Locating Faults Through Automated Predicate Switching

Xiangyu Zhang   Neelam Gupta   Rajiv Gupta

The University of Arizona
Department of Computer Science
Tucson, Arizona 85721

ABSTRACT

Typically debugging begins when during a program execution a point is reached at which an obviously incorrect value is observed. A general and powerful approach to automated debugging can be based upon identifying modifications to the program state that will bring the execution to a successful conclusion. However, searching for arbitrary changes to the program state is difficult due to the extremely large search space. In this paper we demonstrate that by forcibly switching a predicate's outcome at runtime and altering the control flow, the program state can not only be inexpensively modified, but in addition it is often possible to bring the program execution to a successful completion (i.e., program produces the desired output). By examining the switched predicate, also called the critical predicate, the cause of the bug can then be identified. Since the outcome of a branch can only be either true or false, the number of modified states resulting by predicate switching is far less than those possible through arbitrary state changes. Thus, it is possible to automatically search through modified states to find one that leads to the correct output. We have developed an implementation based upon dynamic instrumentation to perform this search through program re-execution – the program is executed from the beginning and a predicate's outcome is switched to produce the desired change in control flow. To evaluate our approach, we tried our technique on several reported bugs for a number of UNIX utility programs. Our technique was found to be practical (i.e., acceptable in time taken) and effective (i.e., we were able to automatically identify critical predicates). Moreover we show that bidirectional dynamic slices of critical predicates capture the faulty code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Debuggers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools, Tracing*

General Terms

Algorithms, Measurement, Reliability, Verification

Keywords

automated debugging, dynamic slicing, predicate switching

1. INTRODUCTION

A programmer often becomes aware of the existence of bugs in a program when he/she observes that a program output deviates from the expected output. A standard debugging process consists of setting breakpoints, re-executing the program on the failed input, and examining the program state (e.g., variable values, call stack, etc.) to understand the cause of incorrect output being generated. During this process, the programmer must decide what part of execution to explore to isolate the bug. This process of exploration is often tedious and time consuming. Since the processing power of machines has drastically increased, it is highly desirable to utilize this computing power to make the task of finding bugs less tedious for the programmer. Therefore automated debugging techniques are being explored by researchers. Some examples of such techniques include delta debugging [16, 7, 17, 15, 2], variants of backward dynamic slicing [13, 9, 1, 18, 19, 20, 21], and failure inducing chops [3].

Let us assume that on a given input we observe that the execution of a program fails. An aggressive and general approach to automated 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 program state that cause the program to terminate correctly, we will often be able to determine the cause of the program failure. 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 float).

In this paper we overcome the above problem using the following approach. The state changes are simulated by changing branch predicate outcomes at runtime. More precisely, even though a branch predicate evaluates to true (false), we force the execution along the false (true) path at runtime (note that we do not alter outcomes of switch statements or indirect branches). This process, that is called *predicate switching*, is performed by running the program under the control of a dynamic instrumentation system. By restricting the simulated changes to program state to predicate switching, we greatly reduce the state search space since a branch predicate has only two possible outcomes, true or false. Our goal is to find a runtime predicate switch that causes the program to produce correct results. The predicate instance whose switching produces correct result is called a *critical predicate*. Predicate switching is simple and surprisingly powerful in producing desired state changes. The changes result because, following a predicate switch, the set of assignment statements executed by the program is altered.

We illustrate predicate switching using the faulty version of the *flex* (a fast lexical analyzer generator) program shown in Figure 1

```
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 1: Example of flex.

which is taken from the following website [24]. This website provides the faulty versions and associated test suites for several programs. The program in Figure 1 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 which produced an erroneous output. We observed that the output differs from the expected output at the 538th character. A '1' is produced as output due to the execution of *printf* in the *else* part (at line 2696) instead of a '0' that should be output by execution of the *printf* in the *else if* part (at line 2690). 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*. Due to 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 backwards the data dependences, that the incorrect value of *ch[0]* was a stale value and it did not come from most recent execution of 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. The above example also illustrates that it is important to alter the outcome of a *selected predicate instance* 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. In the above *flex* example, 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 can take many different values.

The remainder of the paper is organized as follows. In section 2 we begin with a study that provides insight into the power of predicate switching. In section 3 we describe how automated predicate switching is performed. To improve the runtime of our approach strategies for switching and the order in which switchings are explored are considered. In section 4 we describe our dynamic instrumentation framework used to implement predicate switching. In section 5 we present an experimental evaluation of the success rate and runtime cost of predicate switching. We show how to use results of predicate switching in identifying faulty code with the assistance of dynamic slicing techniques. Related work is discussed in section 6 and conclusions are given in section 7.

2. MOTIVATING STUDY

The motivation of the above approach based on predicate switching can be found in the following observation. Given a program run, from the perspective of a program output, the computation performed to compute an output can be divided into two parts: the *Data Part* (*DP*) and the *Select Part* (*SP*).

The *Data Part*, *DP*, consists of executed instructions which compute data values that are involved in computing the actual value that is output. These instructions can be identified by computing the backward data slice, i.e. transitive closure of dynamic data dependences starting from the output value. Note that the dynamic data slice does not contain any branches or branch predicates. For the example in Figure 2 the *data part* will consist of statement at line 1, definition of x (statement at line 4 or 5 depending upon outcome of branch predicate at line 3), and definition of y (statement at line 9 or 10 depending on outcome of branch predicate at line 8).

```
 1.  read(a,b);
 2.  c = f(a,b);
 3.  if c < 5
 4.  then x=a+b
 5.  else x=a-b
 6.  endif
 7.  d = g(a,b);
 8.  if c < 5
 9.  then y=a*b
10.  else y=a/b
11.  endif
12.  output(x+y)
```

Figure 2: Data Part and Select Part.

The *Select Part*, *SP*, is essentially the part of the computation that caused the selection of instructions in the dynamic data slice for execution. Different executions of a program may involve execution of different dynamic data slices leading to generation of differently computed output values. The presence of faulty code in *SP* may cause an incorrect dynamic data slice to be selected for execution and thus the generation of a wrong output value. In contrast to *DP*, the size of *SP* is much bigger. *SP* is computed by unioning the full dynamic slices of the predicates on which the instructions in the data slice are dynamically control dependent. A full dynamic slice is computed by taking the transitive closure over dynamic data and control dependences starting at a predicate. In the example in Figure 2 there are four possible data slices: (1,4,9), (1,4,10), (1,5,9) and (1,5,10). The data slice that is executed to compute the output value is determined by outcomes of branch predicates at lines 3 and 8. Therefore the select part includes the two predicates (3 and 8) and statements (1,2,7) that compute values used by the predicates.

Now let us consider the possible situations under which an erroneous result is produced. If the faulty code in the program belongs to *DP*, typically finding the fault is often not difficult as dynamic data slices are relatively small and the error is a computational error (e.g., a mistake in an expression). If the faulty code is not present in the *DP*, we need to examine a much larger *SP*. When the faulty code belongs to *SP*, the goal of exploring state changes is to affect the contributions to the program state by instructions in *SP* such that correct output can be generated. No matter whether the faulty code in *SP* is a branch predicate or an instruction that computes a value that is used in the evaluation of a branch predicate, the effect of the fault is the same, i.e. selection of a wrong data slice for execution and hence the generation of an incorrect output value. Therefore simply switching branch predicate outcomes is an appropriate way to explore possible changes in program state due to *SP* that may produce the correct output. This is an important observation because the number of branch predicates evaluated (i.e., the number of conditional branches encountered) is significantly smaller than the entire *SP* computation.

From the above discussion we conclude that the effects of presence of faulty code in a large part of the computation, i.e. the *SP* computation, can be often overcome by causing state changes by switching predicates that represent a small part of the *SP* computation. We carried out an experimental study to confirm our two claims on which the above reasoning is based: the *DP* computation is much smaller than the *SP* computation; and the branch predicates are a small part of *SP* computation. In this study instrumented binaries of programs were run to collect the desired data. The results of this study are presented in Table 1. In this table we present the total number of instructions executed (Total # Executed), instructions in *DP* and *SP* as a percentage of total instructions executed, and the number of instances of conditional branches in *SP* both as a percentage of total executed instructions and the actual number. The data presented is obtained by averaging the above information across all distinct output values and corresponds to the run on the first test input which came with each package. From the data we can clearly see that the number of executed instructions in *DP* is a small percentage (0.26% - 8.33%) of the total number of executed instructions and this is 3 to 7 times smaller than the number of executed instructions in *SP*. Note that *DP* and *SP* do not add up to 100% because they represent only part of the overall computation. Second we can see that the number of conditional branch instructions (*Preds*) in *SP* is quite small in comparison to the total number of instructions in *SP* as well as the total number of executed instructions. Therefore exploring modifications to program state by switching outcomes of *Preds* is far more manageable than exploring the effects on program state by making changes to values computed by all instructions in *SP*.

Table 1: Distribution of Executed Instructions.

Program	Total # Executed	*DP* %	*SP* %	*Preds* % (#)
flex-2.5.31	17,895,047	0.26	1.76	0.29 (51,691)
grep-2.5.1	160,071	6.10	35.0	5.58 (8,924)
make-3.80	1,181,569	6.70	36.2	6.61 (78,065)
bc-1.06	3,943,354	8.33	21.1	1.35 (53,102)
gzip-1.2.4	1,598,515	6.13	16.7	2.58 (41,305)
tar-1.13.25	83,353	5.10	15.8	3.23 (2,696)
tidy	9,556,715	7.06	31.4	5.68 (542,540)

In our discussion so far we have assumed that the execution of faulty code causes the program to terminate with a wrong output value. However, our approach also applies to situations where the observed erroneous value is not an output value. In practice it is very common that the execution of faulty code will cause a program to terminate prematurely (e.g., divide by zero, memory segmentation error). In such situations the data or address reference which caused the program to fail and terminate prematurely must be clearly erroneous. Therefore the goal of predicate switching would be to make the generation of this erroneous value, and corresponding error, to go away. In fact our experiments consider some faulty versions of programs that produce incorrect outputs and some faulty versions that crash.

3. AUTOMATED PREDICATE SWITCHING

In this section we develop a detailed algorithm for predicate switching. As we have already stated, the general idea of our approach is to perform repeated executions of the program on the failing input and switch conditional branch outcomes during these re-executions till we find a predicate switching that causes the program to produce the correct output. In doing so, it is our goal to develop a search strategy that is both practical and effective. To achieve this goal we design a search strategy which incorporates the following features that together limit the search space and order the search.

- *Only one predicate switch at a time.* Even though predicate switching greatly reduces the search space by limiting state changes to conditional branch outcomes, there are still a substantial number of instances of executed conditional branches whose outcomes are candidates for switching (see last column of Table 1). Therefore we limit our search such that during each new program execution, the outcome of only a *single predicate instance* is switched. In other words, we do not explore program behavior by simultaneously switching outcomes of multiple predicate instances because number of such possibilities is very large.

- *Last Executed First Switched (LEFS) Ordering.* Now that we have decided that we will switch the outcome of one branch predicate instance in each re-execution, we decide on the order in which possible predicate switchings are explored. One simple ordering strategy that we employ is based upon the following observation: *execution of faulty code (i.e., root cause of a failed run) is often not far away from the point at which the program fails (e.g., program crashes or it produces a wrong output value).* Therefore we explore possible predicate switchings in the reverse order of the predicate executions, i.e. the outcome of the conditional branch instance encountered last is switched first.

- *Prioritization-based (PRIOR) Ordering.* In addition to the simple *LEFS* ordering strategy, we developed another more aggressive prioritization based ordering strategy (*PRIOR*) that we describe next. This strategy consists of two main steps. In the first step we use an algorithm to partition the set of all branch predicate instances into two subsets: those that are expected to be influenced by the faulty code via dependences and those that are not expected to be influenced by faulty code. The ones in the first subset are explored before the ones in the second subset. In the second step we order the branch predicate instances within the first subset according to their dynamic dependence distance from the erroneous output value. More precisely, the predicate instances that are separated by fewer dependence edges from the erroneous output value are explored before those that are separated by greater number of dependence edges. The resulting ordering of branch predicate instances is then used in our search.

Next we present some data confirming the observations on which the above design choices are made. This data is based upon a set of faulty versions of several commonly used programs. In Table 2 the names of programs, description of faults, the sources of faulty versions, and times at which the faults were reported are given. All faults are real reported faults except for those in versions of *s-flex*. Now let us consider the data in Table 3. The total number of instructions executed, excluding the instructions from library code, before program terminated during a failing run is given first in column (*Total Instns*). In 15 out of 20 faults that were studied we were

Table 2: **Suite of Faulty Versions Used in Experiments.**

Program	Bug Description	Report Website	Report Date	Output
flex 2.5.31	(a) some variable is not defined with option -l, which fails the compilation of xfree86	http://soureforge.net	12/27/2003	wrong
	(b) string ”]]” is not allowed in user’s code	http://soureforge.net	2/18/2004	wrong
	(c) the generated code contains extra #endif	http://soureforge.net	8/22/2003	wrong
grep 2.5	using -i -o together produces wrong output	http://savannah.gnu.org	7/25/2004	wrong
grep 2.5.1	(a) using -F -w together produces wrong output	http://savannah.gnu.org	8/16/2003	wrong
	(b) using -o -n together produces wrong output	http://comments.gmane.org/ gmane.comp.gnu.grep.bugs/	-	wrong
	(c) ”echo dor̂e — grep dor̂e” finds no match	http://comments.gmane.org/ gmane.comp.gnu.grep.bugs/	4/12/2005	wrong
make 3.80	(a) Backslashes in dependency names are not removed	http://savannah.gnu.org	2/24/2005	wrong
	(b) Fail to recognize the updated file status while there are multiple target in the pattern rule	http://savannah.gnu.org	2/21/2005	wrong
bc-1.06	misuse of bounds variable corrupts heap objects	AccMon [23]	-	crash
tar-1.13.25	wrong loop bounds lead to heap objects overflow	AccMon [23]	-	crash
tidy	memory corruption	AccMon [23]	-	crash
s-flex	8-versions; errors in a single statement/predicate	Website [24]	-	wrong

Table 3: Search Strategies: ***LEFS*** **vs.** ***PRIOR*****.**

Program	Total Instr.	After Fault	Dep . dist.	Total Preds	LEFS	PRIOR
flex 2.5.319(a)	17,637	2,583	23	3,669	432	6
flex 2.5.319(b)	366,624	search		60,481	failed	
flex 2.5.319(c)	303,121	search		46,820	failed	
grep 2.5	21,001	416	27	2,555	61	56
grep 2.5.1 (a)	4,290	232	26	844	38	21
grep 2.5.1 (b)	10,337	search		1,652	failed	
grep 2.5.1 (c)	41,068	185	15	9,561	32	28
make 3.80 (a)	1,907,361	163,050	23	166,837	155,492	102
make 3.80 (b)	1,787,616	404,400	50	218,778	50,909	7,108
bc-1.06	68,336	15,676	6	9,684	2,079	2
tar-1.13.25	2,471	1,783	12	470	388	3
tidy	771,154	108	3	131,336	39	1
s-flex-v4	321,888	11,728	5	59,352	4,228	37
s-flex-v5	171,953	search failed		30,203	error in DP	
s-flex-v6	8,252	search failed		1,717	error in DP	
s-flex-v7	187,903	139,799	21	33,136	26,028	6
s-flex-v8	9,848	1,522	NA	1,943	218	NA
s-flex-v9	69,258	59,209	33	13,010	11,085	190
s-flex-v10	177,821	41	16	29,240	4	4
s-flex-v11	185,724	27,809	11	33,199	7,189	13

able to find a predicate instance to switch that caused the failure to be removed. We refer to this predicate as the *critical predicate*. In three cases (faults (b) and (c) in *flex 2.5.319*; fault (b) in *grep 2.5.1*) our technique could not identify a predicate instance whose switching caused the failure to be removed because the error is too complex for any predicate switch to produce correct output value. In versions 5 and 6 of *s-flex*) the search failed because the errors were in the data part of the computation. The number of instructions executed by the program following the execution of the critical predicate and before the program’s termination are given in column *After Fault*. This number is considerably smaller than the number in *Total Instrns*. This difference motivates the *LEFS* strategy. The next column, *Dep. Dist.*, is the shortest dependence distance between the output and the critical predicate in the dynamic dependence graph. As we can see, this distance is quite small and thus this provides the motivation for our *PRIOR* strategy. The remaining data in the table demonstrates the effectiveness of the two search strategies. The column labeled *Total Preds* is the total number of conditional branches that were executed during the program runs while the last two columns, *LEFS* and *PRIOR*, give the number of predicate instances that were actually explored by switching before finding a predicate instance whose switching produced the correct output (i.e., a critical predicate). As we can see, these numbers are considerably smaller than the numbers in *Total Preds*. In addition, the *PRIOR* number is far smaller than the *LEFS* number in most of the cases. In the case of *s-flex-v8*, although we found a critical predicate using the *LEFS* strategy, we could not do so using the *PRIOR* strategy. This is because we could not compute the dynamic slices on which the ordering of predicate instances is based. As explained later, the dynamic slices needed by *PRIOR* are for the erroneous output produced and the failure-inducing input difference. However, the program produced no output and the failure-inducing input difference could not be identified. Overall, the above data indicates that the more aggressive *PRIOR* strategy for ordering predicate instances is very effective.

Given the choices in search strategy described above, now we present our predicate switching algorithm. The overview of our algorithm is given in Figure 3. The algorithm has three major steps: finding the first erroneous value in a failing run; identifying the predicate instances which will be considered using predicate switching; and finally searching for a critical predicate reversal of whose outcome causes the program run to succeed. Let us consider the above steps in greater detail:

Step 1: Locate the first erroneous output value. A program run is considered to be a failing run if it produces incorrect output. Given the correct output, we determine the first deviation between the output produced by the failing run and the correct output and also identify the execution instance I_e of instruction I that produced the erroneous output value. The goal of our algorithm is to find a predicate instance switch that causes correct output value to be produced.

Step 1: Find Erroneous Value

Examine failed run to identify the first erroneous value
– erroneous output or value that crashes the program.

Step 2: Find Predicates for Switching

Run the program again for the following:
(a) Generate *Predicate Trace* (PT) identifying all instances of branch predicates executed and their execution order.
(b) Perform *Predicate Ordering* of predicates in PT using *LEFS* or *PRIOR*.

Step 3: Find Critical Predicate

```
for each pred. instance $P$ in ordered $PT$ do
  Generate instrumented program to switch $P$'s outcome;
  Execute above program; if program run succeeds,
    report $P$ and terminate search.
endfor
```

Figure 3: Algorithm Overview.

As mentioned earlier, if the program crashes at some execution instance I_e of instruction I, the value or address referenced by I_e that caused the program to crash takes the place of the erroneous output value. The goal of our algorithm in this situation is to find a predicate instance switch such that, following the switch, when execution instance I_e is encountered, the program does not crash.

Step 2: Identify predicate instances for switching. In this step we rerun the program and collect the *Predicate Trace* (PT). The predicate trace is a record of all instances of conditional branches executed during the failing program run from the start of the execution to the point at which the failing run produced the erroneous value identified in the preceding step (i.e., when I_e was executed). The program execution performed in this step not only generates the predicate trace, but in addition it also provides information using which we perform predicate instance ordering. If *LEFS* is used, the ordering is already clear from the predicate trace. If *PRIOR* is used we perform ordering as follows. We generate the dynamic dependence graph containing both dynamic data and control dependences during this run. Partitioning of predicates into high and low priority subsets is performed using a slicing based chopping algorithm that we presented in [3]. Here first we compute the backward dynamic slice (BS) of the erroneous output value. We also identify the failure inducing input using delta debugging technique [17] and compute the forward slice (FS) of the failure inducing input. The predicate instances that belong to the intersection of the forward and backward slice ($FS \cap BS$) form the subset that contains predicate instances that are expected to be influenced by faulty code as they were involved in producing the erroneous output. The remaining predicate instances form the lower priority subset. The predicate instances in the higher priority subset are further arranged in the order of increasing dependence distance from the erroneous output. The distances needed to perform this ordering are obtained from the dynamic dependence graph.

Step 3: Searching for a critical predicate. Figure 4 pictorially shows the search for a critical predicate when the simple *LEFS* strategy is used. The first line represents the failed run up to the point it produced the first erroneous value. The small ovals mark execution of predicate instances which are also labeled. Then the subsequent steps show how the predicate instances are switched one at a time in each subsequent run in the *LEFS* order. The predicate instance that is switched is marked using a larger oval. During each run the new output value is observed. The above process is repeated till correct output value is generated. The basic functioning of this step is the same when *PRIOR* strategy is used, only the order in which predicate instances are switched changes.

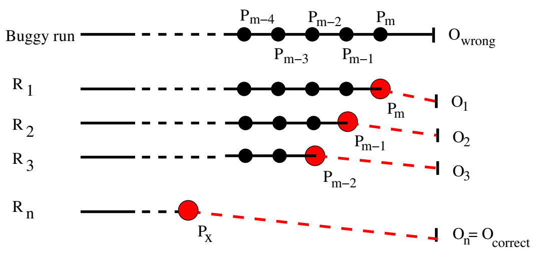


Figure 4: Search Method.

In the above algorithm we have assumed that once a predicate instance whose switching produces the correct output has been found, no further search is performed. In general it is possible that the predicate switch found is not meaningful and thus it does not enable us to understand the cause of the error. In such a situation, the search may be continued till another more useful predicate switch is identified. However, as our experimental results given later show, when *PRIOR* strategy is used, the first critical predicate found was a meaningful one except in one case.

There are a two important practical issues that must be considered during the above search. These issues are discussed next:
(1) *Correct output.* The first issue is that of determining if the predicate switch has generated the correct output. In case the program terminates producing output value, the value generated can simply be compared with the output value known to be correct. However, in case program crashes, as described earlier we are simply looking for a predicate instance switch that makes the cause of the crash to disappear. Determining that this has indeed happened requires some additional analysis. We know the instruction I, and its execution instance I_e, that caused the crash. When the program is being run with a predicate switch three cases can arise: program executes I_e and does not crash at I_e; program executes I_e and crashes again at I_e; and program does not execute I_e. In the first case we have found the desired predicate instance switch and therefore the search terminates. In the second case it is clear that the current predicate instance did not produce the desired result and hence we must continue the search. In the third case although the program did not crash, it is unclear whether the predicate instance switch really resolved the problem. This is because the predicate switch may have simply avoided the error by avoiding the execution of the instruction at which crash previously took place, but in reality the problem may still exist. Therefore in this case also we continue the search.
(2) *Infinite loops.* The second issue we consider is that sometimes a predicate switch may introduce an infinite loop.

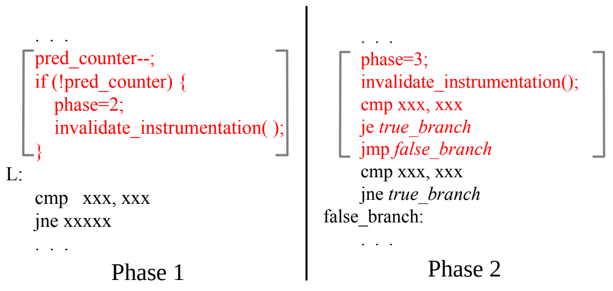
```
while (i!=1000) {
     ...
     i=i+1;
}
```

Consider the loop shown above. If the predicate instance in which the condition i!=1000 takes the $FALSE$ branch (i.e., when i is equal to 1000) is switched, the loop will continue to execute as i will take values of 1001, 1002, 1003, $\cdots$ and so on. This problem is handled by maintaining an internal basic block counter and if this counter exceeds a certain very large preset value, the execution is aborted and search is continued using the next predicate.

Finally we would like to mention that our technique does have its limitations. If the fault in the program is quite complex, our technique may fail to find a critical predicate due to following reasons. Overcoming the problems created by faulty code may require switching multiple predicate instances, i.e., simply switching one predicate instance at a time may not produce the desired result. If the fault is very significant, for instance some functionality is missing from the program, it is highly unlikely that the desired output can ever be generated by predicate switching. In this case simple modifications to program state may never yield the correct output.

4. DYNAMIC INSTRUMENTATION

We implemented our work within Valgrind [26], a well-known memory debugger and profiler for x86-linux binaries. Even though this tool works at binary level, the mapping back to source code level can be performed using the debugging information generated by the `gcc` compiler. Valgrind's kernel is a dynamic instrumenter which takes the binary and before executing any new (i.e., never instrumented) basic block, it calls the instrumentation function provided by us. The instrumentation function instruments the provided basic block and returns the new basic block to *Valgrind* kernel. The kernel executes the instrumented basic block instead of the original one. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The instrumentation is *dynamic* in the sense that we can enforce the expiration of any instrumented basic block such that the original basic block has to be instrumented again (in a different way).



```
    . . .
    pred_counter--;
    if (!pred_counter) {
       phase=2;
       invalidate_instrumentation( );
    }
L:
    cmp   xxx, xxx
    jne xxxxx
    . . .
```

```
      . . .
      phase=3;
      invalidate_instrumentation();
      cmp xxx, xxx
      je true_branch
      jmp false_branch
      cmp xxx, xxx
      jne true_branch
false_branch:
      . . .
```

Figure 5: Instrumentation and Phases.

To switch a particular predicate instance $(pred, inst)$ in a run R, we divide R into three phases. Each phase has its unique instrumentation. **Phase One** is from the beginning of the execution to the predicate instance of interest. In this phase, the program is instrumented in such a way that it surrenders the control to our framework when the execution reaches $(pred, inst)$. This is done by instrumenting a counter at $pred$ as shown in Figure 5. The counter is initialized to $inst$. Therefore, when it counts down to 0, it reaches the execution point of $(pred, inst - 1)$. Current instrumentation is invalidated such that *Valgrind* can re-instrument the predicate next time it sees the predicate and the execution enters the second phase. In **Phase Two**, as shown in Figure 5, the branch outcome of the predicate instance $(pred, inst)$ is reversed in this phase by switching the two branch targets. Once the instrumentation gets executed, it also invalidates itself to guarantee that the predicate is switched only once (i.e., future instances are not switched). After this instrumented predicate is executed once, *Valgrind* gains control and the execution enters the third phase. In **Phase Three**, *Valgrind* cleans up all the instrumentation and lets the program run to completion on its own without any interference.

5. EXPERIMENTAL RESULTS

5.1 Finding Critical Predicates

Table 4 shows how often our technique is successful in finding a critical predicate. As column *Found* shows, in 15 out of 20 cases we found a predicate instance switch which caused the program to produce correct output or eliminated the cause of the program crash. The critical predicate identified is indicated in columns *Where* and *Which*. Here column *Where* gives the file name and source line number at which the switched predicate can be found and *Which* is the dynamic instance of the predicate that was switched. The predicate instance number is measured from the point at which erroneous output is produced or program crashed. A value of 0 corresponds to the most recent execution instance of the predicate while greater values correspond to earlier instances of the predicate. As we can see, in many cases the most recent instance of a predicate is the critical instance while in some cases it is not the most recent instance. Finally, column *False +ves* represents the number of dynamic predicate switches, which produced correct output but were not related to the fault, that were found by *PRIOR* (except in case of *s-flex-v8* which uses *LEFS*) before the desired predicate switch was located. As we can see, in all cases except one, this number is 0 indicating that the first predicate switch located by *PRIOR* was related to the fault. In one case first predicate switch found was not useful but the second one found was meaningful.

Table 4: Successful/Failed Searches.

Program	Found	Where	Which	False +ves
flex 2.5.319(a)	yes	gen.c @ 1813	0	0
flex 2.5.319(b)	no	search failed		
flex 2.5.319(c)	no	search failed		
grep 2.5	yes	grep.c @ 532	0	0
grep 2.5.1 (a)	yes	search.c @ 549	0	0
grep 2.5.1 (b)	no	search failed		
grep 2.5.1 (c)	yes	dfa.c @ 2854	2	0
make 3.80 (a)	yes	read.c @ 6162	143	1
make 3.80 (b)	yes	remake.c @ 652	1	0
bc-1.06	yes	storage.c @ 176	9	0
tar-1.13.25	yes	prepargs.c @ 81	0	0
tidy	yes	parser.c @ 3496	0	0
s-flex-v4	yes	flex.c @ 2978	0	0
s-flex-v5	no	search failed – error in DP		
s-flex-v6	no	search failed – error in DP		
s-flex-v7	yes	flex.c @ 9171	0	0
s-flex-v8	yes	flex.c @ 11833	0	0
s-flex-v9	yes	flex.c @ 5046	0	0
s-flex-v10	yes	flex.c @ 2687	1	0
s-flex-v11	yes	flex.c @ 3559	0	0

We had shown earlier that *PRIOR* locates the desired predicate instance switch far sooner than *LEFS*. Now we measured the time taken by *PRIOR* to locate the desired predicate instance switch. The results are given in Table 5. As we can see, the time taken to locate critical predicates is quite reasonable. In many cases it is around 1 minute. The cases in which the search failed, the time is large (few hours) as it went through all the predicate instances.

5.2 Locating Faulty Code

After having found the critical predicate, the next step is to use this information in locating faulty code. One approach to this step is to simply require the user to manually examine the entire code

Table 5: Search time.

Program	PRIOR
flex 2.5.319(a)	2.51 sec
flex 2.5.319(b)	search failed (364 min)
flex 2.5.319(c)	search failed (274 min)
grep 2.5	8.83 sec
grep 2.5.1 (a)	2.59 sec
grep 2.5.1 (b)	search failed (4 min 28 sec)
grep 2.5.1 (c)	4.46 sec
make 3.80 (a)	26.92 sec
make 3.80 (b)	30 min 37 sec
bc-1.06	0.49 sec
tar-1.13.25	2.83 sec
tidy	0.90 sec
s-flex-v4	8.76 sec
s-flex-v5	search failed (96 min 20 sec)
s-flex-v6	search failed (3 min 56 sec)
s-flex-v7	3.34 sec
s-flex-v8	34.35 sec
s-flex-v9	34.51 sec
s-flex-v10	2.76 sec
s-flex-v11	2.56 sec

Table 6: Sizes of bidirectional slices and chops.

Program	EXEC	BiS (%EXEC)	FiChop (%EXEC)	BiChop (%EXEC)	Where
flex 2.5.319(a)	1871	225 (12.03%)	256 (13.68%)	27 (1.44%)	Pred.
flex 2.5.319(b)	2198	-	102 (4.64%)	102 (4.64%)	-
flex 2.5.319(c)	2053	-	5 (0.24%)	5 (0.24%)	-
grep 2.5	1157	88 (7.61%)	731 (63.18%)	86 (7.43%)	Down
grep 2.5.1 (a)	509	111 (21.81%)	32 (6.29%)	25 (4.91%)	Down
grep 2.5.1 (b)	1123	-	599 (53.34%)	599 (53.34%)	-
grep 2.5.1 (c)	1338	453 (33.86%)	12 (0.90%)	12 (0.90%)	Up
make 3.80 (a)	2277	1372 (60.25%)	739 (32.45%)	739 (32.45%)	Up
make 3.80 (b)	2740	1436 (52.41%)	1104 (40.29%)	1051 (38.36%)	Up
bc-1.06	636	267 (41.98%)	102 (16.03%)	102 (16.03%)	Up
tar-1.13.25	445	117 (26.29%)	103 (23.15%)	45 (10.11%)	Down
tidy	1519	541 (35.62%)	164 (10.80%)	161 (10.60%)	Up
s-flex-v4	1631	37 (2.27%)	7 (0.43%)	7 (0.43%)	Pred.
s-flex-v5	1882	-	544 (28.91%)	544 (28.91%)	-
s-flex-v6	424	-	156 (36.79%)	156 (36.79%)	-
s-flex-v7	2045	836 (40.88%)	63 (3.08%)	63 (3.08%)	Up
s-flex-v8	610	280 (45.90%)	-	280 (45.90%)	Pred.
s-flex-v9	1396	230 (16.48%)	112 (8.02%)	112 (8.02%)	Pred.
s-flex-v10	1683	640 (38.03%)	574 (34.11%)	574 (34.11%)	Miss
s-flex-v11	1749	27 (1.54%)	102 (5.83%)	27 (1.54%)	Up

to understand why the switching of a predicate instance caused the program to produce correct output. Another approach is to assist in this step by automatically narrowing the set of potentially faulty statements and then having the user examine these statements in conjunction with the critical predicate. We consider both these approaches next.

First we present an approach to locate *potentially faulty code* that is based on the use of dynamic slicing. One possible scenario is in which the critical predicate outcome was wrong due to incorrect values used in its computation. The faulty statements that produced the incorrect value(s) can be typically found in the *backward slice* of the critical predicate. Another scenario that arises is one in which the changing the critical predicate outcome avoids the program crash. In this case the *forward slice* of the critical predicate captures the code causing the crash. Given the above two scenarios, we conclude that to identify potentially faulty code we must compute the *bidirectional dynamic slice* of the critical predicate (i.e., the union of the backward and forward dynamic slices of the critical predicate as shown in Figure 6a).

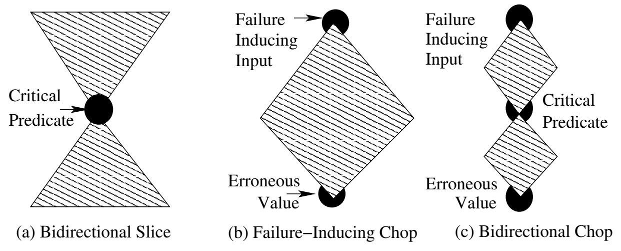


Figure 6: Bidirectional Dynamic Slice and Chop.

Consider the results in Table 6. *EXEC* is the number of distinct program statements that are executed at least once while the size of the subset of these statements that belong to the bidirectional dynamic slice of the critical predicate is given by *BiS*. We observe that *BiS* is significantly smaller than *EXEC*. In fact the size of *BiS* ranges from only 1.54% to 60.25% of the size of *EXEC*. In other words bidirectional slices are highly effective in reducing the scope of *potentially faulty code*. The faulty code was captured by the bidirectional slice in all cases except for `s-flex-v10`.

In prior work we introduced the notion of *failure-inducing chop* [3] which is obtained by intersecting the contents of the *backward slice* of an incorrect output value and the *forward slice* of the failure-inducing input difference. This is another approximation of potentially faulty code. The sizes of failure-inducing chops are given in column *FiChop* in Table 6. The *BiS* and *FiChop* can be used in conjuction – by interescting the two, the *bidirectional chop* (BiChop) as shown in Figure 6 is obtained. As the results in column *BiChop* in Table 6 show, the sizes of *BiChops* are very small for majority of the cases. In fact in 13 cases it is less than 16% of *EXEC*.

After having located the set of potentially faulty statements in form of *BiChop*, the next step for the user is to locate the actual cause of the fault. We consider a strategy for this task to further reduce the number of statements examined by the programmer before locating the fault. In Table 6, the column *Where* indicates the location of faulty code – the critical predicate (Pred.), the upward chop in *BiChop* (Up), or the downward chop in *BiChop* (Down). Of course, first we should look at the critical predicate itself. As we can see in 4 cases the fault was in the predicate itself. If the fault is not in the predicate, the following technique for ordering the statements in *BiChop* for examination by the programmer can be used. A strategy that orders the statements based upon their dynamic dependence distance from the critical predicate was considered. In other words, the strategy examines the statements that are closer to the critical predicate first. We found that this ordering is quite effective because even though the chops may contain many statements, if the above ordering is followed, the user encounters the faulty statement after examining only a few statements. The result of a study to demonstrate this point for *s-flex* versions are presented in Table 7. In the six versions we succeeded in finding critical predicates, to locate the faulty statement we only needed

Table 7: Dependence Distance Based Search.

Program	Statements	Dep. Distance
s-flex-v4	1	0
s-flex-v5	search failed – error in DP	
s-flex-v6	search failed – error in DP	
s-flex-v7	2	1
s-flex-v8	2	0
s-flex-v9	1	0
s-flex-v10	3	1
s-flex-v11	3	2

to examine 1 to 3 statements before finiding the faulty statement. The reason for this is that the faulty statement was at a very small dependence distance from the critical predicate.

Finally we also considered a simpler strategy that manually examines the statements in *BiS*. We found that in many cases even though *BiS* may be large, in practice, the location of the fault through manual examination was easy to perform. Given the predicate instance switch one may have to examine a small subset of statements identified above to locate the cause of erroneous behavior. Next we examine three of the real bugs to demonstrate this claim.

Flex It has been reported that XFree86 does not compile when $flex$-2.5.31 is installed. The reason for this problem is that variable $yy\_prev\_more\_offset$ is used but not defined in the lexical analyzer generated by $flex$. The bug is reproduced in the left column of Figure 7. Our technique identifies that switching one predicate instance in function $make\_tables(\ )$, as shown in the right column of Figure 7, produces the desired output. By looking at the code, it is apparent that it is not correct that the definition of *yy_prev_more_offset* should appear in the generated code only when *reentrant* is *TRUE*. *Reentrant* is *TRUE* when the option *–reentrant* is specified, which tells *flex* to produce a reentrant analyzer. Moving the statement at line 1816 out of the *else* branch fixes the bug.

Grep In *grep* version 2.51, if both the option $-F$ and $-w$ are specified, the result may not be correct. For instance, in the following case.

```
-bash-2.05>echo "test1 test test2" | grep -Fw test
-bash-2.05>
```

Option $-F$ prescribes that the pattern expression is used as a string to perform matching. Option $-w$ means searching for the pattern expression as a word. Obviously, empty output is not desired because the pattern string $"test"$ occurs in the input string as a word. Instead, the input string should be printed out as the result of a match.

We apply our technique on this buggy execution and find two different critical predicate instances. These two critical predicates (at lines 549 and 554) are shown in Figure 8. They are in *function Fexecute( )*, which is called when $-F$ is specified. From line 516 to line 548, *Fexecute( )* takes the input string and matches it to the pattern string. If a match exists, *beg* is the start of the matched substring and *len* contains length of the match. At line 549, if $-w$ is not specified, the program claims the matching is successful and then prints the input string; otherwise, it tries to decide whether the match just found is a word match in lines 550-570. If it is a word match, the program claims success and prints the input string. In the original buggy run, both the predicate instance at 549 and the predicate instance at 554 take the *TRUE* branch, which indicates $-w$ is specified and the first match is not a word match. The first match is the $"test"$ substring in $"test1"$. Therefore, *beg* equals to the beginning of the input string and *len* equals to 4. The *substring(beg, –len)* is $"tes"$, which fails to match to the pattern string at line 556. As a result, the procedure returns failed at line 563. Apparently, the procedure should not return failed at 563, it should break out of loop 550-570 and continue to the second match, which is the second $"test"$ substring in the input and is a word match. Replacing the return statement at 563 with a break statement successfully produces the desired output. Either of the two critical predicates point us right to where the bug is. They also provide information on how to fix the bug.

Make In make version 3.80, if backslashes are used for quoting or escaping colons in dependency names, it may create some problem. This error is manifested in the left column of Figure 9. We can see in the first run, *make -f input1.mk xyz:1*, the program correctly identifies the target named ”xyz\:1”. However, in the second run, *make -f input1.mk xyz*, it fails to find the target of ”xyz\:1”, which is the dependency of target ”xyz”. We find that switching either of the two predicates produces the correct output as marked in the right column with shaded rectangles. These two predicates are in function *find_char_unquote( )*, which parses a string stopping at char *stop1*, *stop2*, or *blank* if specified. It starts from the beginning of the string and searches for any stop char or char '\0', if the char right before the stop char is a backslash, which means the stop char is quoted and thus not a real stop sign, it removes that backslash and continues. We further investigate the two predicates and find that they are in a call by *parse_file_seq( )*, which generates the list of names for target dependencies. The stop char is defined as '\0'. When parsing the target names, the stop char is specified as ':' in a similar call to *find_char_unquote( )*. Because the stop char is '\0', *find_char_unquote( )* keeps increasing p at 2164 till the end of string when parsing ”xyz\:1” which is the dependency for target ”xyz”. These two predicates correspond to terminating the loop in $2162 - 2164$ at char ':' before reaching the end. Because now *p[-1]=='\'* at 2172, the \ is removed from the string. Finally string ”xyz:1” is returned as the dependency name. Because char ':' is specified as the stop char when parsing the target names, target ”xyz\:1” has the internal name of ”xyz:1” as well. Therefore, the program is able to find a match between the dependency and the target such that it generates the correct output. Note that reversing predicate **p!='\0'* at 2162 does not have the same effect because the compiler optimization combines this predicate with the one at 2169 such that the generated code directly breaks out of the outer while loop.

Except for *s-flex*, in all other cases the bugs are logical errors which require changes to the code that are not localized to a specific faulty statement. The above study shows that logical errors can be understood by studying the statements in the chops and the critical predicate. However, as is the case in *s-flex*, there can also be faults in a program which are contained in a specific statement such that fixing that statement fixes the program.

6. RELATED WORK

Dynamic slicing was introduced as a aid to debugging [9, 1]. Our recent works [19, 20] have greatly reduced the space and time cost of dynamic slicing. In [21], we evaluated the effectiveness of *backward dynamic slices* in fault location. Our result showed that even though dynamic slices can capture the faulty code, identifying the faulty code from the set of statements in the slice still requires nontrivial human effort. We further narrowed the scope of potentially faulty code in [3] by, for the first time, using *forward dynamic slices* of failure-inducing input difference. In contrast, in this paper, we have shown that *bidirectional dynamic slices* of critical predicates can further narrow the search for faulty code. The computation of *Bidirectional Chop* is based upon identifying multiple kinds of *negative evidence*, i.e. program entities related to execution of faulty code. In recent work we have demonstrated the use of *postive evidence* in form of correct portions of the outputs produced during a failing run to order and prune statements in the potentially faulty code [22].

In a series of articles [17, 16, 15], the *delta debugging* algorithm has been developed to automatically simplify or isolate a failure-inducing input [17, 16], produce cause effect chains [15] and to link cause transitions [2] to the faulty code. In [2] delta debugging algorithm is used to analyze *program state changes* during the execution of a failed run to identify points of *cause transitions*. Code

```
-bash-2.05>cat input
\%{
\%}
\%\%
-bash-2.05>flex -l -t -Cae input
...
(yy_prev_more_offset is not defined)
...
-bash-2.05>

The CORRECT output is,

-bash-2.05>flex -l -t -Cae input
...
static int yy_prev_more_offset = 0;
...
-bash-2.05>
```

```
void make_tables ()
{
    . . .
        if (!reentrant){
                    indent_puts ("static int yy_more_offset  = 0;");
        } else{
            indent_puts ("static int yy_prev_more_offset = 0;");
        }
   . . .
                                   gen.c
```

Figure 7: Bug in flex-2.5.31

```
Fexecute (char const *buf, size_t size, size_t *match_size, int exact)
{ . . .
  for (beg = buf; beg <= buf + size; ++beg)
    {  /*match the substring (beg, bug+size-buf) to any of the keywords, len contains the length of matched;*/
      . . .
     else if (match_words)    /*if -w is specified*/
       for (try = beg; len; )
         {   . . .
           if (. . .)             /*if the matched substring is not a word*/
             {
               offset = . . .;   /*match substring(try, --len) to any of the keywords, result is put in offset*/
               if (offset == (size_t) -1)            /*there is no match*/
                 {  . . .
                   return offset;
                 }
                 . . .                       /*update try, len to the head and the length of the matched substring*/
             }
           else
             goto success;
         }
       else
         goto success;
    }
```

Figure 8: Bug in search.c of grep 2.51.

```
-bash-2.05>cat input1.mk
xyz: xyz\:1
@echo $@: $<

xyz\:1: input1.mk
@echo $@: $<

.PHONY: none

-bash-2.05>make -f input1.mk xyz:1
xyz:1: input1.mk
-bash-2.05>make -f input1.mk xyz
make: *** No rule to make target ‘xyz\:1’,
needed by ‘xyz’.  Stop.
-bash-2.05>

The CORRECT output is

-bash-2.05>make -f input1.mk xyz
xyz:1: input1.mk
xyz: xyz:1
```

```
char *
find_char_unquote (char * string, int stop1, int stop2, int blank)
{
  register char *p = string;
  while (1)
  {  . . .
    else if (blank)
      while (*p != '\0' && *p != stop1
           && ! isblank ((unsigned char) *p))
        ++p;
     . . .
    if (*p == '\0')
        break;
    if (p > string && p[-1] == '\\')
      {
        /*remove  the backslash from the the string*/
      }
     . . .
  }                              read.c
```

Figure 9: Bug in make3.80

executed at the points of cause transitions is expected to be relevant to the fault. Comparing and changing memory states of C program executions at a point is difficult due to pointers [2]. In addition, to identify points of cause transitions, the above state-based analysis has to be performed at a large number of points along the failed run. Therefore, program state based analysis is difficult and time consuming for C programs [2]. In comparison our approach is inexpensive in terms of time taken.

A number of statistical approaches that analyze program spectra of program runs for multiple inputs, including inputs corresponding to both failed and successful runs, are being employed for fault location. Harrold et al. [5] compared the spectra of passing and failing runs and found that failing runs tend to have unusual coverage spectra. Jones et al. [8] ranked each statement according to its ratio of failing tests to correct tests and used this information to assist fault location. Liblit et al. [10] describe a sampling framework and present an approach to guess and eliminate predicates to isolate a deterministic bug. For isolating nondeterministic bugs, they use statistical regression techniques to identify predicates that are highly correlated with the program failure. In contrast, Renieris and Reiss [12] focused on the difference between the failing run and a *single* passing run with similar spectra as a means to narrow down the search space for faulty code. Our work is complementary to the above work as it focusses on a failed run corresponding to single input for fault location. However, one advantage of our approach is that it provides dependence relationships between various points of interest, i.e. failure-inducing input, critical predicate, and erroneous output. This information is useful to the programmer during debugging.

Some additional works include the following. Xie et al. show that many redundancies [14] in programs correspond to hard program errors. Hangal et al. [4] identified the causes of some programming errors in Java programs by observing violations of program invariants. In [6], we developed a technique that used a notion of path based weakest preconditions to automatically locate faulty code in a function when the precondition and postcondition of the function are available as first order predicate logic formulas.

7. CONCLUSIONS

In this paper we presented the idea of *critical predicates* and presented an efficient automated algorithm for locating critical predicates. A critical predicate is an instance of a conditional branch such that if the outcome of this instance is switched, the failing run changes to a successful run either by causing correct output to be generated instead of incorrect output or by causing the crash that previously occurred not to happen. We show that not only can critical predicates be very often located in many real reported faulty programs, they provide valuable clues to the cause of the failure and hence assist in fault location. We also demonstrated how critical predicates when coupled with dynamic slicing can reduce the effort for fault location.

Acknowledgements

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

8. REFERENCES

[1] H. Agrawal and J. Horgan, “Dynamic Program Slicing,” *SIGPLAN Conference on Programming Language Design and Implementation*, pages 246-256, 1990.

[2] H. Cleve and A. Zeller, “Locating Causes of Program Failures,” *27th International Conf. on Software Engineering*, pages 342-351, 2005.

[3] N. Gupta, H. He, X. Zhang, and R. Gupta, “Locating Faulty Code Using Failure-Inducing Chops,” *IEEE/ACM International Conf. on Automated Software Engineering*, Long Beach, CA, Nov. 2005.

[4] S. Hangal and M.S.Lam, “Tracking Down Software Bugs Using Automatic Anomaly Detection,” *International Conference on Software Engineering*, 2002.

[5] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, “An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults,” *Journal of Software Testing Verification and Reliability*, 10(3):171-194, 2000.

[6] H. He and N. Gupta, “Automated Debugging using Path-Based Weakest Preconditions,” *Fundamental Approaches to Software Engineering*, Barcelona, Spain, 2004.

[7] R. Hildebrandt and A. Zeller, “Simplifying Failure-inducing Input,” *International Symposium on Software Testing and Analysis*, pages 135-145, 2000.

[8] J.A. Jones, ”Fault Localization Using Visualization of Test Information”, *26th International Conference on Software Engineering*, page 54-56,2004.

[9] B. Korel and J. Laski, “Dynamic program slicing,” *Information Processing Letters*, (29)3:155-163, 1988.

[10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug Isolation via Remote Program Sampling,” *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[11] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss, “Automated Fault Localization Using Potential Invariants,” *Fifth Int. Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, Sept. 2003.

[12] M. Renieris and S. Reiss, “Fault Localization with Nearest Neighbor Queries,” *Automated Software Engineering*, 2003.

[13] M. Weiser, “Program Slicing,” *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, pages 352-357, 1982.

[14] Y. Xie and D. Engler, “Using Redundancies to Find Errors,” *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 51-60, 2002.

[15] A. Zeller, “Isolating Cause-effect Chains from Computer Programs,” *SIGSOFT Symposium on Foundations of Software Engineering*, Charleston, South Carolina, US, 2002.

[16] A. Zeller, “Yesterday, my program worked. Today, it does not. Why?,” *7th European Software Engineering Conference/ 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 253-267, Sept. 1999.

[17] A. Zeller and R. Hildebrandt, “Simplifying and Isolating Failure-inducing Input,” *IEEE Transactions on Software Engineering*, Vol 28, No 2, Feb. 2002.

[18] X. Zhang, R. Gupta, and Y. Zhang, “Precise Dynamic Slicing Algorithms,” *IEEE International Conference on Software Engineering*, pages 319-329, Portland, Oregon, May 2003.

[19] X. Zhang, R. Gupta, and Y. Zhang, “Effective Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams,” *IEEE International Conference on Software Engineering*, pages 502-511, May 2004.

[20] X. Zhang and R. Gupta, “Cost Effective Dynamic Program Slicing,” *SIGPLAN Conference on Programming Language Design and Implementation*, pages 94-106, June 2004.

[21] X. Zhang, H. He, N. Gupta and R. Gupta, “Experimental evaluation of using dynamic slices for fault location,” *Sixth International Symposium on Automated and Analysis-Driven Debugging*, Monterey, California, September 2005.

[22] X. Zhang, N. Gupta, and R. Gupta, “Pruning Dynamic Slices With Confidence,” *SIGPLAN Conference on Programming Language Design and Implementation*, to appear, June 2006.

[23] P. Zhou, W. Liu, f. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrelas, “Accmon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants,” *International Symposium on Microarchitecture*, pages 269-280, Nov. 2004.

[24] *http://www.cse.unl.edu/∼galileo/sir*

[25] *http://www.elis.ugent.be/diablo/*

[26] *http://valgrind.org/*