Locating Faulty Code Using Failure-Inducing Chops[*]

Neelam Gupta Haifeng He Xiangyu Zhang Rajiv Gupta

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
{ngupta, hehf, xyzhang, gupta}@cs.arizona.edu

ABSTRACT

Software debugging is the process of locating and correcting faulty code. Prior techniques to locate faulty code either use program analysis techniques such as backward dynamic program slicing or exclusively use delta debugging to analyze the state changes during program execution. In this paper, we present a new approach that integrates the potential of delta debugging algorithm with the benefit of forward and backward dynamic program slicing to narrow down the search for faulty code. Our approach is to use delta debugging algorithm to identify a minimal failure-inducing input, use this input to compute a forward dynamic slice and then intersect the statements in this forward dynamic slice with the statements in the backward dynamic slice of the erroneous output to compute a failure-inducing chop. We implemented our technique and conducted experiments with faulty versions of several programs from the Siemens suite to evaluate our technique. Our experiments show that failure-inducing chops can greatly reduce the size of search space compared to the dynamic slices without significantly compromising the capability to locate the faulty code. We also applied our technique to several programs with known memory related bugs such as buffer overflow bugs. The failure-inducing chop in several of these cases contained only 2 to 4 statements which included the code causing memory corruption.

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, forward dynamic program slicing, backward dynamic program slicing, failure-inducing input

1. INTRODUCTION

As Mark Paulk from Carnegie Mellon's University's Software Engineering Institute noted,"*A fundamental problem with software quality is that programmers make mistakes*" [15]. Programming being a primarily human activity, errors creep into software in spite of the advances made in the areas of programming languages and software development processes. Locating and correcting errors in software is a difficult and time consuming activity that requires understanding of the software. Techniques and tools that can help software developers in narrowing down the search for faulty code can greatly reduce the time and resources spent on locating and correcting software errors.

Delta Debugging. Zeller introduced the term *delta debugging* [27] 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. Zeller and Hildebrandt [28] then applied the delta debugging approach to simplify and isolate the failure-inducing input. To *simplify* a failing test case, the delta debugging algorithm finds a minimal test case where removing any single input entity would cause the failure to disappear. For *isolating* a minimal failure-inducing input difference between a failing and a passing test case, the generalized delta debugging algorithm can be used. In [26], Zeller further extended this idea to isolate failure-inducing differences in program states and build a cause-effect chain in terms of relevant state differences. However, as we show later with an example, it may not always be easy to link the cause-effect chains (which are in terms of values of variables at different execution points) to the faulty source code. Therefore, it is still up to the programmer to use these cause-effect chains in terms of program states to decide where the failure causing code could be or what the failure causing circumstance could be. Recently, Cleve and Zeller [7] focus on cause transitions in an effort to link the failures to the program code. The focus of this work is on analyzing *program state* transitions in space and time in order to narrow down the search for faulty code.

Dynamic Slicing. In contrast to the above techniques that analyze *program states* as they occur during program execution, the traditional program analysis techniques such as program slicing [1, 2, 3, 4, 6, 9, 17, 18, 24, 29, 30, 31, 32] focus on *source code* analysis of the programs. The concept of program slicing was first introduced by Mark Weiser [24]. Since debugging is usually performed by analyzing the statements of the program when it is executed using a specific input, Korel and Laski proposed the idea of *dynamic program slicing* [17]. There are two kinds of dynamic slices: backward dynamic slices and forward dynamic slices. The *backward dynamic slice* of a variable at a point in the execution trace includes all those

[*] Supported by grants from IBM, Microsoft, Intel, and NSF grants CCR-0324969, CCR-0220262, CCR-0208756, and EIA-0080123 to the Univ. of Arizona.

ASE'05, November 7–11, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-993-4/05/0011 ...$5.00.

```
1  main(int argc, char *argv[ ])
2  {
3     int red, green, blue, yellow;
4     int sweet,sour,salty,bitter;
5     int i;
6
7     red = atoi (argv[1]);
8     blue = atoi (argv[2]);
9     green = atoi (argv[3]);
10    yellow = atoi (argv[4]);
11
12    red = 2*red;        Error: red = 5*red
13    sweet = red*green;
14    sour = 0;
15    i = 0;
16    while ( i < red) {
17          sour = sour + green;
18          i = i + 1;
19     }
20     salty = blue + yellow;
21     yellow = sour + 1;
22     bitter = yellow + green;
23
24    printf ("%d %d %d %d\n", bitter,sweet,
25               sour,salty);
26    return 0;
27  }
```

Initial Inputs: *input1* : = [1,5,8,2] - ×
input2 : = [0,0,0,0] - √

1-minimal failure-inducing inputs: *input3* : = [1,0,8,2] - ×
input4 : = [0,0,8,2] - √

Incorrect outputs at line 24: *bitter, sweet, sour*

FwdSlice$(input3, argv[1]) = \{7, 12, 13, 16, 17, 18, 21, 22, 24\}$

BwdSlice$(input3, bitter@24) = \{7, 9, 12, 14, 15, 16, 17, 18, 21, 22, 24\}$
Failure-inducing Chop$(input3, argv[1], bitter@24) = \{7, 12, 16, 17, 18, 21, 22, 24\}$

BwdSlice$(input3, sweet@24) = \{7, 9, 12, 13, 24\}$
Failure-inducing Chop$(input3, argv[1], sweet@24)=\{7, 12, 13, 24\}$

BwdSlice$(input3, sour@24)= \{7, 9, 12, 14, 15, 16, 17, 18, 24\}$
Failure-inducing Chop$(input3, argv[1], sour@24)= \{7, 12, 16, 17, 18, 24\}$

Figure 1: An example program.

executed statements which affect the value of the variable at that point. In contrast, the *forward dynamic slice* of a variable at a point in the execution trace includes all those executed statements that are affected by the value of the variable at that point. Backward dynamic slicing has been proposed to guide programmers in the process of debugging [2, 4, 9, 18, 31] by focusing the attention of the user on a subset of program statements which are expected to contain the faulty code. The effectiveness of backward dynamic slice in fault location is determined by two factors: *How often is the faulty statement present in the slice?* and *How big is the slice, i.e. how many statements are included in the slice?* In our previous work [31], we have evaluated and compared the effectiveness of different backward dynamic slicing algorithms in fault location. For those erroneous statements that are present in the static slices of the faulty outputs, we observed that dynamic slices are able to contain the faulty statement in most of the cases and in general dynamic slices are quite small compared to the number of executed statements. In fact results of a study reported in [32] show that the number of executed statements can range from 2.46 to 56.08 times the number of statements in a backward dynamic slice. However, we also observed that the number of statements in a backward dynamic slice could still be large and in addition many of the statements are apparently unlikely to be related to the fault. The goal of this paper is to further reduce the number of statements that need to be examined to locate faulty code.

Integrating Delta Debugging and Dynamic Slicing. Surprisingly, none of the prior research on delta debugging [27, 28, 26, 7] integrates the potential of dynamic program slicing with the delta debugging approach in narrowing down the search for faulty code. In this paper, we propose the following novel approach. First we use delta debugging to either find a simplified failure-inducing input or isolate a minimal failure-inducing input difference. We simply refer to this relevant part of the input as the minimal failure-inducing input. Next, we compute the intersection of the statements in the *forward dynamic slice* of the failure-inducing input and the *backward dynamic slice* of the faulty output to locate the likely faulty code. We simply refer to the statements in the above intersection of forward and backward dynamic slices as the *failure-inducing chop*. The failure-inducing chops are expected to be much smaller than backward dynamic slices since they capture only those statements of the dynamic slices that are affected by the minimal failure-inducing input.

Let us consider the example program shown in the left column of Figure 1 which is taken from the Unravel tool set [22]. Let us introduce an error in this program by modifying the statement at line 12 to *red* $= 5 *$ *red*. Inputs for a failed test case and a passed test case for this program are shown on the right. Starting with these inputs for the failed and passed runs, we use the delta debugging algorithm to isolate minimal failure-inducing input difference. The algorithm isolated *argv[1]* as the minimal failure-inducing input difference. The inputs for the failed test case and the correct test case corresponding to the isolated minimal failure-inducing input difference are $[1, 0, 8, 2]$ and $[0, 0, 8, 2]$. The three outputs *bitter, sweet* and *sour* are found to have wrong value for input $[1, 0, 8, 2]$. The forward slice on *argv[1]* and the backward slices on *bitter, sweet* and *sour*, and the resulting failure-inducing chops are also shown in the right column in Figure 1. We can see that the failure-inducing chop for each of the faulty outputs contains the faulty statement 12. In addition, the failure-inducing chop in each case is smaller than the respective backward and forward slices.

Note that in this example, both the failed run as well as successful run contain the error statement. Hence an approach based on program *dices* (i.e., set difference of execution slices of failed and successful runs) discussed in [4] would not be able to locate the error statement in this case. We also used AskIgor [5], the automated debugging service based on the delta debugging technique, to obtain the cause-effect chains for the example program in Figure 1 with line 12 in error. The cause-effect chains are given below.

1. Execution reaches line 14 of test_dd.c in main. Since the program was invoked as `"test_dd 1 5 8 2"`, local variable `sweet` is now `40`.
2. Execution reaches line 21 of test_dd.c in main. Since `sweet` was `40`, local variable `salty` is now `7`.
3. Execution reaches line 22 of test_dd.c in main. Since `salty` was `7`, local variable `salty` is now `7`.
4. Execution ends. Since `salty` was `7`, the program exits with status code 255. The program fails.

From the above cause-effect chains, it is not easy to figure out that the statement at line 12 in the source code was in error. Also, the values of the variable `salty` at steps 2, 3, and 4 in the cause-effect chain do not seem to be directly linked with the failure. In fact, the output `salty` was correct for both the failed and successful runs. We would like to point out here that although delta debugging provides a novel technique to select a set of test cases that may be highly relevant to the failure, it may not always be easy to link the cause-effect chains in term of values of variables at different execution points to the faulty source code.

Our approach attempts to combine the novelty of delta debugging algorithm in selecting relevant test cases, with the benefits of dynamic source code analysis, to automatically provide information to the programmer about which statements are most likely to be cause of failure. Note that the cause-effect chains may also be able to capture failures resulting from other circumstances (such as incorrect environmental settings etc.) besides the failures resulting from the faulty source code. However, the goal of our work is to be able to provide more direct source code related help to the programmers in the cases where failures of programs result from faulty source code. We implemented our approach and evaluated its effectiveness in locating faulty code for several programs [20, 33, 14, 13]. Our results show the promise of combining dynamic slicing with delta debugging for locating faulty code. The contributions of our paper are:

- We propose a novel approach that integrates the delta debugging algorithm with the forward and backward dynamic program slicing to narrow down the search for faulty statements to a *failure-inducing chop*.
- Prior work on using program slicing for debugging has proposed the use of *backward* dynamic program slicing. However, we show how delta debugging enables the use of *forward* program slicing in locating faulty code.
- We conducted experiments with the Siemens suite of programs [14, 13] which provides a few faulty versions injected with faults and the test pools for each program. Our results show that on average, the faulty statement was present in the failure-inducing chop in 43% to 100% cases across different programs. Also, the average size of failure-inducing chop is 64% to 73% of the size of backward dynamic slice and only 7% to 14% of the size of the whole program.
- We applied our technique to several programs with known memory related bugs. In many cases our technique was able to locate very few (2 to 4) statements which included the faulty code.

The rest of the paper is organized as follows. In section 2 we explain our technique and present the algorithms. We describe our implementation in section 3 and present experimental results in section 4. Related work is presented in section 5 and conclusions are given in section 6.

2. FAILURE-INDUCING CHOPS

The basic idea of our approach is shown in Figure 2. Figure 2(a) represents a successful execution which takes a set of inputs, carries out a series of computations and then produces a set of outputs. Figure 2(b) represents a failing run (faulty output is observed) resulting from the execution of a *faulty statement*. The figure shows a case in which the backward dynamic slice *BS* on the faulty output includes the faulty statement. By using delta debugging algorithm [28], we identify the *minimal failure-inducing input* Δ_{min} as shown in the figure. Therefore, if we perform forward dynamic slicing on the Δ_{min}, it is reasonable to expect that the *faulty statement* would be present in the failure-inducing chop *(FChop)* which is the intersection of *BS* and *FS*. As shown in Figure 2, we can expect the *(FChop)* to be much smaller than either *BS* or *FS*. The outline of our algorithm is given in Algorithm 1. Next we discuss each step of our algorithm.

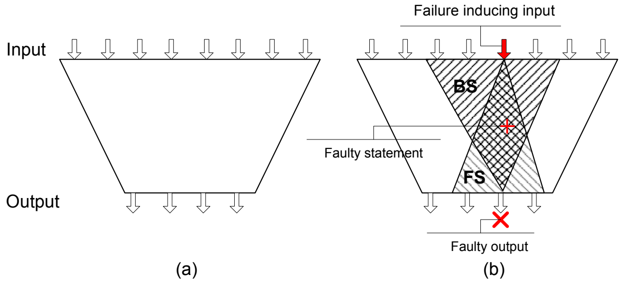


Figure 2: Failure-inducing chops.

Algorithm 1 *Fault location using failure-inducing chops.*

```
Step 1: Compute minimal failure-inducing input by:
    either use $ddmin$ to Simplify input [28]:
            $I'_f = ddmin(I_f)$
            $\Delta_{min} = I'_f$
    or use $dd$ to Isolate input difference [28]:
            $(I'_s, I'_f) = dd(I_s, I_f)$
            $\Delta_{min} = I'_s - I'_f$
Step 2: Compute forward dynamic slice:
        $FS = FwdSlice(I'_f, \Delta_{min})$
Step 3: Compute backward dynamic slice:
        $BS = BwdSlice(I'_f, failed\_output)$
Step 4: Compute failure-inducing chop:
        $FChop = FS \cap BS$
```

Step 1: Finding minimal failure-inducing input. To find a failure-inducing input, any of the two algorithms given by Zeller and Hildebrandt in [28] can be used. The first algorithm (ddmin) *simplifies* a failing test case I_f to produce a minimal test case I'_f such that removing any single input entity from I'_f causes the failure to disappear. Therefore Δ_{min} is I'_f in this case. The second algorithm (dd) *isolates* a minimal failure-inducing input difference between a failing and a passing test case. Given inputs I_f and I_s for a failed run and a successful run respectively, this algorithm returns a pair of inputs (I'_f, I'_s), such that I'_s and I'_f correspond to a successful run and a failed run respectively and any single part of $I'_f - I'_s$ if removed from I'_f would make the failure disappear or if added to I'_s would make the failure occur. Therefore in this case Δ_{min}=I'_f-I'_s.

Step 2: Compute Forward Dynamic Slice. The minimal failure-inducing input Δ_{min} computed by the first step defines the slicing criteria for the forward dynamic slicing. In this step, we compute the forward dynamic slice *FS* = $FwdSlice(I'_f, \Delta_{min})$. We illustrate the computation of forward dynamic slice using the example in Figure 1. In Figure 1, the value of *red* at statement 12 is affected by input *argv*$[1]$ because it is data dependent on *argv*$[1]$. Therefore statement 12 is in the forward slice of *argv*$[1]$. Even though *argv*$[1]$ does not directly contribute to the values computed at statements 17 and 18, it decides the execution of those statements by affecting the predicate outcome at 16. Therefore statements 17 and 18 are also in the forward slice of *argv*$[1]$. In other words, given an input and the corresponding execution, the dynamic *forward slice* is the set of statements which are affected by that particular input via data/control dependences.

While a statement can be statically control dependent upon multiple predicates, at runtime, each execution instance of a statement is dynamically control dependent upon a single predicate. The predicate on which the execution of a statement is control dependent is found as follows. First, a statement execution s and its intraprocedural control ancestor p must correspond to the same function invocation. Second, the dynamic control dependence of execution of s is on the most recently executed predicate p on which s is statically control dependent. Timestamps are associated with execution instances of statements in order to evaluate the above condition. Third, inter-procedural control dependence is computed by introducing extra dependence edges between call sites and the corresponding function entries.

Algorithm 2 *Updating forward slicing information.*

```
Procedure Update($s_i$, stack)
 1: IsMarked = 0;
 2: for (each use $v$ in Use[$s_i$]) do
 3:     IsMarked = IsMarked $|$ MARKED[$v$];
 4: end for
 5: $cd$ = the predicate in CD(s) s.t. stack.ts[cd] is maximum;
 6: IsMarked = IsMarked $|$ stack.marked[cd];
 7: if (IsMarked $==$ 1) then
 8:     slice= slice $\cup \{s\}$
 9: end if
10: if ($s$ is a predicate) then
11:     stack.marked[$s$] $=$ IsMarked;
12:     $stack.ts[s]$=$timestamp$++;
13: end if
14: for (each definition $v$ in Def[$s_i$]) do
15:     MARKED[$v$] = $IsMarked$;
16: end for
```

Next we present a forward computation algorithm [6, 30] for forward slicing. This algorithm updates the forward slices of variables after execution of each statement. The updating of dynamic forward slice following the execution of statement instance s_i is presented in the Algorithm 2. The variables used in the algorithm are as follows: *MARKED*$[v]$ denotes whether v is affected by the specified input; *IsMarked* indicates whether or not statement s is to be included in the slice; *slice* denotes the currently computed forward slice for the specified input; $timestamp$ denotes the current time; and $stack$ is the current stack frame. We store the timestamps of latest executions of predicates in *stack.ts[]* and the information about whether these predicates were affected by the specified input in *stack.marked[]*. This guarantees that when we search for the predicate instance with largest timestamp in the set $CD(s)$ of predicates on which s is statically control dependent, we only consider those instances that have the same stack frame as s_i. In the Algorithm 2, lines 2 to 4 set *IsMarked* if there is a use of a marked variable v in s_i. Lines 5 and 6 set *IsMarked* if immediate control ancestor of s_i is marked. If *IsMarked* is set, lines 7 to 9 include s in the forward slice. Lines 10 to 13 update the information on the stack if s is a predicate. Finally, lines 14 to 16 mark each of the variable defined by s_i. To initiate the propagation of marks, we need to mark the specified input variable.

Step 3: Compute the backward dynamic slice. In this step the backward dynamic slice *BS* = $BwdSlice(I'_f, failed\_output)$ for the failed output is computed for the failed run corresponding to the input I'_f generated in Step 1. The backward dynamic slice is computed from the statement instance where the first erroneous output is generated. When the faulty version generates a segmentation fault for I'_f, the backward slicing criterion is the pointer which caused the segment fault instead of the output. The backward dynamic slicing algorithm we used is the *full slicing* algorithm presented in our prior work [31].

Step 4: Compute the failure-inducing chop. In this step, we compute the statements in the intersection of the forward dynamic slice *FS* computed in Step 2 and the backward dynamic slice *BS* computed in the Step 3. Our algorithm identifies the set of statements in this intersection called the failure-inducing chop *FChop* as the set of statements that are likely to contain the faulty code.

3. IMPLEMENTATION

We have developed a dynamic slicing tool which was used to conduct experiments. Our tool executes `gcc` compiler generated binaries for Intel x86 and computes dynamic slices based upon forward computation algorithms. Even though our tool works at binary level, the slices can be mapped back to source code level using the debugging information generated by `gcc`.

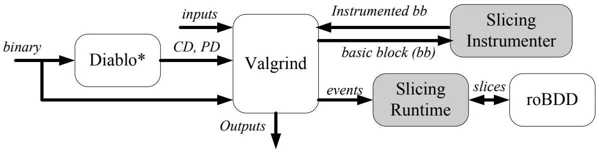


Figure 3: Slicing infrastructure.

Fig. 3 shows the main components of the tool. The *static analysis* component of our tool computes static control dependence (CD) required for forward/backward slice computations from the binary. The static analysis was implemented using the *Diablo* [8] 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 upon the *Valgrind* memory debugger and profiler [23] accepts the same `gcc` generated binary, 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 which 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 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 the user can also enforce the expiration of any instrumented basic block. Thus, we can easily turn off/on the slicing instrumentation for sake of time performance or for certain code, e.g. library code.

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.). We intercept any output system call (_*WRITE etc.*) and then augment the original output with their slices represented by *Reduced ordered Binary Decision Diagram*s (roBDD)s [30]. More details about why and how we use roBDD in slicing can be found in our previous work [30, 31]. One of the very important features of roBDD is that it can represent a unique set by one unique (integer) number and from that number the full set can be easily retrieved from the roBDD. In other words, by using roBDD we are able to represent a slice by one integer. This is critical to our design because now for each variable (memory location) we only need to store one integer.

The basic idea of forward computation is that when some operation is performed on operands, the slices/marks of source operands are fetched and unioned/ored together with the current statement. The resulting slice is assigned to the destination operand. Since one slice can be represented as one integer, we need to store one integer (one mark bit) for each operand which could be memory location, register, or predicate.

The implementation of delta debugging was carried out separately as it mainly involves repeatedly manipulating program inputs and executing the program on these inputs. Once the failure-inducing input has been identified, the above set up is used to execute the program and compute the chop from the forward and backward dynamic slices.

4. EXPERIMENTAL EVALUATION

4.1 Evaluation for Siemens Suite

Our first experimental study is based upon the programs from the Siemens suite [14, 13]. For each program, the Siemens suite provides its test cases and several faulty versions with manually injected faults.

Table 1: Overview of benchmark programs.

Program	Description	Versions	LOC	Tests
`print_tokens`	lexical analyzer	5	565	4072
`print_tokens2`	lexical analyzer	7	510	4057
`schedule`	priority scheduler	6	412	2627
`schedule2`	priority scheduler	2	307	2683
`replace`	pattern replacement	18	563	5542

Table 1 shows the Siemens suite programs used in our experimentation. We excluded the programs *tcas* and *tot_info* because *tcas* is too small and *tot_info* has floating point operations, which are currently not supported by our slicing tool. Since the omitted statement will not be present in any static or dynamic slice, we excluded the faulty versions corresponding to errors of code omission from our experiments. Each faulty version of the program used in our experiments had exactly one fault injected. Some faulty programs were excluded because they produce no output and thus it is unclear how a proper slicing criterion should be defined. We instrument the faulty programs in minor ways in order to run the experiments. For example, we replaced the input functions like *fgets, fgetc*, and output functions *fprintf, printf, fputc* with our customized functions using which it is more convenient for our slicing tool to find the forward and backward slicing criterion.

Failure-inducing chops. Table 2 lists the programs and their faulty versions used in our experiments. We used the *ddmin* input simplification algorithm [28] in these experiments. The number of failed test cases in the test pool, for each faulty version of each program used in our experiments, are shown in column *FTs*. When we applied the *ddmin* input simplification algorithm to each of these failed test cases, the number of unique simplified inputs produced for each faulty version are given in column *USIs*. The latter number is smaller than the former because in some cases different failing inputs produce the same simplified input. The average sizes of inputs for failed test cases and the average sizes of simplified inputs are given in columns *Orig.* and *Simplified* respectively. The input sizes are in terms of entities appropriate for the program. For *print_tokens* and *print_tokens2* it is the number of tokens, for *schedule* and *schedule2* it is the number of commands, and for *replace* it is the number of characters. As we can see, simplified inputs are much smaller than original inputs.

Table 2: Number and size of original and simplified inputs.

Program	Version	Number of		Avg. Input Size	
		FTs	USIs	Orig.	Simplified
`print_tokens`	v1	5	4	78	3
	v2	48	1	240	1
	v4	28	1	39	2
	v6	186	2	58	1
	v7	24	24	53	1
`print_tokens2`	v4	403	9	128	2
	v5	172	1	18	1
	v6	518	93	33	1
	v7	292	1	153	2
	v8	256	74	49	2
	v9	60	1	76	2
	v10	172	1	18	1
`schedule`	v1	7	7	21	6
	v2	210	210	64	7
	v3	161	148	53	5
	v4	294	288	57	8
	v6	7	7	21	6
	v7	35	29	54	6
`schedule2`	v5	32	32	30	6
	v7	39	33	52	6
`replace`	v1	64	1	59	3
	v3	123	87	110	5
	v5	267	62	63	10
	v6	94	30	49	8
	v7	69	1	45	2
	v8	53	15	102	3
	v9	23	21	64	7
	v10	20	19	63	6
	v11	23	21	64	7
	v12	221	84	88	19
	v14	131	72	52	7
	v15	59	1	25	1
	v16	82	1	38	2
	v18	210	70	42	5
	v21	2	2	92	3
	v23	21	11	55	3
	v25	2	2	19	3
	v26	93	59	55	11

Table 3 shows the results for this experiment. The columns labeled *Avg. BS*, *Avg. FS*, and *Avg. FChop* show the average sizes of backward slices, forward slices, and failure-inducing chops respectively. The averages are computed over the number of unique simplified inputs (*USIs*) for each version. The columns labeled *In BS*, *In FS*, and *In FChop* respectively show the *fraction (out of total number of USIs for each faulty version)* of backward dynamic slices, forward dynamic slices and their failure-inducing chops that contain the faulty statement. This fraction ranges from 0 to 1. However, in most of the cases the fraction is 1 indicating that faulty statements are being captured by the failure-inducing chops.

Table 3: Results of fault location using simplified inputs.

Program	Version	Avg. *BS*	In *BS*	Avg. *FS*	In *FS*	Avg. *FChop*	In *FChop*
`print_tokens`	v1	72	1.00	60	1.00	50	1.00
	v2	44	1.00	32	1.00	23	1.00
	v4	65	1.00	56	1.00	44	1.00
	v6	63	1.00	54	1.00	42	1.00
	v7	66	1.00	59	1.00	44	1.00
`print_tokens2`	v4	54	0.00	73	1.00	40	0.00
	v5	52	1.00	64	1.00	36	1.00
	v6	77	1.00	75	1.00	58	1.00
	v7	54	0.00	70	1.00	40	0.00
	v8	48	0.00	57	1.00	33	0.00
	v9	54	0.00	70	1.00	40	0.00
	v10	46	1.00	54	1.00	33	1.00
`schedule`	v1	57	1.00	44	1.00	34	1.00
	v2	87	0.15	75	1.00	57	0.15
	v3	86	0.61	80	1.00	56	0.61
	v4	87	0.24	82	1.00	58	0.24
	v6	57	1.00	44	1.00	34	1.00
	v7	90	1.00	77	1.00	60	1.00
`schedule2`	v5	60	1.00	73	1.00	43	1.00
	v7	65	1.00	75	1.00	42	1.00
`replace`	v1	42	1.00	52	1.00	27	1.00
	v3	111	1.00	112	1.00	80	1.00
	v5	84	0.97	96	1.00	56	0.97
	v6	110	1.00	106	1.00	79	1.00
	v7	42	1.00	38	1.00	27	1.00
	v8	59	1.00	55	1.00	36	1.00
	v9	81	1.00	94	1.00	53	1.00
	v10	93	1.00	96	1.00	65	1.00
	v11	81	1.00	94	1.00	53	1.00
	v12	68	1.00	59	0.00/1	41/56	0.00/1
	v14	98	1.00	98	1.00	68	1.00
	v15	39	1.00	36	1.00	24	1.00
	v16	42	1.00	38	1.00	27	1.00
	v18	94	1.00	95	1.00	61	1.00
	v21	55	1.00	57	1.00	34	1.00
	v23	69	0.64	81	1.00	42	0.64
	v25	92	1.00	96	1.00	60	1.00
	v26	85	0.12	99	1.00	57	0.12

Table 4: Average per benchmark: results of fault location using simplified inputs.

Program	Avg. BS	In BS	Avg. FS	In FS	Avg. FChop	In FChop	FChop/BS	FChop/ALL
print_tokens	62	1	52.2	1	40.6	1	0.65	0.07
print_tokens2	55	0.43	66.14	1	40	0.43	0.73	0.08
schedule	77.33	0.67	67	1	49.83	0.67	0.64	0.12
schedule2	62.5	1	74	1	42.5	1	0.68	0.14
replace	74.72	0.93	77.89	1	50.78	0.93	0.68	0.09

Table 5: Average time in seconds for simplifying inputs and computing slices.

Program	# of failed Test Cases	Simplification Time	# of Unique Simplified Inputs	Average Slicing Time
print_tokens	291	1.08	32	7.41
print_tokens2	1873	0.75	180	3.23
schedule	714	0.96	689	10.19
schedule2	71	0.58	65	8.11
replace	1557	1.5	559	12.67

Table 6: Potential benefits of failure-inducing input differences.

Program	Version	Avg. *BS*	In *BS*	Avg.*FS*	In *FS*	Avg. *FChop*	In *FChop*
print_tokens	v1	72	1.00	55	1.00	46	1.00
	v2	44	1.00	32	1.00	23	1.00
	v4	65	1.00	49	1.00	39	1.00
	v6	63	1.00	50	1.00	40	1.00
	v7	66	1.00	59	1.00	44	1.00
print_tokens2	v4	54	0.00	50	0.50	24	0.00
	v5	52	1.00	64	1.00	36	1.00
	v6	77	1.00	75	1.00	57	1.00
	v7	54	0.00	48	1.00	26	0.00
	v8	48	0.00	41	1.00	22	0.00
	v9	54	0.00	48	1.00	26	0.00
	v10	46	1.00	54	1.00	33	1.00
schedule	v1	57	1.00	19	0.76	15	0.76
	v2	87	0.15	43	0.74	31	0.13
	v3	86	0.61	50	0.53	35	0.33
	v4	87	0.24	52	0.57	37	0.16
	v6	57	1.00	19	0.76	15	0.76
	v7	90	1.00	49	0.63	37	0.63
schedule2	v5	59	1.00	49	0.77	29	0.77
	v7	64	1.00	46	0.86	26	0.86
replace	v1	42	1.00	43	0.67	22	0.67
	v3	111	1.00	75	0.83	50	0.83
	v5	84	0.97	68	0.67	40	0.66
	v6	110	1.00	79	0.88	56	0.88
	v7	42	1.00	34	1.00	25	1.00
	v8	59	1.00	51	1.00	34	1.00
	v9	81	1.00	65	0.56	33	0.56
	v10	93	1.00	71	0.60	45	0.60
	v11	81	1.00	65	0.56	33	0.56
	v12	68	1.00	52	0.00/1	35/50	0.00/1
	v14	98	1.00	71	0.86	48	0.86
	v15	39	1.00	36	1.00	24	1.00
	v16	42	1.00	34	1.00	25	1.00
	v18	94	1.00	68	0.82	42	0.82
	v21	55	1.00	38	0.57	20	0.57
	v23	69	0.64	46	0.46	20	0.20
	v25	92	1.00	61	0.83	39	0.83
	v26	85	0.12	71	0.81	38	0.09

Table 7: Average per benchmark: potential benefits of failure-inducing input differences.

Program	Avg. BS	In BS	Avg. FS	In FS	Avg. FChop	In FChop	FChop/BS	FChop/ALL
print_tokens	62	1	49	1	38.4	1	0.62	0.07
print_tokens2	55	0.43	54.29	0.93	32	0.43	0.58	0.06
schedule	77.33	0.67	38.67	0.67	28.33	0.46	0.37	0.07
schedule2	61.5	1	47.5	0.82	27.5	0.82	0.45	0.09
replace	74.72	0.93	57.11	0.78	36.28	0.73	0.49	0.06

Table 8: Overview of benchmark programs for memory related bugs.

Program	Fault Description	Fault Location	Isolation On
gzip-1.0.7	1024 byte long filename overflows into global variable	line 40 in strcpy.c	file name
gzip-1.2.4	1024 byte long filename overflows into global variable	line 1009 in gzip.c	file name
ncompress-4.2.4	1024 byte long filename corrupts stack return address	line 886 in compress42.c	file name
polymorph-0.4.0	2048 byte long filename corrupts stack return address	line 118 in polymorph.c	file name
tar-1.13.25	wrong loop bounds lead to heap object overflow	line 92 in prepargs.c	env. variables
bc-1.06	misuse of bounds variable corrupts heap objects	line 176 in storage.c	file contents
tidy-34132	memory corruption problem	line 3505 in parser.c	file contents

The version v12 of *replace* program presents an interesting case. There is a faulty `#define` statement in this version. We modified the program by replacing all `#define` statements with corresponding assignment statements. For example, the erroneous `#define` in this faulty version `#define MAXPAT 50` is replaced by `int MAXPAT = 50;`. Thus, the error in a `#define` becomes an error in an assignment statement. The problem in this error, however, is that the faulty assignment statement does not depend upon any program input. Therefore, it will never be present in the forward slice of any input. This shows a limitation of using forward slices for locating faulty code. Even if the faulty statement is in the backward slice, the result of intersection is negative. To address this problem, we make a little change in our fault location algorithm. We simply added all the statements in the backward dynamic slices, which do not depend upon any input, into the intersection. After that, the average size of the *FChop* increased, from 41 to 56, but the result became positive.

The overall average results for each program are shown in Table 4. The table shows that the range for the average fraction of *In BS* is from 0.43 to 1.0 and the average fraction of *In FS* is always 1. Also, the range for average fraction of *In FChop* is from 0.43 to 1.0. In this table, the column labeled *FChop/BS* shows the average ratio of the size of *FChop* to *BS* and the column labeled *FChop/ALL* shows the average ratio of the size of *FChop* to the size of the whole program. As we can see, the range of *FChop/BS* is from 0.64 to 0.73 (i.e., significant reductions over backward slices are observed) and the range of *FChop/ALL* is 0.07 to 0.14, which is less than 14% of the size of the whole program. Another interesting observation is that although the sizes of forward dynamic slices in these experiments were comparable to the sizes of backward dynamic slices, forward dynamic slices were found to be more effective at containing the faulty code than backward dynamic slices.

A recent paper [7] compares the performance of using cause transitions [7] and nearest neighbors [21] approaches to locate faulty code in the Siemens suite. The results show that the nearest neighbors method can locate the faulty statement within 10% of code in 16.51% of all test runs whereas the cause transitions method can locate a faulty statement within 10% of code in about 26.36% of all test runs. Although we have not considered the faulty versions from Siemens suite containing faults of omission in our experiments, for the 38 faulty versions and for 1525 failed runs corresponding to the total number of *USIs* for these faulty versions, our approach located the faulty statement in 60.78% of these failed runs with the average size of the *FChop* being 10.4% of the size of the program.

The time performance of the implementation of our algorithm is shown in Table 5. The average time taken for simplification of input and the average time taken to compute the forward and backward slices is given in seconds. As seen from the table, the total time taken to narrow down the search for faulty code to a small subset of program statements is typically few seconds. The experiments were conducted on Intel Pentium 4 with 3.00GHz CPU and 1G memory and Linux 2.6.10 platform.

Recall that above results were obtained by implementing the *ddmin* algorithm and not the *dd* algorithm. The simplified input obtained by *ddmin* may in general contain several input values. If the algorithm *dd* to isolate failure-inducing input differences is applied to the simplified input, a smaller input difference and hence smaller chop may result. To estimate the potential of isolating failure-inducing input differences for further reducing the chop sizes, we conducted another experiment. In this experiment, forward slices, and hence the chops, were computed with respect to each value in the simplified input. For example, when the simplified input for *print_tokens* contained more than one token, the forward slices for each of these tokens are separately computed. The chops produced in this manner will clearly be smaller. Table 6 shows average results for the above chops. We give two results for *replace v12* - one without changing the fault location algorithm and the second by applying the modification mentioned previously.

The overall average results for each program are shown in Table 7. The table shows that the range for the average fraction of *In FS* is from 0.67 to 1.0 (recall that it was always 1.0 for *In FS* in the previous experiment). The range for average fraction of *In FChop* is from 0.43 to 1.0. Note that for some programs this average has fallen in comparison to earlier *In FChop* values. In this table, the column labeled *FChop/BS* shows the average ratio of the size of *FChop* to *BS* and the column labeled *FChop/ALL* shows the average ratio of the size of *FChop* to the size of the whole program. As we can see, the range of *FChop/BS* is from 0.37 to 0.62 (as opposed to 0.64 to 0.73 in the previous experiment) and the range of *FChop/ALL* is 0.06 to 0.09, which is at most 9% of the size of the whole program.

4.2 Reported Memory Related Faults

Our next evaluation is based upon a set of program versions taken from [33, 20] for which real faults have been reported. In these programs, listed in Table 8, the fault causes memory corruption which eventually leads to a segmentation error. We observe that the crashes caused in the above programs are directly related to either the length of the input or some particular value in the input. When a corrupted memory location is accessed, the program crashes.

To find a failure-inducing input we again simplify the input for a failed run. Unlike many of the faulty versions of programs in the Siemens suite, these programs do not produce an output since the program crashes before output can be produced. Therefore we distinguish between a failing run and a successful run as follows. Consider an input on which the program crashes at some execution point p. Now this input is reduced by removing the last input value. The program is run again with a breakpoint set just past the point p. If the breakpoint is not reached, i.e. program crashes again at p, the input is further reduced and the process is repeated. Note that due to the nature of bugs and the way the input is reduced (i.e., from the end), the cause of the crash remains the same. On the other hand, if the breakpoint is reached, it means that the latest change in input resulted in the crash being avoided. In this case the part of the input that was most recently removed must be the *failure-inducing input difference*. Thus, the above approach *isolated* the failure-inducing input difference for these memory related bugs. By executing the program on the most recent failing input, we are now able to collect the backward slice of the erroneous output and the forward slice of the failure-inducing input difference. Thus, the failure-inducing chop can now be computed.

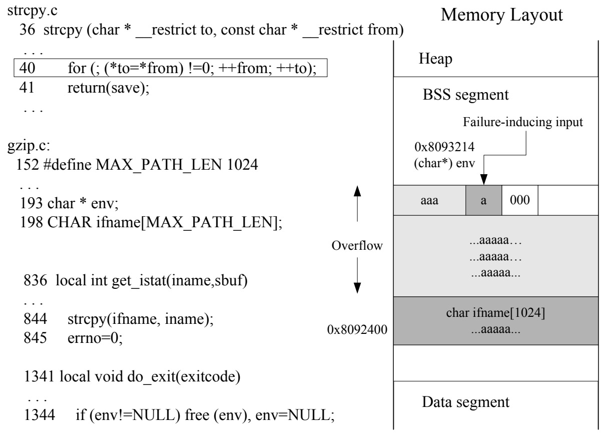
The results of our experiments are summarized in Table 9. In this table *Exec* is the number of lines of code that are executed at least once during the program run, *BS* and *FS* are the number of lines of code that belong to the backward and forward slices respectively, and *FChop* is the number of lines of code in the failure-inducing chop. The results we obtained are very encouraging. First we found that the statement whose execution corrupted the memory was present in the failure-inducing chops in all of these cases. Second we observe that in first four programs the number of lines of code in *FChop* is so small (2 to 4) that the chop essentially pinpoints the appropriate statement. Moreover the *FS* is as effective as the *FChop* in these cases. Finally we observe that for the last three programs, although the chops are larger, they are significantly smaller than the sizes of *Exec*, *BS*, and *FS*. The time to isolate the failure-inducing input for these programs was in the range of 0.05

seconds to 3.06 seconds. For all programs other than *tidy*, the time to compute the forward and backward slices was in the range of 2.43 seconds to 13.58 seconds. For the *tidy* program, it took 174.86 seconds to compute the forward and backward slices due to long length of the run. In conclusion, these results show that our technique for computing chops based upon integration of delta debugging and dynamic slicing is effective in identifying code causing memory corruption.

Table 9: FChop sizes.

Program	Exec	BS	FS	FChop
gzip-1.0.7	115	39	4	4
gzip-1.2.4	118	34	3	3
ncompress-4.2.4	59	18	2	2
polymorph-0.4.0	45	21	3	3
tar-1.13.25	445	180	122	76
bc-1.06	636	204	182	102
tidy-34132	1519	540	346	164

To provide understanding of how failure-inducing chop captures the statement causing memory corruption, let us consider the case of gzip-1.0.7 which has a known buffer overflow problem. Figure 4 illustrates the details of this problem. On the left hand side of Figure 4, we show the relevant code segment for the problem. The problem happens 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 the string `iname`. If the length of string `iname` is longer than 1024, then the buffer overflows. If the length of string `iname` is larger than 3604, the value of `env` is changed due to buffer overflow. This is because according to the memory layout shown in Figure 4, the difference between `env` and `ifname` is 3604 bytes. Later when at line 1344 *free(env)* is executed, the program crashes due to presence of an illegal memory address in `env`.



```
strcpy.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  local 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: gzip buffer overflow.

We picked an input file name `'a <repeated 3610 times>'` on which the above program crashes because the length of `iname` is larger than 3604. After applying the simplification process described in this section, we find a successful input in which the file name is `'a <repeated 3604 times>'` and the most recently failed run had its input as the file name `'a <repeated 3605 times>'`. Thus, the failure-inducing input difference between them is the last character 'a' in the input of the most recently failed run. We used our slicing tool to compute the forward slice of the above failure-inducing input difference and the backward slice of `env` at line 1344 for the most recently failed run. The size of the forward slice was 4 lines of code and the size of the backward slice was 35 lines of code. Their intersection which is the same as the forward slice includes the `for` statement at line 40 in `strcpy.c`, which is the place where the buffer overflow occurred. Note that the overflow occurred at a statement executed inside `strcpy.c` which is not a part of the source code of `gzip`. However, since our slicing tool is running on the binary code, it is able to perform slicing on both the source code and the library code.

5. RELATED WORK

In a series of articles [28, 27, 26], the *delta debugging* algorithm has been developed to automatically simplify or isolate a failure-inducing input [28, 27], produce cause effect chains [26] and to link cause transitions [7] to the faulty code. These works use delta debugging algorithm to analyze *program state* changes during the execution of the program. Program state based analysis is difficult and expensive for C programs [7].

On the other hand, *program source code* analysis based techniques have explored the use of backward dynamic slicing [1, 2, 3, 4, 6, 9, 17, 18, 24, 29, 30, 31, 32] for debugging. One problem that had existed for some time was the cost of computing dynamic slicing. However, in our recent work [32] we developed a practical slicing algorithm whose average slicing times range from 1.74 to 36.25 seconds across several benchmarks from SPECInt2000/95. In [32, 29] we have also shown that the number of distinct statements executed at least once during a program execution can be 2.46 and 56.08 times more than the number of statements in a backward dynamic slice. In [31], we presented an experimental study which shows dynamic slicing is effective in locating faulty code. Although, backward dynamic slices typically contain only a small fraction of executed statements, the number of statements can still be large. This is what motivated our current work. The prior research has explored only the use of *backward dynamic slicing* for debugging. However, in this paper we have shown how a minimal failure-inducing input computed using delta debugging can enable the use of *forward dynamic slices* for debugging. In our experiments, the forward dynamic slices were found to be effective in containing the faulty code and were often smaller than the backward dynamic slices. Our work in this paper shows how the use of delta debugging can be combined with both forward and backward dynamic slices to compute failure-inducing chops that are *smaller* than either backward or forward dynamic slice.

Harrold et al. [11] compared the spectra of passing and failing runs and found that failing runs tend to have unusual coverage spectra. Jones et al. [16] ranked each statement according to its ratio of failing tests to correct tests and used this information to assist fault location. Liblit et al. [19] 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 [21] 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. Xie et al. show that many redundancies [25] in programs correspond to hard program errors. Hangal et al. [10] identified the causes of some programming errors in Java programs by observing violations of program invariants. In [12], 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.

6. CONCLUSIONS

In this paper, we propose a novel way to combine the the delta debugging algorithm with the forward and backward dynamic slicing to narrow down the search for faulty code. Although, in prior work backward dynamic slicing has been considered useful for program debugging, our work for the first time shows how the application of forward slicing in locating faulty code is enabled by the delta debugging technique. As our experiments show the sizes of chops induced by the combination of forward and backward slicing are much smaller than either forward or backward slices without significantly compromising the fault detection effectiveness.

7. REFERENCES

[1] H. Agrawal and J. Horgan, “Dynamic program slicing,” *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 246-256, 1990.

[2] H. Agrawal, R. DeMillo, and E. Spafford, “Debugging with dynamic slicing and backtracking,” *Software Practice and Experience* (SP&E), Vol. 23, No. 6, pages 589-616, 1993.

[3] H. Agrawal, J.R. Horgan, E.W. , and S.A. London, “Incremental regression testing,” *IEEE Conference on Software Maintenance* (ICSM), Montreal, Canada, 1993.

[4] H. Agrawal, J. Horgan, S. London, and W. Wong, “Fault localization using execution slices and dataflow tests,” *Sixth IEEE International Symposium on Software Reliability Engineering* (ISSRE), pages 143-151, 1995.

[5] *AskIgor*, Automated Debugging Service. http://www.st.cs.uni-sb.de/askigor/

[6] A. Beszedes, T. Gergely, Z.M. Szabo, J. Csirik, and T. Gyimothy, “Dynamic slicing method for maintenance of large C programs,” *5th European Conference on Software Maintenance and Reengineering* (CSMR), pages 105-113, March 2001.

[7] H. Cleve and Andreas Zeller, “Locating causes of program failures,” *27th International Conference on Software Engineering* (ICSE), pages 342-351, 2005.

[8] Diablo Is A Better Link-time Optimizer. *http://www.elis.ugent.be/diablo/*

[9] T. Gyimothy, A. Beszedes, I. Forgacs, “An efficient relevant slicing method for debugging,” *7th European Software Engineering Conference/ 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (ESEC/FSE), Toulouse, France, 1999.

[10] S. Hangal and M.S. Lam, “Tracking down software bugs using automatic anomaly detection,” *International Conference on Software Engineering* (ICSE), 2002.

[11] 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.

[12] H. He and N. Gupta, “Automated Debugging using Path-Based Weakest Preconditions,” *Fundamental Approaches to Software Engineering* (FASE), ETAPS Joint Conference, Barcelona, Spain, March 29-31, 2004.

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

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, “Experiments on the effectiveness of dataflow and controlflow based test adequacy criteria,” *16th International Conference on Software Engineering* (ICSE), pages 191-200, 1994.

[15] Information Week, Issue on Software Quality, Jan 21, 2002.

[16] J.A. Jones, M.J. Harrold, and J. Stasko, ”Visualization of test information to assist fault localization,” *International Conf. on Software Engineering* (ICSE), page 467-477, 2002.

[17] B. Korel and J. Laski, “Dynamic program slicing,” *Information Processing Letters* (IPL), Vol. 29, No. 3, pages 155-163, 1988.

[18] B. Korel and J. Rilling, “Application of dynamic slicing in program debugging,” *3rd International Workshop on Automatic Debugging* (AADEBUG), Linkoping, 1997.

[19] 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* (PLDI), San Diego, California, June 2003.

[20] S. Narayanaswamy, G. Pokam, and B. Calder, “BugNet: continuously recording program execution for deterministic replay debugging,” *32nd International Symposium on Computer Architecture* (ISCA), pages 284-295, 2005.

[21] M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries,” *Automated Software Engineering* (ASE), 2003.

[22] The Unravel Project. *http://hissa.nist.gov/unravel/*

[23] Valgrind. *http://valgrind.org/*

[24] M. Weiser, “Program slicing,” *IEEE Transactions on Software Engineering* (TSE), Vol. SE-10, No. 4, pages 352-357, 1982.

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

[26] A. Zeller, “Isolating cause-effect chains from computer programs,” *10th ACM SIGSOFT Symposium on Foundations of Software Engineering* (FSE), Charleston, South Carolina, 2002.

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

[28] A. Zeller and R. Hildebrandt, “Simplifying and isolating failure-inducing input,” *IEEE Transactions on Software Engineering* (TSE), Vol 28, No 2, Feb. 2002.

[29] X. Zhang, R. Gupta, and Y. Zhang “Precise dynamic slicing algorithms,” *IEEE/ACM International Conference on Software Engineering* (ICSE), pages 319-329, Portland, Oregon, May 2003.

[30] 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* (ICSE), pages 502-511, 2004.

[31] 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* (AADEBUG), Monterey, California, September 2005.

[32] X. Zhang and R. Gupta, “Cost effective dynamic program slicing,” *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 94-106, Washington D.C., June 2004.

[33] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S.P. Midkiff, and J. Torrellas, “AccMon: automatically detecting memory-related bugs via program counter-based invariants,” *37th Annual International Symposium on Microarchitecture* (MICRO), pages 269-280, 2004.