_{App(xS)} Size(xS) + \sum_{\overline{App}(xS)} EXEC}{\sum_{All} EXEC}$$

From the data in Table III we find that $FCS(\text{BwS}) = 0.3781$, $FCS(\text{FwS}) = 0.4919$, and $FCS(\text{BiS}) = 0.4920$. Hence, using backward slices (BwS), on average, we only need to look at 37.8% of the executed statements. For forward and bidirectional slices this number is just over 49%. Thus according to this metric all three techniques are very effective. It should be noted that different benchmarks come with different number of versions and these benchmarks may not be cover all types of bugs.

From the MIN column in Table III we observe that no one type of dynamic slice is consistently the smallest. Out of 18 cases for which we were able to compute the BwS, in eight cases the BwS is the best choice (i.e. it is the smallest). In nine out of the 20 cases where we could compute the FwS if was found to be the best choice Finally, out of 17 cases where we could compute a bidirectional slice, if proved to be the best choice in six cases. From this perspective we can see that the FwS is slightly better than the BwS, which, in turn, is slightly better than the BiS. From the MIN column we also observe that in only one case the smallest slice size exceeds a thousand and in eight cases the slice size is no more than 50. The numbers in parenthesis are slice sizes as a percentage of LOC. As we can see, this number is quite small ranging from 0.04% to 6.98%. Given that the numbers in this range are quite small, we would like to point out that a substantial part of the reduction is the result of many statements in the program not being executed at all during the failed runs.

Next we further consider the bidirectional slice to see if the effort required by the programmer to examine the FCS BiS can be further reduced. We notice that BiS begins at the critical predicate and then extends backwards as well as forwards. Let us denote the backward and forward slices of the critical predicate as Bw/BiS and Fw/BiS, respectively. Instead of presenting the entire BiS to the programmer at once we can present it in three steps. First the critical predicate can be examined by the programmer and if the fault is in the predicate itself, the programmer is done. Otherwise, each component of the slice is presented separately. We considered three strategies for ordering the backward and forward components for presentation to the programmer: after the predicate, the backward component is examined before the forward component (PBF); after the predicate, the forward component is examined before the backward component (PFB); and after the predicate, either the forward component is examined first if the program crashed or the backward component is examined first if the program gave the wrong output (PBF/FB). The motivation for the last strategy is that we have observed that sometimes when changing the critical predicate outcome we avoid a program crash, as the execution of statements causing the crash is avoided and these statements are thus in the forward slice of the critical predicate.

Now let us examine the data in Tables IV and V. In Table IV we provide the sizes of Bw/BiS and Fw/BiS in comparison to BiS. In addition, we indicate the manner in which the error manifests itself (i.e. by producing a wrong output or causing the program to crash due to a segmentation fault). Finally, the Where column indicates the component of BiS that contains the faulty code (i.e. predicate, Bw/BiS, or Fw/BiS). Based upon the data in Table IV, we computed how many lines of code will be examined when the three ordering strategies (PBF, PFB, PBF/FB) are used and present the resulting data in Table V. In each case where the ordering strategy results in a reduction in the number of LOC that are

Table IV. Backward and forward parts of a bidirectional slice.

Program	Bw/BiS	Fw/BiS	BiS	Where	Behavior
flex 2.5.31 (a)	199	27	225	Predicate	Wrong output
(b)	—	—	—	—	Wrong output
(c)	—	—	—	—	Wrong output
grep 2.5	11	80	88	Fw/BiS	Wrong output
grep 2.5.1 (a)	101	15	111	Fw/BiS	Wrong output
(b)	—	—	—	—	Wrong output
(c)	448	7	453	Bw/BiS	Wrong output
make 3.80 (a)	468	1189	1372	Bw/BiS	Wrong output
(b)	1053	590	1436	Bw/BiS	Wrong output
gzip-1.2.4	37	4	39	Fw/BiS	Segmentation fault
ncompress-4.2.4	27	5	30	Fw/BiS	Segmentation fault
polymorph-0.4.0	20	4	22	Fw/BiS	Segmentation fault
tar-1.13.25	19	110	117	Fw/BiS	Segmentation fault
bc-1.06	207	106	267	Bw/BiS	Segmentation fault
tidy-34132	521	21	541	Bw/BiS	Segmentation fault
s-flex-v4	35	3	37	Predicate	Wrong output
s-flex-v5	—	—	—	—	Wrong output
s-flex-v6	—	—	—	—	Segmentation fault
s-flex-v7	239	604	836	Bw/BiS	Wrong output
s-flex-v8	131	173	280	Predicate	Wrong output
s-flex-v9	230	1	230	Predicate	Wrong output
s-flex-v10	623	18	640	Missed	Wrong output
s-flex-v11	24	4	27	Bw/BiS	Wrong output

examined before locating the fault, in parenthesis we present the examined LOC as a percentage of total LOC in BiS. As we can see, in many cases this approach can yield significant reductions. In four cases where the fault is in the predicate itself this approach is the most helpful. In other situations reductions can also be significant. We also observe that PBF, PFB, and PBF/FB strategies provide benefits in 11, 10, and 13 cases, respectively, out of a total of 18 cases where BiS could be computed. Thus, from the limited data points we have, we observe that the three strategies do not differ significantly in their effectiveness.

4.3. MPSs: dynamic chops

In the preceding section we carried out a relative evaluation of the three types of dynamic slices. In contrast, in this section we study the synergy between these techniques, which is essentially the motivation for the *multiple points dynamic slicing* that we proposed earlier in this paper. In particular, it is our goal to study whether MPSs are significantly smaller than the individual dynamic slices.

Table V. Heuristics for an ordering examination.

Program	PBF (%BiS)	PFB (%BiS)	PBF/FB (%BiS)
flex 2.5.31 (a)	1 (0.44%)	1 (0.44%)	1 (0.44%)
(b)	—	—	—
(c)	—	—	—
grep 2.5	88	80 (90.90%)	88
grep 2.5.1 (a)	111	15 (13.51%)	111
(b)	—	—	—
(c)	448 (98.90%)	453	448 (98.90%)
make 3.80 (a)	468 (34.11%)	1372	468 (34.11%)
(b)	1053 (73.33%)	1436	1053 (73.33%)
gzip-1.2.4	39	4 (10.26%)	4 (10.26%)
ncompress-4.2.4	30	5 (16.67%)	5 (16.67%)
polymorph-0.4.0	22	4 (18.18%)	4 (18.18%)
tar-1.13.25	117	110 (94.12%)	110 (94.12%)
bc-1.06	207 (77.53%)	267	267
tidy-34132	521 (96.30%)	541	541
s-flex-v4	1 (2.70%)	1 (2.70%)	1 (2.70%)
s-flex-v5	—	—	—
s-flex-v6	—	—	—
s-flex-v7	239 (28.59%)	836	239 (28.59%)
s-flex-v8	1 (0.36%)	1 (0.36%)	1 (0.36%)
s-flex-v9	1 (0.44%)	1 (0.44%)	1 (0.44%)
s-flex-v10	640	640	640
s-flex-v11	24 (88.89%)	27	24 (88.89%)

4.3.1. Dynamic chop

First we computed the sizes of the MPS obtained by intersecting the BwS with the FwS. Intersection of a forward and backward slice is also referred as a *dynamic chop*. The resulting data is given by the column labeled *Dynamic chop* in Table VI. We compare the size of this MPS with the size of the smaller of the BwS and FwS, which is given in column labeled Min(BwS, FwS) in the table. We only provide this data for those cases where both BwS and FwS were applicable since only in these cases can a dynamic chop be computed. From the data in Table VI we can see that the size of this dynamic chop can be significantly smaller than BwS or FwS. The numbers in parenthesis give the size of the dynamic chop as a percentage of the size of Min(BwS, FwS). As we can see, in seven out of 13 cases, the size of the dynamic chop slice is less than half the size of BwS or FwS.

4.3.2. Bidirectional dynamic chop

Following the dynamic chop we computed the sizes of the MPS obtained by intersecting all three dynamic slices: BwS, FwS, and BiS. The resulting data is given by the column labeled Bidirectional

Table VI. Sizes of dynamic chops and bidirectional dynamic chops.

Program	Min (BwS,FwS)	Dynamic chop	Min (BwS,FwS,BiS)	Bidirectional dynamic chop
flex 2.5.31 (a)	605	256 (42.31%)	225	27 (12.00%)
(b)	257	102 (39.69%)	—	—
(c)	50	5 (10.00%)	—	—
grep 2.5	—	—	88	86 (97.73%)
grep 2.5.1 (a)	—	—	32	25 (78.13%)
(b)	—	—	—	—
(c)	—	—	12	12
make 3.80 (a)	981	739 (75.33%)	981	739 (75.33%)
(b)	1290	1104 (85.58%)	1290	1051 (81.47%)
gzip-1.2.4	3	3	3	3
ncompress-4.2.4	2	2	2	2
polymorph-0.4.0	3	3	3	3
tar-1.13.25	105	103 (98.09%)	105	45 (42.86%)
bc-1.06	188	102 (54.26%)	188	102 (54.26%)
tidy-34132	367	164 (44.69%)	367	161 (43.87%)
s-flex-v4	39	7 (17.95%)	37	7 (18.92%)
s-flex-v5	692	544 (78.61%)	—	—
s-flex-v6	—	—	—	—
s-flex-v7	243	63 (25.93%)	243	63 (25.93%)
s-flex-v8	—	—	—	—
s-flex-v9	236	112 (47.46%)	230	112 (47.46%)
s-flex-v10	727	574 (78.95%)	727	574 (78.95%)
s-flex-v11	—	—	27	27

dynamic chop in Table VI. We compare the size of the bidirectional dynamic chop with the size of the smallest of the BwS, FwS, and BiS, which is given in column labeled Min(BwS, FwS, BiS) in the table. We only provide this data for those cases where all three (BwS, FwS, and BiS) were applicable. From the data in Table VI we can see that the size of the bidirectional dynamic chop can be significantly smaller than the size of the smallest of the BwS, FwS, and BiS. The numbers in parenthesis give the size of bidirectional dynamic chop as a percentage of the size of Min(BwS, FwS, BiS). We can see in six out of 11 cases, the size of the bidirectional dynamic chop is less than half the size of the smallest of the BwS, FwS, and BiS dynamic slices. Therefore bidirectional dynamic chopping is a very promising technique.

4.3.3. Discussion

Finally we would like to summarize the benefits that the dynamic slicing techniques studied provide to the programmer in carrying out fault location. Table VII summarizes the number of lines in the FCS produced using the techniques described in this paper, which when compared to the total LOC

Table VII. Summary of dynamic slice sizes.

Program	LOC	FCS (%LOC)
flex 2.5.31 (a)	26 754	27 (0.10%)
(b)	26 754	102 (0.38%)
(c)	26 754	5 (0.02%)
grep 2.5	8581	86 (1.00%)
grep 2.5.1 (a)	8587	25 (0.29%)
(b)	8587	599 (6.98%)
(c)	8587	12 (0.14%)
make 3.80 (a)	29 978	739 (2.47%)
(b)	29 978	1051 (3.51%)
gzip-1.2.4	8164	3 (0.04%)
ncompress-4.2.4	1923	2 (0.10%)
polymorph-0.4.0	716	3 (0.42%)
tar-1.13.25	25 854	45 (0.17%)
bc-1.06	8288	102 (1.23%)
tidy-34132	31 132	161 (0.52%)
s-flex-v4	12 418	7 (0.06%)
s-flex-v5	12 418	544 (4.38%)
s-flex-v6	12 418	156 (1.26%)
s-flex-v7	12 418	63 (0.51%)
s-flex-v8	12 418	280 (2.25%)
s-flex-v9	12 418	112 (0.90%)
s-flex-v10	12 418	574 (4.62%)
s-flex-v11	12 418	27 (0.22%)

in the test programs is very small. The number in parenthesis is the size of the FCS as a percentage of the LOC. As we can see, in 15 cases this percentage is no more than 1%. In all other cases it is only a few per cent. However, in a few cases the FCS contains a significant number of statements. In these cases the statements contained can be ranked according to their dependence distance from the erroneous output as proposed in [9,22]. As a study in [9] illustrates, such ranking leads to the user having to examine less than half of the statements in a BwS. Finally, fault location techniques such as ours eventually require the user to understand the cause of failure and to correct the faulty code. We believe this effort is reduced by using a dynamic-slicing-based approach because not only is the user able to examine faulty code but also the statements on which the faulty code depends and the statements that depend upon the faulty code. Examining the dependence relationships is very helpful in understanding the cause of the failure. Finally, we would like to mention that the examples shown in Section 2, which were taken from the failures studied in our experiments, illustrate that locating the fault by examining the dynamic slices can be quite easy in some cases. Additional examples from the considered bugs also illustrating a similar behavior can be found in a case study presented in [5].

4.4. Cost of dynamic slicing

There are two kinds of costs associated with our technique. The first cost is that of running an instrumented version of the program to capture the dynamic dependence graph from which the various kinds of dynamic slices are computed. The second cost results from the identification of the various slicing criteria (the erroneous output, the minimal failure inducing input difference, and a critical predicate instance) for which dynamic slices are computed and the cost of computing the dynamic slices is calculated.

Let us first consider the cost of constructing the dynamic dependence graph. As described earlier, the data and control dependences exercised during a program run are captured at runtime and then explicitly represented in the form of a dynamic dependence graph. The cost of carrying out the above task includes the space cost for representing the dynamic dependence graph and the execution time cost of executing the instrumented version of the program. Table VIII provides the number of executed instructions, the size of the dynamic dependence graph (DDG size), and the time taken to run the instrumented version of the program (DDG time). To reduce the space and time costs, we implemented an optimization that identifies the cases where dependences need not be captured or explicitly represented. In particular, if static analysis shows that an instruction always receives a data operand from a unique instruction or it is control-dependent upon a unique predicate, then the runtime and space cost of these dependences can be avoided. From the data in Table VIII we can see that the size of the graph ranges from hundreds of kilobytes to hundreds of megabytes for the program runs considered. The runtime cost of generating these graphs ranges from a few seconds to nearly 139 seconds. This study shows that for short program runs involving execution of up to 25 million instructions, our tool can be used interactively. Finally, we would like to mention that the original execution times of these program runs typically took less than one-tenth of a second. Therefore, the generation of dynamic dependence graphs slows down the program execution by two to three orders of magnitude. However, for short program runs, such as those considered in our experiments, the slowdown in execution is acceptable.

Now let us consider the cost of performing dynamic slicing. In comparison to the cost of building a dynamic dependence graph, the cost of computing dynamic slices is very small. For the programs considered, the computation of dynamic slices required no more than a few seconds (0.01–6.7 s). The identification of critical predicates also took little time (0.9–34.5 s). During the identification of minimal failure inducing input difference, the time taken by program executions was minimal as it does not even require execution of instrumented program versions.

The additional cost of using dynamic slicing is in terms of the potential programmer effort required to identify the slicing criteria. When using backward dynamic slicing, if the program crashes the erroneous value/address is the one whose use caused the crash. However, if the program does not crash but rather terminates with the wrong output, the user must inspect the output and determine the part that is incorrect. When using forward dynamic slicing, to identify the minimal failure inducing input difference, manipulation of the input can only be properly performed if the user provides a parser for the input. Moreover, the outputs of the program on modified inputs must be examined. Finally, when using bidirectional dynamic slicing, although the search for the critical predicate is entirely automated, the user must examine the output produced after altering a branch predicate instance outcome to determine if the output is correct. In our experiments the test inputs that cause the programs to fail and the corresponding correct outputs were already available to us and much of the above costs were avoided. However, in general this may not be the case.

Table VIII. Cost of constructing the dynamic dependence graph.

Program	Instructions executed (millions)	Instrumented execution: DDG size (KB)	Instrumented execution: DDG time (seconds)
flex 2.5.31 (a)	24.95	196 131	135.5
(b)	25.77	202 441	138.9
(c)	25.40	199 131	130.2
grep 2.5	0.11	760	35.5
grep 2.5.1 (a)	0.12	794	29.2
(b)	0.06	333	4.4
(c)	0.15	968	20.1
make 3.80 (a)	2.09	17 409	28.0
(b)	2.40	15 801	34.6
gzip-1.2.4	0.03	164	1.2
ncompress-4.2.4	0.02	211	1.1
polymorph-0.4.0	0.03	173	0.4
tar-1.13.25	0.14	420	10.9
bc-1.06	0.19	1404	6.7
tidy-34132	2.66	14 928	12.9
s-flex-v4	5.86	33 195	16.6
s-flex-v5	1.16	6173	18.6
s-flex-v6	0.15	743	2.6
s-flex-v7	1.24	6650	14.4
s-flex-v8	0.16	797	3.6
s-flex-v9	0.91	4740	8.4
s-flex-v10	1.20	6457	10.5
s-flex-v11	1.22	6485	10.9

Let us briefly discuss the scalability of our technique. As we observed from the above experiments, for short program runs of a few million instructions, our tool can be used interactively. For longer program runs of a few billion instructions, the construction of the dynamic dependence graph may take a long time and therefore it must be performed non-interactively. This is the result of a slowdown ranging from one to three orders of magnitude being observed when instrumented programs are run. Once the graph has been built, the computation of dynamic slices can be carried out interactively. This is because in our prior work [3,23] we developed a compressed dynamic dependence graph representation, which enables the dynamic dependence graph of program runs of a couple of billion instructions to be held in less than 1 GB of memory. Moreover, the dynamic slices for these longer program runs can be computed in tens of seconds [3,23]. For very long programs runs of tens of billions of instructions, the dynamic dependence graph would have to be kept on disk and thus the cost of dynamic slicing will also increase. Therefore, the computation of dynamic slices must also be carried out non-interactively. Finally, we would like to note that the scalability of our technique is limited mainly by the length of the program run and to a lesser extent by the static size of the program.

5. RELATED WORK

5.1. Dynamic slicing

Dynamic slicing was introduced as an aid to debugging by Korel and Laski in 1988 [2]. Although the idea seemed very promising, it has not been used in practice. There is a practical reason for this. The problem of the high cost of computing dynamic slices had, until recently, not been addressed. In recent work [3,24], we developed practical implementations of dynamic slicing for both backward computation [3,7] and forward computation [24,25] algorithms. We demonstrated that dynamic slices of program runs that were 67–140 million instructions in length, on an average, took 1.92–36.25 s to compute [3].

Dynamic slicing has been studied as an aid to debugging by many researchers [8,26–30]. Agrawal *et al.* [28] proposed subtracting a single correct execution trace from a single failed execution trace. In [29], Pan and Spafford presented a family of heuristics for fault localization using dynamic slicing. Compared to these previous works, we are the first to compare the effectiveness of dynamic slicing algorithms in fault location.

General studies of dynamic slice sizes have also been conducted. For example, our work in [3] showed that the number of distinct statements executed at least once during a program execution were 2.46–56.08 times more than the number of statements in the dynamic slice. However, these results are based upon computing dynamic slices of randomly selected values computed by correct versions of programs. In another study [9] we compared variants of BwSs (data slices, full slices, and relevant slices) based upon failed runs of faulty versions of programs. In contrast, the study presented in this paper performs comparative evaluation of the BwS of an erroneous value, the FwS of a failure inducing input, and a BiS of a critical predicate. We also proposed the multiple points dynamic slicing technique and demonstrated its effectiveness.

5.2. Intersecting and differencing dynamic slices: dynamic chops and dynamic dicing

Since the main idea of slicing is to focus the user's attention on a relevant subset of statements in the program, it is only natural that researchers have explored techniques for narrowing the relevant set of statements beyond what is contained in a single slice. In [31], a dynamic chop, which is the dynamic dependence graph between two nodes, is used to derive dynamic path conditions. A constraint solver is then used to test whether the derived conditions can be satisfied. If so, the resolved input serves as a witness to the failure. If not, there is no dependence between the two nodes even though there exists a dependence path between them. In [32] difference between backward slices is computed with the aim of eliminating those statements that are less likely to be faulty from the backward slice of an erroneous output. In comparison with the above works the main contribution of our work is as follows. We have identified criteria whose dynamic slices are highly effective in capturing faulty code and therefore their intersection also captures the faulty code. This is a significant contribution especially in light of an experimental evaluation of a form of dicing presented in [33]. In [33] it is shown that the dynamic dice computed by taking the difference between BwSs of the erroneous output and correct output very often fails to capture the faulty code.

5.3. Delta debugging

A significant amount of research in the field of fault location has involved study in areas other than dynamic slicing. Zeller has presented a series of techniques [12,34,35] from isolating the critical input to isolating cost-effect chains in both space and time. The basic idea is to find the specific part of the *input/program state*, which is critical to the program failure by minimizing the difference between the *input/program state* leading to a passing run and that leading to a failing run. We believe our technique can be combined with Zeller's technique in many aspects, for instance the isolated *causes* are perfect slicing criteria from which dynamic slicing may produce a much smaller FCS than from the failure point.

5.4. Statistical debugging

Recently, a significant amount of research has focused on the use of statistical techniques for fault location [36–39]. Our dynamic slicing work differs from statistical debugging techniques in several significant ways, and to some extent our approach is complimentary to statistical techniques. Statistical debugging techniques rely on dynamic information (e.g. patterns of predicate outcomes) collected for a large number of program runs. In contrast, in our approach, all dynamic slices are based upon the dynamic dependence graph of a single failed program run. Statistical techniques may be able to identify bugs whose existence may not be known while the scenario we have considered assumes that a failing run is already available. Another important characteristic of statistical techniques is that they rank program statements according to a score that captures the likelihood of the statement representing faulty code. The programmer can then examine the statements in the order of ranking to locate faulty code. In this paper, we report the contents of the slice to the programmer who can then examine the statements and the dependences among them to locate faulty code. However, ranking can be used in conjunction with dynamic slicing. In [9,22] the statements in BwSs are ranked according to their dependence distance from the point at which erroneous output is observed and in [33] they are ranked according to *confidence* values that measure the likelihood that they produced correct results. Experimental data in [9,33] demonstrates that these ranking schemes can significantly reduce the portion of the dynamic slice that is examined before identifying the faulty code. We would also like to note that the techniques in [22] and [37] make use of ranking to enhance the visualization aspect of the fault location environment.

Finally, we would like to comment on a metric that is sometimes used to evaluate statistical fault location strategies. It is often reported that by examining a certain percentage of code, a certain number of bugs were detected. Such data is computed by first, for each bug, examining the statements in the order of ranking and determining what percentage of code is examined until the faulty code for that bug is encountered and then aggregating such data across all bugs. While the above data is useful, it is to some extent optimistic owing to the following reason. In practice, when a user examines a statement to see if it contains a fault, the user can be expected to examine additional statements that are related to it through dependences in order reach any conclusion. However, the above metric does not take into account the inspection of these additional statements. In contrast, we have reported the sizes of slices as a percentage of the number of statements in the program. These slices include chains of dependences whose inspection helps the user identify faulty code and understand its cause. For example, in Figure 2, when the user examines the relevant statements (i.e. statements at lines 167, 176, and 177) and the dependences among them, the cause of the fault becomes obvious. However, by simply examining the statement at line 167 it is difficult identify the fault. Therefore we consider the size of slice as a more accurate indicator of the user effort needed to identify faulty code. Nevertheless, as we can see from our data, even if the entire slice is examined, this represents only a small fraction of the total statements in the program.

To elaborate on the above point let us consider the technique proposed by Renieris and Reiss [36]. This paper proposes the nearest neighbor model to identify potentially faulty statements by differencing program spectra of a failing run with that of a successful run with the closest spectra. The statements present in the difference are identified as being potentially faulty. We note that during differencing, the statements (and hence the dependences) that are common to the failing and successful run will be eliminated. However, these dependences may be crucial in understanding the fault. In contrast such dependences will be captured by the dynamic slice. To compare the nearest neighbor model with other differencing models (i.e. union and intersection models [36]), scores for statements are computed as follows. For each potentially faulty statement, the smallest dependency sphere that includes a faulty node is found. It is assumed that the correct version of the program is available so that the faulty nodes can be found by comparing the correct and the faulty version. Scores are then computed by normalizing the number of nodes in the smallest dependency sphere with respect to the nodes in the overall program dependency graph. Note that in [36] the dependences are used to compute the scores used for evaluating the proposed fault localization model and not for identifying the faulty statement(s). In contrast, dynamic slicing uses dynamic dependences to identify the set of potentially faulty statements.

5.5. Other work

We have already described several types of dynamic analysis being used to assist in fault location. We would like to mention that the development of fault location strategies and applications of dynamic analysis represent very active areas of research that are being studied under many different scenarios. For example, dynamic information is also being used to compare behaviors of different program versions [40,41] and fault location strategies are being developed specifically to assist the growing community of end-user programmers [42].

6. CONCLUSIONS

The development of dynamic slicing was motivated by the problem of locating the faulty code when an execution of a program fails. There has been a significant amount of research on the use of BwS. The contribution of this paper is to first present a comparative study of BwS, FwS, and BiS and then to propose the multiple points dynamic slicing technique that is observed to be significantly more effective than the individual dynamic slicing techniques. Our work shows that there are significant advantages to supporting multiple kinds of dynamic slices. We observe that for each type of dynamic slice there are distinct situations in which it is not applicable. Therefore we must support multiple types of slices in order to handle a wide range of situations. When multiple types of dynamic slicing techniques are applicable to a situation, these slices can be used together to compute multiple points dynamic slices which provide significantly smaller FCSs than if the dynamic slices were being used individually.

ACKNOWLEDGEMENTS

We are grateful to the reviewers for their detailed and constructive comments. The comments were very helpful for improving both the content and presentation of the material in this paper.

This work is supported by grants from Microsoft, IBM, and NSF grants CNS-0614707, CCF-0541382, CCF-0324969, and EIA-0080123 to the University of Arizona.

REFERENCES

1. Weiser M. Program slicing. *IEEE Transactions on Software Engineering* 1982; **10**(4):352–357.
2. Korel B, Laski J. Dynamic program slicing. *Information Processing Letters* 1988; **29**(3):155–163.
3. Zhang X, Gupta R. Cost effective dynamic program slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004. ACM Press: New York, 2004; 94–106.
4. Gupta N, He H, Zhang X, Gupta R. Locating faulty code using failure-inducing chops. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, November 2005. ACM Press: New York, 2005; 263–272.
5. Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. *Proceedings of the International Conference on Software Engineering*, Shanghai, China, May 2006. ACM Press: New York, 2006.
6. Zeller A. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann: San Fransisco, CA, 2005.
7. Agrawal H, Horgan J. Dynamic program slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York, 1990; 246–256.
8. Agrawal H, DeMillo R, Spafford E. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience* 1993; **23**(6):589–616.
9. Zhang X, He H, Gupta N, Gupta R. Experimental evaluation of using dynamic slices for fault location. *Proceedings of the SIGSOFT-SIGPLAN 6th International Symposium on Automated and Analysis-Driven Debugging*, Monterey, CA, September 2005. ACM Press: New York, 2005; 33–42.
10. Agrawal H, Horgan JR, Krauser EW, London SA. Incremental regression testing. *Proceedings of the IEEE Conference on Software Maintenance*, Montreal, Canada, 1993. IEEE: Piscataway, NJ, 1993; 348–357.
11. Zeller A. Yesterday, my program worked. Today, it does not. Why? *Proceedings of the 7th European Software Engineering Conference, 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, September 1999. Springer: Berlin, 1999; 253–267.
12. Hildebrandt R, Zeller A. Simplifying failure-inducing input. *Proceedings of the International Symposium on Software Testing and Analysis*. ACM Press: New York, 2000; 135–145.
13. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 2002; **28**(2):183–200.
14. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of the 16th International Conference on Software Engineering*. ACM Press: New York, 1994; 191–200.
15. Software-artifact Infrastructure Repository. http://www.cse.unl.edu/∼galileo/sir.
16. Valgrind. http://valgrind.org/.
17. Diablo Is A Better Link-time Optimizer. http://www.elis.ugent.be/diablo/.
18. SourceForge. http://soureforge.net/.
19. Savannah. http://savannah.gou.org/.
20. Gmane: Bug Reports and Development Discussion. http://comments.gmane.org/gmane.comp.gnu.grep.bugs/.
21. Zhou P, Liu W, Fei L, Lu S, Qin F, Zhou Y, Midkiff SP, Torrellas J. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *Proceedings of the 37th Annual International Symposium on Microarchitecture*. IEEE: Piscataway, NJ, 2004; 269–280.
22. Krinke J. Visualization of program dependence and slices. *Proceedings of the International Conference on Software Maintenance*. IEEE: Piscataway, NJ, 2004; 168–177.

23. Zhang X, Gupta R. Whole execution traces and their applications. *ACM Transactions on Architecture and Code Optimization* 2005; **2**(3):301–334.
24. Zhang X, Gupta R, Zhang Y. Effective forward computation of dynamic slices using reduced ordered binary decision diagrams. *Proceedings of the IEEE International Conference on Software Engineering*, Edinburgh, U.K., 2004. ACM Press: New York, 2004; 502–511.
25. Beszedes A, Gergely T, Szabo ZM, Csirik J, Gyimothy T. Dynamic slicing method for maintenance of large C programs. *Proceeding of the 5th European Conference on Software Maintenance and Reengineering*, March 2001. IEEE: Piscataway, NJ, 2001; 105–113.
26. Kamkar M. Interprocedural dynamic slicing with applications to debugging and testing. *PhD Thesis*, Linköping University, 1993.
27. Korel B, Rilling J. Application of dynamic slicing in program debugging. *Proceedings of the 3rd International Workshop on Automatic Debugging*. Linköping Electronic Articles in Computer and Information Science: Linköping, Sweden, 1997; 43–58.
28. Agrawal H, Horgan J, London S, Wong W. Fault localization using execution slices and dataflow tests. *Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*. IEEE: Piscataway, NJ, 1995; 143–151.
29. Pan H, Spafford EH. Heuristics for automatic localization of software faults. *Technical Report SERC-TR-116-P*, Purdue University, 1992.
30. Wang T, Roychoudhary A. Using compressed bytecode traces for slicing java programs. *Proceedings of the International Conference on Software Engineering*. ACM Press: New York, 2004; 512–521.
31. Hammer C, Grimme M, Krinke J. Dynamic path conditions in dependence graphs. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics*. ACM Press: New York, 2006; 58–67.
32. Chen TY, Cheung YY. Dynamic program dicing. *Proceedings of the International Conference on Software Maintenance*. IEEE: Piscataway, NJ, 1993; 378–385.
33. Zhang X, Gupta N, Gupta R. Pruning dynamic slices with confidence. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006. ACM Press: New York, 2006; 169–180.
34. Zeller A. Isolating cause-effect chains from computer programs. *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, SC, 2002. ACM Press: New York, 2002; 1–10.
35. Cleve H, Zeller A. Locating causes of program failures. *Proceedings of the 27th International Conference on Software Engineering*. IEEE: Piscataway, NJ, 2005; 342–351.
36. Renieris M, Reiss S. Fault localization with nearest neighbor queries. *Proceedings of the IEEE International Conference on Automated Software Engineering*, 2003; 30–39.
37. Jones JA. Fault localization using visualization of test information. *Proceedings of the 26th International Conference on Software Engineering*. ACM Press: New York, 2004; 54–56.
38. Liu C, Yan X, Fei L, Han J, Midkiff S. SOBER: Statistical model-based bug localization. *Proceedings of the 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press: New York, 2005; 286–295.
39. Liblit B, Aiken A, Zheng AX, Jordan MI. Bug isolation via remote program sampling. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press: New York, 2003; 141–154.
40. Harrold MJ, Rothermel G, Wu R, Yi L. An empirical investigation of program spectra. *Proceedings of the ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*. ACM Press: New York, 1998; 83–90.
41. Zhang X, Gupta R. Matching execution histories of program versions. *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, September 2005. ACM Press: New York, 2005; 197–206.
42. Ruthruff JR, Burnett M, Rothermel G. An empirical study of fault localization for end-user programmers. *Proceedings of the International Conference on Software Engineering*. ACM Press: New York, 2005; 352–361.