Fault Localization Using Value Replacement

Dennis Jeffrey [(1)] jeffreyd@cs.ucr.edu
Neelam Gupta guptajneelam@gmail.com
Rajiv Gupta [(1)] gupta@cs.ucr.edu

[(1)]Univ. of California at Riverside, CSE Department, Riverside, CA 92521

ABSTRACT

We present a value profile based approach for ranking program statements according to their likelihood of being faulty. The key idea is to see which program statements exercised during a failing run use values that can be altered so that the execution instead produces correct output. Our approach is effective in locating statements that are either faulty or directly linked to a faulty statement. We present experimental results showing the effectiveness and efficiency of our approach. Our approach outperforms *Tarantula* [9] which, to our knowledge, is the most effective prior approach for statement ranking based fault localization using the benchmark programs we studied.

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, Experimentation, Measurement, Reliability, Verification

Keywords

automated debugging, fault localization, value replacement, interesting value mapping pair

1. INTRODUCTION

Software debugging is the process of locating and correcting faulty program statements. This is a necessary phase of software development because building software is a human-intensive activity and is therefore prone to error. Unfortunately, debugging can be a difficult task. The point of a program failure (e.g., when a program produces unexpected output) can be much later than when the faulty statement responsible for the failure is executed. After a faulty statement is identified, the appropriate correction may not be obvious; it may require additional time to analyze the fault and deduce the appropriate correction. Techniques to help automate the debugging process can reduce developer burden and increase the rate at which program faults can be corrected. This can result in deployed software that is more robust and reliable.

Prior research in automating the debugging process has focused primarily on *fault localization*. This is the process of narrowing or guiding the search for faulty code to help a developer find erroneous statements more quickly. Dynamic analysis approaches for fault localization include: *dynamic program slicing* [1, 10, 16, 22], *delta debugging* [2, 19, 20], the *nearest neighbor* technique [15], and other statistical techniques [9, 12, 13, 14]. Statistical techniques use data collected during execution of failing and successful runs to rank statements in the order of their likelihood of being faulty. In particular, the *Tarantula* approach [9] uses a formula to rank program statements according to the intuition that the statements primarily executed by failing runs are more likely to be faulty than the statements primarily executed by successful runs. It was shown [9] that *Tarantula* is more effective at identifying erroneous statements than *cause transitions* [2] on the Siemens benchmark suite. The *Tarantula* approach for fault localization is very fast at ranking program statements because it takes into account only the statement coverage information of failing and successful runs. However, this statistical information can be misleading. For instance, a failing run that exercises a particular statement does not necessarily imply that the statement influences the incorrect output of that run. Also, a successful run may exercise a faulty statement but still produce correct output due to many-to-one mappings of used and defined values in a program. Therefore, there is clearly scope to improve upon *Tarantula*.

In this paper, we propose a new value profile based approach for fault localization that involves *searching for the program statements that can be shown to affect the output of a failing run such that the incorrect output becomes correct.* This is done by replacing the values used at a statement during the execution of a failing run with an alternate set of values, then checking to see whether the resulting output changes to become correct. If so, then we have identified what we call an *Interesting Value Mapping Pair (IVMP)*. An IVMP consists of the original value mapping used at the statement instance, and the corresponding alternate value mapping that can be substituted at that point to correct the output of the failing run. We have seen that IVMPs often occur at faulty statements or statements that are directly linked to faulty statements via a dependence edge. Our approach therefore uses IVMPs to rank program statements according to their likelihood of being faulty. Our experiments show that our approach for fault localization is generally more effective than *Tarantula* – and therefore, more effective than *cause transitions* – on the Siemens benchmark programs.

The rest of this paper is organized as follows. In the next section, we describe IVMPs in detail and show how they can be directly linked to faulty statements. Section 3 describes how we rank program statements using IVMPs to assist in fault localization. Our experimental study is given in Section 4. Section 5 presents related work, and Section 6 summarizes our conclusions.

2. IVMPs AND FAULTY STATEMENTS

2.1 Definitions and IVMP Identification

A *value mapping* is the set of all values involved at a single execution instance of a statement during a program run. Thus, a value mapping includes both the used values and the corresponding defined values. The values used at a particular statement instance are essentially "mapped" by the statement to the defined values. For predicate statements, the defined value is considered to be the branch outcome.

An *Interesting Value Mapping Pair (IVMP)* is a pair of value mappings associated with a particular statement instance in a failing program execution, with a special property: it shows how the value mapping used at that statement instance can be replaced with an alternate value mapping so that the execution instead produces the correct output.

> An **Interesting Value Mapping Pair (IVMP)** is a pair of value mappings (*original*, *alternate*) associated with a particular statement instance in a failing run, such that: (1) *original* is the original value mapping used by the failing run at that instance; and (2) *alternate* is an alternate (different) value mapping such that if the values in *original* are replaced by the values in *alternate* at that instance during re-execution of the failing run, then the incorrect output of the failing run becomes correct.

Given a failing run, the task of searching for IVMPs is straightforward: simply consider statement instances in the failing run one at a time, replacing the value mapping used at each one with a different value mapping, then checking to see if the output of the run becomes corrected. If so, an IVMP has been found. Searching for IVMPs requires only a failing test case execution with the corresponding incorrect and correct outputs, and some set of alternate value mappings that can be applied at different statement instances in the failing execution.

In general, the set of all possible alternate value mappings at a statement instance can be infinite. A method is required to select a finite set of alternate mappings that can be applied at each statement. To do this, we extract the (finite) set of alternate mappings for each statement from the execution traces of *all* test cases in an available test suite. We call this set of alternate mappings the *value profile*.

> A **Value Profile** for a program with respect to a test suite is a table mapping each program statement to the set of all unique value mappings occurring at that statement during execution of test cases in the test suite.

It is reasonable to assume the existence of a test suite for computing the value profile since a failing test case is usually part of a larger suite of test cases. We have observed that rich value profiles can result from only a few test cases, and yet the sizes of value profiles seem to increase logarithmically in general as the number of test cases in the suites increase. This is because as more test cases are added, the value mappings used by a test case will tend to match value mappings already added to the value profile from previous test cases. Also, in the value profile we do not distinguish between alternate value mappings from successful or failing executions. This is because alternate mappings that may result in IVMPs can potentially come from *any* test case executions, regardless of whether the executions pass or fail.

There may be situations in which the correct output of a failing test case is not known. In these cases, our approach is applicable as long as there is a way to determine whether the failing program output "improves" when an alternate value mapping is applied. For example, if a failing run results in a program crash and the expected output is unknown, then the expected output may be considered as "no crash." In this case, applying any alternate value mappings that cause the failing run to avoid the crash will yield IVMPs.

The general algorithm for searching for IVMPs is given in Fig. 1. Given a test suite with a failing test case for some program, the first step initializes the value profile using the traces of the tests in the test suite. The second step searches for IVMPs by replacing the value mapping at each statement instance in the failing execution with every alternate value mapping from that statement as specified in the value profile. The runtime of this algorithm is therefore bounded by $O(t \times m)$, where t is the number of statement instances in the execution trace of the failing run, and m is the maximum number of alternate mappings for any statement in the value profile. Although this algorithm shows how to compute all possible IVMPs for a failing run (with respect to a particular test suite), we will later see how we can more efficiently search for a much smaller subset of IVMPs that are effective for fault localization.

2.2 Link to Faulty Statements: Examples

IVMPs occur precisely at faulty statements in many cases. In cases where this is not possible, IVMPs can occur at statements that are just one dependence edge away from faulty statements. Because of this, IVMPs can be useful for fault localization. Here we give several examples showing different ways in which IVMPs can be closely linked to faulty statements. These examples are based on situations that we encountered using the Siemens benchmark programs [8].

IVMPs at a Faulty Statement. IVMPs can be found precisely at a faulty statement when applying an alternate value mapping causes the faulty statement to define the correct value. Fig. 2 shows a code fragment and a test suite based on Siemens program `schedule`, faulty version v9. This fragment of code involves a check on the number of input arguments (*argc*), so that the program terminates with an error if there are too few input arguments specified. There is an off-by-1 fault in this condition.

```
input:
    Faulty program $P$, and failing test case $f$ (with
    actual and expected output) from test suite $T$.
output:
    Set of identified IVMPs for $f$.
algorithm SearchForIVMPs
begin
Step 1:  [Compute value profile for $P$ w/ respect to $T$]
1:      $valProf$ := {};
2:      for each test case $t$ in $T$ do
3:          $trace$ := trace of value mappings from
                execution of $t$;
4:          augment $valProf$ using the data in $trace$;
        end for
Step 2:  [Search for IVMPs in $f$]
5:      $trace_f$ := trace of value mappings from
            execution of $f$;
6:      for each statement instance $i$ in $trace_f$ do
7:          $origMap$ := value mapping from $trace_f$ at $i$;
8:          $s$ := the statement associated with instance $i$;
9:          for each $altMap$ in $valProf$ at $s$ do
10:             execute $f$ while replacing $origMap$
                    with $altMap$ at $i$;
11:             if output of $f$ becomes correct then
12:                 output IVMP ($origMap$, $altMap$) at $i$;
            end for
        end for
end SearchForIVMPs
```

Figure 1: General algorithm for computing IVMPs with respect to a failing run and its test suite.

```
    argc := ...;
1:  if (argc < 3) /* 3 should actually be 4 */
2:      print ("Too few");
3:  else
4:      print ("Okay");
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$argc = 2$	Too few	Too few	PASS
B	$argc = 3$	Okay	Too few	FAIL
C	$argc = 4$	Okay	Okay	PASS

Figure 2: Code fragment and test suite based on `schedule` faulty version v9.

The effect of the above fault is that when *argc* is equal to 3, the program will erroneously proceed as normal when it should have terminated early due to too few input arguments. Thus, executing test case B in Fig. 2 results in a failure. However, for test case B, changing the value of *argc* at line 1 from 3 to 2 (which is the value used by test case A) causes the output of the failing run to become correct. Therefore, this represents an IVMP providing an important clue that at line 1 in the code fragment, the value of variable *argc* should be decremented by 1 (or equivalently, the value of constant 3 should be incremented by 1). In this case, the IVMP is located at precisely the faulty statement.

IVMPs Directly Linked to a Faulty Statement. In some cases, we may not be able to find IVMPs at precisely the faulty statements. One situation where this can happen is when there is a fault in a constant assignment statement. A constant assignment will never be associated with an IVMP because there are no alternate values at the assignment; every executed instance of a constant assignment will define the same constant value. Instead, we can get IVMPs at the statements in which the defined constant values are used. Fig. 3 shows a code fragment and test suite based on Siemens program `tcas`, faulty version v7. The code fragment shows an erroneously-defined constant value at line 2, which is larger than it should be.

```
    AltLayVal := ...;
1:  Pos_RA_Alt_Thresh[0] = 400;
2:  Pos_RA_Alt_Thresh[1] = 550; /* Should be 500 */
3:  Pos_RA_Alt_Thresh[2] = 640;
4:  Pos_RA_Alt_Thresh[3] = 740;
    ...
5:  if (Pos_RA_Alt_Thresh[AltLayVal] < 525)
6:      print (0);
7:  else
8:      print (1);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	$AltLayVal = 0$	0	0	PASS
B	$AltLayVal = 1$	1	0	FAIL
C	$AltLayVal = 2$	1	1	PASS

Figure 3: Code fragment and test suite based on `tcas` faulty version v7.

The effect is that when the array index $AltLayVal$ is 1, the condition at line 5 will erroneously evaluate to `false` instead of `true` due to the incorrect constant value defined at that position. Thus, executing test case B in Fig. 3 results in a failing run. However, for test case B, changing the value of $AltLayVal$ at line 5 from 1 to 0 (which is the value used by test case A) causes the output of the failing run to become correct. Assuming the value of index variable $AltLayVal$ is correct, this IVMP provides the important clue that the value stored at array index 1 is incorrect. Further, since accessing array index 0 (with value 400) corrects the output of the failing run, this provides the hint that the value 550 at array index 1 should be changed to something smaller. In this case, the IVMP is located at line 5, which is one data dependence edge away from the faulty statement at line 2.

IVMPs in the Presence of Erroneously-Omitted Statements. Another situation in which IVMPs cannot be found at an erroneous statement is when the fault involves one or more missing statements. In these cases, IVMPs can still be found at nearby statements that can compensate for the effects of the missing code. Fig. 4 shows an erroneous function and accompanying test suite inspired by `schedule2`, faulty version v1. The purpose of this function is to return the inputted value of x incremented by one, only when the value of y is positive (in bounds). If y is equal to 0, the function returns 0.

The missing code at line 1 is meant to check whether y is negative (out of bounds), and if so, to return the original value of x without having incremented it. Since test case A has y with out-of-bounds value -1, then the function erroneously increments the value of x in this case when it should not have done so. When the value of x at line 3 is changed from 1 to 0 (which is the value used by test case C), then the output becomes 1 and is correct. This IVMP

```
   int foo(int x, int y)
1:     /* if (y < 0) return x; */
2:     if (y == 0) return 0;
3:     return x + 1;
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	(x,y) = (1,-1)	2	1	FAIL
B	(x,y) = (2,2)	3	3	PASS
C	(x,y) = (0,1)	1	1	PASS

Figure 4: Code fragment and test suite inspired by `schedule2` faulty version v1.

at line 3 provides the important clue that for the failing run corresponding to test case A, the value for x should actually not have been incremented. This suggests that a statement (the one at line 1) is missing in the above function that will prevent test case A from incrementing the value of x.

Some program faults may involve extraneous statements. We do not have a detailed example of this case due to space limitations. However, from our experiments we have found that IVMPs often occur precisely at extraneous statements where they have the effect of "canceling out" the effects of the extra code. For instance, an extraneous assignment statement can have an IVMP that forces the original value to be defined, rather than the new value that resulted from the extra code. An extraneous condition can have an IVMP that alters the conditional outcome in such a way that the behavior is as if the condition is not present.

3. RANKING STATEMENTS USING IVMPs

While IVMPs often occur at or near faulty statements, a significant challenge to using IVMPs for fault localization is that IVMPs can be found at other statements besides those that are faulty. This is possible because there are often multiple statements exercised during a failing execution whose values can be changed to cause the output to become correct. We have identified two main causes for this. We refer to these two causes as the *dependence cause* and the *compensation cause* for IVMPs at multiple statements.

Dependence Cause. IVMPs may be found at different statements that are all part of the same definition-use chain in a program. This is because if a statement S_1 defining a variable x has an IVMP associated with it, there's a chance that another statement S_2 that uses x will also have an IVMP associated with it. In such cases, changing the value of x at either S_1 or S_2 can correct the program output, even though only one of the two statements may be faulty.

Compensation Cause. This occurs when IVMPs are found at two different statements that do not appear to be related to each other at all, yet they both influence the output such that applying an alternate value mapping at either statement can compensate for the effects of the fault on the program output, thereby making the output correct.

To address the challenge posed by the dependence and compensation causes for IVMPs at multiple statements, we consider the IVMPs that are computed from *multiple* failing runs. A dependence chain with IVMPs in one failing run may not exist in another failing run that may have very different dependence chains. Also, IVMPs that happen to compensate for a fault in one failing run are unlikely to compensate for the fault in the same way in another failing

```
1:  read (x,y);
2:  a := x + y; /* should be x − y */
3:  if (x < y)
4:      z := a;
5:  else
6:      z := a + 1;
7:  print (z);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	(x,y) = (0,0)	1	1	PASS
B	(x,y) = (-1,0)	-1	-1	PASS
C	(x,y) = (1,1)	3	1	FAIL
D	(x,y) = (0,1)	1	-1	FAIL

Figure 5: Example with test suite to motivate the need for considering multiple failing runs when ranking statements using IVMPs.

run. Considering multiple failing runs is particularly effective when the failing runs exercise very different paths in the program. Since all failing runs must traverse the fault, the statements that are associated with IVMPs in more failing runs have a greater likelihood of being faulty. We therefore rank IVMP statements using the intuition that *statements associated with IVMPs in more failing runs are more likely to be faulty than statements that are associated with IVMPs in fewer failing runs.* Consider the example program with accompanying test suite in Fig. 5.

In this example program, there is a fault at line 2 in which the addition operator is mistakenly used instead of the subtraction operator. In cases where inputted value y is 0, the defined value of a at line 2 will be correct regardless of the fault. As a result, only test cases A and B in Fig. 5 pass, while test cases C and D fail.

Consider failing test case C. We will get an IVMP at line 2 because changing the values of x and y respectively from 1 and 1, to 0 and 0 (which are used by test case A), will correct the program output. Also, we will get an IVMP at line 6 because changing the used value of a from 2 to 0 (which is the value of a used by test case A), will correct the output as well. Although we get IVMPs at lines 2 and 6, only one of these lines contains the actual fault. The IVMP at the other line is present due to the *dependence cause* for IVMPs at multiple statements. To help us distinguish between these two statements, we consider another failing run.

When considering failing test case D, we get an IVMP at line 2 because changing x and y from 0 and 1, to -1 and 0 (used by test case B), will correct the output. Also, we get an IVMP at line 4 because changing the value of a here from 1 to -1 (the value used for a in test case B) will correct the output. Here, we get IVMPs at lines 2 and 4.

We now consider the statements with IVMPs in both failing runs C and D. Line 2 is associated with IVMPs in both failing runs, whereas lines 4 and 6 are associated with IVMPs in only one failing run each. Therefore, line 2 is more likely to be faulty than either lines 4 or 6.

The example described above shows the benefit of considering IVMPs from multiple failing runs when ranking program statements using IVMPs. Formally, we rank the statements exercised by failing runs in decreasing order of their *suspiciousness* values (likelihood of being faulty). Let F be the set of all failing runs in an available test suite, and let $STMT_{IVMP}(f)$ refer to the set of all program statements associated with at least one IVMP identified from failing run f. Then we define the *suspiciousness of a statement s*, *suspiciousness*(s), as the number of failing runs in which at least one IVMP was identified for that statement.

$$suspiciousness(s) := |\{f : f \in F \wedge s \in STMT_{IVMP}(f)\}|$$

Given the above definition of *suspiciousness*, there can be many ties since suspiciousness values will always be whole integers in the range $[0...|F|]$, where $|F|$ is the total number of failing runs. To break ties, we use the *Tarantula* approach [9] that considers the number of failing versus successful test cases in a test suite that exercise each statement. We chose *Tarantula* as our method of breaking ties for several reasons. First, it is very quick because it considers only statement coverage information. Second, it has been shown [9] to be more effective in fault localization on the Siemens benchmarks [8] than either *cause transitions* [2] or *nearest neighbor* [15]. Finally, the *Tarantula* approach is complementary to our approach. Tarantula considers statement coverage information from failing and successful tests, whereas our IVMP approach looks for statements which can be shown to correct the output of failing runs by using alternate value mappings. In the *Tarantula* approach, a statement is more suspicious if it is exercised more often by failing runs than by successful runs. The $suspiciousness_{tarantula}$ of a statement s is defined [9] as follows.

$$suspiciousness_{tarantula}(s) = \frac{\frac{failed(s)}{totalFailed}}{\frac{passed(s)}{totalPassed} + \frac{failed(s)}{totalFailed}}$$

In this equation, the variables $failed(s)$ and $passed(s)$ respectively refer to the number of failing and successful runs exercising statement s. The variables $totalFailed$ and $totalPassed$ respectively refer to the total number of failing and successful runs (test cases).

Our overall approach for ranking program statements using IVMPs is therefore composed of two main steps: (1) compute IVMPs from multiple failing runs; (2) rank statements in decreasing order of *suspiciousness*, breaking ties in decreasing order of $suspiciousness_{tarantula}$. However, computing IVMPs for multiple failing runs in the first step can be very time-consuming: for each failing run, each executed statement instance must be considered for an alternate value mapping replacement using all alternate mappings for that statement given in the value profile. To make our approach more practical, we limit our search for IVMPs in the first step so that we do not need to consider all statement instances from all failing runs. At each statement instance we do consider, we do not need to consider all alternate mappings in the value profile. As shown in our complete IVMP based statement ranking approach in Fig. 6, we accomplish this as follows.

```
input:
    Faulty program $P$, and test suite $T$ containing
    a set $F$ of failing runs.
output:
    A ranked list of statements exercised by tests in $F$.
algorithm IVMPBasedStatementRank
begin
Step 1: [Compute IVMPs for each test in $F$]
1:  $valProf$ := construct value profile for $P$ wrt. $T$;
2:  sort the tests in $F$ in increasing order of trace size;
3:  $workingList$ := the set of stmts exercised by the
        first failing test case in sorted $F$;
4:  for each test $f$ in $F$ taken in sorted order do
5:      $trace_f$ := stmt instances executed by $f$;
6:      for each stmt instance $i$ in $trace_f$ do
7:          $s$ := the stmt associated with instance $i$;
8:          if $s$ not in $workingList$ then continue;
9:          $altMap$ := alt. mappings for $s$ in $valProf$;
10:         $altMap_{red}$ := subset of $altMap$ with min/max
                values $<$ and $>$ the orig values used at $i$;
11:         for each alt. mapping $m$ in $altMap_{red}$ do
12:             if $s$ has an IVMP in $f$ then break;
13:             if applying $m$ at $i$ corrects $f$'s output then
14:                 report a found IVMP at $s$ in $f$;
            endfor (each alt mapping)
        endfor (each stmt instance)
15:     if $f$ has at least one IVMP then
16:         remove stmts from $workingList$ that are not
                associated with any IVMP in $f$;
    endfor (each failing run)
Step 2: [Use IVMPs to rank program statements]
17: $stmts$ := set of stmts exercised by tests in $F$;
18: for each stmt $s$ in $stmts$ do
19:     compute $suspiciousness(s)$;
20:     compute $suspiciousness_{tarantula}(s)$;
    endfor
21: $stmts_{ranked}$ := sort $stmts$ by $suspiciousness$,
        break ties by $suspiciousness_{tarantula}$;
22: output $stmts_{ranked}$;
end IVMPBasedStatementRank
```

Figure 6: Our IVMP based statement ranking approach using a reduced IVMP search.

Ordering failing runs. We first construct the value profile from the provided test suite (line 1). Then, we sort all failing runs in increasing order of trace size, where trace size is the number of statement instances in the trace (line 2). We consider each failing run in sorted order (line 4), while maintaining a working set of all statements that need to be searched for IVMPs in the currently-considered failing run. The working set is initialized to all statements exercised by the first failing run (line 3). After the search for IVMPs in the current failing run completes, then if no IVMPs are found, the working set remains unchanged. If at least one IVMP is found, then any statements in the working set that are not associated with any IVMPs identified from the current failing run are removed from the working set (lines 15-16). This working set of statements represents those statements for which all previously-considered failing runs yielded IVMPs. These statements are the most likely to be faulty. Although the first-considered failing run needs to have all statements searched for IVMPs, the search of the subsequent failing runs can be significantly reduced if IVMPs are found at relatively few statements in the first failing run. Moreover, by considering failing runs in increasing order of trace size, this ensures that the first-considered failing run is the smallest from among those available.

Limiting statement instances and alternate mappings. When searching for IVMPs in a particular failing run, only the statement instances from those statements that are in the working set are considered (line 8). At each considered statement instance, we apply only those alternate mappings for which the original value of a used or defined variable at that instance would be changed to be one of the following four alternate values: (1) the minimum alternate value less than the original value; (2) the maximum alternate value less than the original value; (3) the minimum alternate value greater than the original value; and (4) the maximum alternate value greater than the original value (line 10). These four particular alternate values span the range of alternate values for each variable, and may be far fewer than the total number of alternate values present in the value profile. Our experience also suggests that these four alternate values are highly likely to reveal an IVMP if IVMPs can in fact be found at the particular statement. This reduces the number of alternate mappings to apply at each statement instance to be a small constant, while still retaining a significant chance of finding IVMPs where they exist. Additionally, whenever an IVMP is found in a failing run, then all subsequent instances of that statement in the failing run need not be searched for further IVMPs (line 12). After identifying IVMPs, the statements exercised by the failing runs can be ranked (lines 17–21).

The total number of required program re-executions when applying alternate value mappings to compute IVMPs is bounded by $O(f \times t \times m)$, where f is the number of failing runs, t is the size of the shortest failing run execution trace, and m is the maximum number of alternate mappings to apply at any given statement. However, our approach reduces m to a small constant. Our experience also suggests that relatively good fault localization results can occur with a small f, such as a few failing runs. As a result, the runtime of our approach is mostly influenced by t.

4. EXPERIMENTAL STUDY

4.1 Setup

Implementation details. Our implementation uses the *Valgrind* infrastructure [6], which provides a synthetic CPU in software and allows for dynamic binary instrumentation of an executable program. Valgrind comes with a set of tools to perform tasks such as debugging and profiling, but we have added our own tools to record definition/use tracing information and to apply alternate value mappings during program re-execution when searching for IVMPs. Valgrind allows for instrumentation at the granularity of machine code instructions, so our implementation records traces in terms of instruction instances, and applies alternate mappings at the instruction level. Instructions are then mapped back to their corresponding statements (source code line numbers) when needed. Note that when an alternate value mapping is applied in our implementation, we actually overwrite the original values in their respective memory or register locations. As a result, any subsequent uses of those locations in later instructions will involve the new, alternate values.

Our experiments were run on a Dell PowerEdge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz, and 16 GB of RAM. Although the task of searching for IVMPs can be parallelized by performing multiple re-executions simultaneously, we did not parallelize the search when conducting our experiments so that we could give conservative timing results. A parallelized search can provide significantly improved timing results.

Subject programs and test suites. The Siemens programs [8] listed in Table 1 are used for our experiments. The programs, along with their corresponding faulty versions and test case pools, were obtained from [7]. All Siemens faulty versions contain seeded faults. These faults are computation-related (as opposed to memory-related), involving fault types such as operator and operand mutations, missing and extraneous code, and constant value mutations. Most faulty versions are seeded with a single fault in a single statement, but some faulty versions involve modifying several statements. We excluded a few faulty versions because they did not yield any failing test cases from the provided test pools. We also excluded one of the faulty versions from `printtokens2` because the fault caused execution to loop indefinitely far past the end of a string, causing traces to be very long and our Valgrind tracing tool to run out of memory.

Prog. Name	LOC	# Ver.	Avg. Suite (Pool) Sizes	Program Description
tcas	138	41	17 (1608)	altitude separation
totinfo	346	23	15 (1052)	statistic computation
sched	299	9	20 (2650)	priority scheduler
sched2	297	9	17 (2710)	priority scheduler
ptok	402	7	17 (4130)	lexical analyzer
ptok2	483	9	23 (4115)	lexical analyzer
replace	516	31	29 (5542)	pattern substituter

Table 1: The Siemens benchmark programs. From left to right: program name, # lines of code, # faulty versions, average suite size (and test case pool size), and description of program functionality.

The `tcas` program contains no loops and represents one big conditional check spread across several functions; it takes as input a set of integer parameters and reports one of three output values (or an error message if too few input arguments are specified). `totinfo` reads a collection of numeric data tables as input and computes statistics for each table as well as across all tables. Programs `sched` and `sched2` are priority schedulers for processes, taking as input a number of processes of various priorities as well as a list of scheduling commands, and outputting the processes as they complete in priority order. Programs `ptok` and `ptok2` are lexical analyzers, parsing an inputted character stream into a list of corresponding tokens. `replace` performs pattern substitution, taking as input a source pattern, destination pattern, and character stream, and replacing all instances of the source pattern in the character stream with the destination pattern.

For each faulty version of each program, we created a branch coverage adequate test suite as follows. We randomly selected a test case from the associated test case pool as long as it increased the cumulative branch coverage of the tests in the suite selected so far. We repeated this process until the created suite achieved the same level of branch coverage as the associated test case pool. We ensured that the created test suites contained at least 5 test cases that resulted in failing runs and at least 5 test cases that resulted in successful runs (if available), to ensure a good mix of failing and successful test cases in each suite.

Once a test suite was selected, we reduced the inputs of failing test cases by removing portions of the failing inputs so long as their actual (incorrect) outputs remained the same. We did this to reduce trace sizes by removing portions of the failing traces that were clearly not associated with the fault. This allowed us to compute IVMPs more quickly. Although this step may have changed the expected (correct) outputs of the test cases, the Siemens programs come with “base versions” that are assumed to be correct, which we used as test oracles to determine the expected outputs of the test cases.

Approaches and scoring. In our experiments we compare the fault localization effectiveness of the following two approaches that rank program statements.

1. IVMP approach. This is the technique from Fig. 6 where ranking is based upon the *suspiciousness* formula using a reduced IVMP search, and ties are broken using the $suspiciousness_{tarantula}$ formula from the Tarantula approach [9]. Here we use all available failing runs in the test suites to compute IVMPs (5 failing runs in most cases).

2. Tarantula approach. We also rank statements using only the $suspiciousness_{tarantula}$ formula, which has been shown [9] to be quite effective and, to our knowledge, provides the best overall fault localization results previously known using the Siemens benchmarks.

For comparison with the above two approaches, we also rank statements according to the following variations.

1. Tarantula-Pool approach. This is the same as the Tarantula approach, but here we considered each test suite to be the entire test case pool (rather than the much smaller branch-adequate suites used in the Tarantula approach). This is to study whether Tarantula is more effective when larger test suites are used.

2. IVMP-1 approach. This is the same as the IVMP approach, but here we rank statements by considering only one failing run when searching for IVMPs in each test suite (rather than by considering all failing runs in the suite as is done by the regular IVMP approach). This is to study the effectiveness of the approach when we do not consider multiple failing runs.

3. IVMP-2 approach. This is the same as the IVMP approach, but here we rank statements by considering just two failing runs when searching for IVMPs in each test suite.

In our experiments, we rank only those program statements that are executed by failing runs according to each of the above approaches, using the test suite associated with each faulty version of each subject program. In the event that multiple statements are tied for a particular rank, all tied statements are given a rank value equal to the maximum rank value from among the tied statements. For example, if there are 5 statements tied for highest rank, then all 5 of them are given rank 5. This allows us to conservatively assume that we would have to examine all tied statements before any faulty statement within that tied set can be found.

To evaluate each approach we assign a *score* to each ranked set of statements that is the percentage of program statements executed by failing runs in the test suite that *need not be examined* if statements are examined in rank order. Suppose that for a ranked list of statements S, the faulty statement occurs at rank r and there are a total of $totalStmtsEx$ total statements exercised by failing runs in the test suite. A rank value of 1 means that the faulty statement is the first statement in the ranked list and there are no ties (the ideal situation). Then the *score* of ranked statement list S can be defined as follows.

$$score(S) = \frac{totalStmtsEx - r}{totalStmtsEx} \times 100\%$$

A higher score is preferable because it means that more of the statements executed by failing runs are ignored before the faulty statement is found.

Finally, there are a few special considerations that we make in our experiments for certain faults. First, faults in constant assignment statements (15 out of a total of 129 faulty versions) will not result in any IVMPs at precisely those constant assignments. However, we can find IVMPs at the statements using those defined constants. We consider a constant assignment fault to be examined if we examine either the statement where it is defined, or else a statement where that constant value is used. Second, faults that involve omitted statements (16 out of 129 faulty versions) will mean that we cannot actually examine the statements that are missing. However, we can examine statements that are adjacent to the location where the code is missing, such that those statements would have influenced or would have been influenced by the missing code.

4.2 Results and Discussion

Effectiveness. Our experimental results are shown for each of the statement ranking approaches in Table 2 and Fig. 7. In Table 2, we show the number (and percentage) of faulty versions in which each approach computes a ranked list of statements in the specified score range. Fig. 7 shows a graphical view of this data. In the graph, the x-axis represents the lower bound of each score range, and the y-axis represents the percentage of faulty versions achieving a score greater than or equal to that lower bound. Percentages are computed with respect to 129 faulty versions from among the Siemens programs. This presentation of data follows the convention of Jones et al. [9]. However, whereas [9] computes scores with respect to the total number of program statements, we compute scores with respect to the total number of statements exercised by failing test cases in the suite. This is because faulty statements can only be from among these exercised statements.

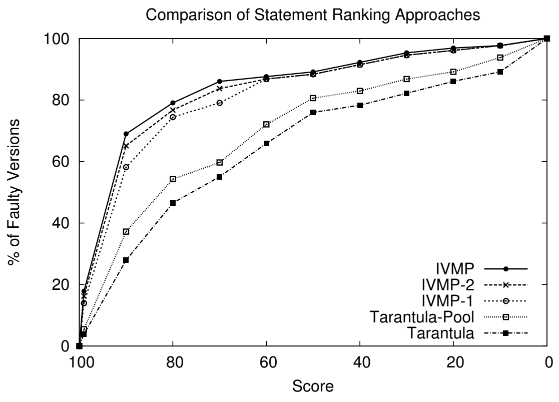

Figure 7: Comparison of statement ranking approaches

IVMP approach versus Tarantula approach. The data shows that the IVMP approach overall performs much better than the Tarantula approach. Almost 18% of the faulty versions analyzed had a score of 99% or higher with the IVMP approach, whereas the same was true for only about 4% of the faulty versions using Tarantula. Similarly, almost 70% of faulty versions had a score of 90% or higher using the IVMP approach, while the same was true for about 28% of the faulty versions using Tarantula. Among all 129 faulty versions, the IVMP approach was able to uniquely identify the faulty statement (assign it rank 1) in 39 cases. Tarantula was able to do so in only 5 cases. Note that even though

Score	Tarantula approach	IVMP approach	Tarantula-Pool approach	IVMP-1 approach	IVMP-2 approach
99-100%	5 (3.88%)	23 (17.83%)	7 (5.43%)	18 (13.95%)	21 (16.28%)
90-99%	31 (24.03%)	66 (51.16%)	41 (31.78%)	57 (44.19%)	63 (48.84%)
80-90%	24 (18.60%)	13 (10.08%)	22 (17.05%)	21 (16.28%)	15 (11.63%)
70-80%	11 (8.53%)	9 (6.98%)	7 (5.43%)	6 (4.65%)	9 (6.98%)
60-70%	14 (10.85%)	2 (1.55%)	16 (12.40%)	10 (7.75%)	4 (3.10%)
50-60%	13 (10.08%)	2 (1.55%)	11 (8.53%)	2 (1.55%)	2 (1.55%)
40-50%	3 (2.33%)	4 (3.10%)	3 (2.33%)	4 (3.10%)	4 (3.10%)
30-40%	5 (3.88%)	4 (3.10%)	5 (3.88%)	4 (3.10%)	4 (3.10%)
20-30%	5 (3.88%)	2 (1.55%)	3 (2.33%)	2 (1.55%)	2 (1.55%)
10-20%	4 (3.10%)	1 (0.78%)	6 (4.65%)	2 (1.55%)	2 (1.55%)
0-10%	14 (10.85%)	3 (2.33%)	8 (6.20%)	3 (2.33%)	3 (2.33%)

Table 2: Number (percentage) of faulty version ranked statement lists in each score range for all approaches.

the IVMP approach was able to uniquely identify the faulty statement in 39 cases, only 23 cases yielded scores of 99% or more. This is because in the `tcas` program, the number of statements exercised by failing test cases was few enough that even a rank of 1 would lead to a score less than 99%.

Out of 129 faulty versions in our experiments, there were only 16 cases where the IVMP approach assigned a lower rank to a faulty statement than Tarantula. These cases occurred where IVMPs happened to be found at non-faulty statements in more failing runs than at faulty statements, giving the non-faulty statements higher rank. However, in many of these cases, the non-faulty statements with higher rank were still near to the faulty statements via dependence edges (recall the *dependence cause* for IVMPs at multiple statements described earlier). In 18 other faulty versions, the IVMP approach and Tarantula gave the faulty statement identical ranks. In some of these cases, this was due to finding no IVMPs in any failing runs, causing *suspiciousness* values to be identical and ranking to be done solely by breaking ties using $suspiciousness_{tarantula}$. In the remainder of the cases (95 of them), the IVMP approach gave the faulty statement higher rank than Tarantula due to IVMPs being found at the faulty statement.

Comparison with other approaches. The results for the Tarantula-Pool approach indicate that fault localization is indeed more effective for Tarantula when larger test suites are used. However, Tarantula-Pool is still considerably less effective overall on the Siemens programs than the IVMP approach. In fact, Tarantula-Pool is also considerably less effective than either the IVMP-1 or the IVMP-2 approaches, which both consider fewer failing test cases when searching for IVMPs than the regular IVMP approach. However, as might be expected, IVMP-2 is slightly less effective overall than the IVMP approach, while the IVMP-1 approach is also slightly less effective than the IVMP-2 approach. Thus, the IVMP approach is generally more effective as more failing runs are considered, but even when considering just a single failing run, the IVMP-1 approach is still more effective than the Tarantula-Pool approach that considers statistical information taken from very large test suites.

Efficiency and Timing. Recall from Fig. 6 that when we search for IVMPs for fault localization purposes, we use a reduced search that considers only a subset of instruction instances from among the failing runs, and a subset of alternate value mappings to apply at each considered instance. To study how effective we are in reducing the IVMP search according to our algorithm, for each faulty version we compute the total number of program re-executions that would be required if we fully search every instruction instance of every failing run and apply every associated alter-

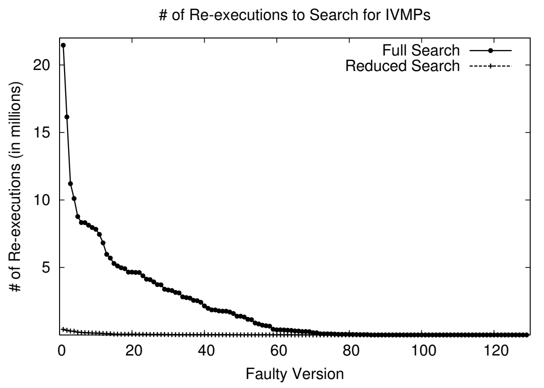

Figure 8: For each faulty version, the number of re-executions (in millions) required for the full and reduced IVMP searches in the IVMP approach.

nate value mapping from the value profile at each instance. We then compare these values to the number of program re-executions actually performed when searching for IVMPs using our reduced search in the IVMP approach. We sort the faulty versions in decreasing order of these number of re-executions, and present the data shown in Fig. 8.

As shown in this figure, the total number of re-executions actually performed using our reduced IVMP search was significantly lower than what would have been required if we had fully searched for all possible IVMPs. For the full search, four faulty versions would have required over 10 *million* program re-executions each to fully search for IVMPs (one of these faulty versions would have required over 20 million program re-executions). On the other hand, the maximum number of re-executions required for any faulty version using our reduced search was only about 412,000. A large majority of the cases (84 of them) required fewer than 10,000 program re-executions to search for IVMPs using our reduced search. The same was true for only a minority of cases (44 of them) using the full search. On average across all faulty versions, the full search requires over 2 million program re-executions per faulty version, while the reduced search requires just under 30,000 program re-executions. However, note that these average values are made large due to a few faulty versions that require unusually many program re-executions. In general, the few cases where our reduced search required relatively more program re-executions than other reduced-search cases, was due to faulty versions where our approach was not able to find any IVMPs. In these cases, we fully searched all instruction instances of all failing runs since no prior IVMPs were found in earlier-considered failing runs.

We have shown a drastic reduction in the number of program re-executions required to search for IVMPs using our

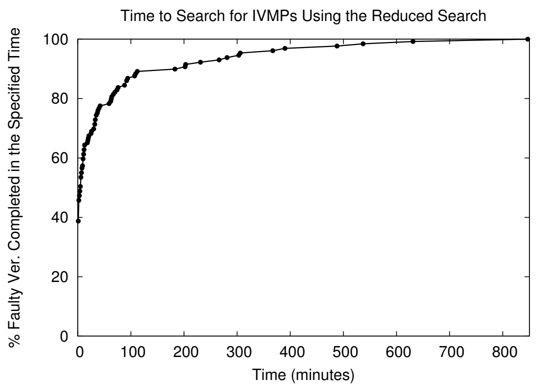

Figure 9: The percentage of faulty versions in which our reduced search for IVMPs is able to complete in the specified amount of time in the IVMP approach.

reduced search as compared to a full search. In Fig. 9, we show the actual time required to search for IVMPs using the reduced search in the IVMP approach. The x-axis represents the time in minutes to search for IVMPs from all failing runs using our reduced search. The y-axis shows the percentage of faulty versions that were fully searched in less than the specified amount of time. Note that the actual time to rank statements with the computed IVMP information (and breaking ties with the Tarantula formula) is negligible compared to the IVMP search time. Computation of the value profiles for each faulty version was very small as compared to the IVMP search time, never taking more than a few dozen seconds per faulty version.

From this figure, it can be seen that most faulty versions required relatively little time to search for IVMPs using our reduced search. 50 faulty versions (many from the tcas program) require less than 1 minute to search all failing runs for IVMPs. A large majority of cases (77 of them) require less than 10 minutes of search time. Almost 90% of cases (112 of them) require less than 100 minutes. Only 17 out of the 129 faulty versions actually require more than 100 minutes of search time. The maximum required time to search was just over 14 hours (840 minutes) for one particular faulty version, but this was an unusual case where failing runs were relatively long and no IVMPs could be found to limit the IVMP search. Searching for IVMPs requires more computation time than just applying Tarantula's formula using the statement coverage information of tests. However, the time required by our approach is reasonable since it is fully automated and is meant for a debugging context where a developer may be stuck for quite awhile looking for the location and cause of a fault without any guidance.

Other Observations. Program totinfo is an unusual case among the Siemens programs. For this particular subject program, only 8 faulty versions resulted in the IVMP approach performing better than the Tarantula approach. 5 cases had the IVMP approach performing worse, and 10 cases had both approaches performing equally well. These results were generally not as good as the results from the other Siemens programs, in which the IVMP approach usually performed much better as compared to Tarantula. We found out that for totinfo, relatively few IVMPs were found as compared to the other Siemens programs. This is due to the fact that totinfo performs floating-point computations. Thus, it is very difficult to cause output to change to become precisely correct when alternate value mappings are applied.

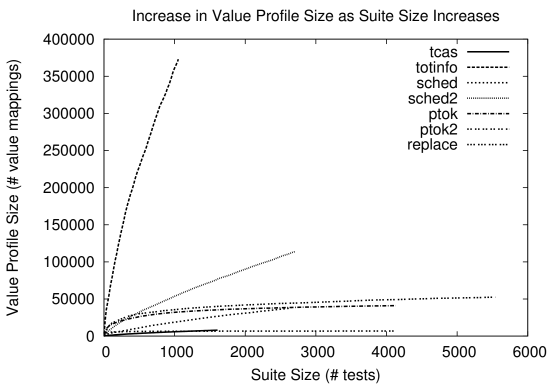

Figure 10: Increase in value profile size as suite sizes increase, for each subject program.

We mentioned earlier in Section 2 that the sizes of the value profiles for each faulty version seem to increase logarithmically as the number of test cases in the suites increase. To analyze this, we constructed value profiles for five arbitrarily-chosen faulty versions from each Siemens program using tests from the available test case pools. The results for each faulty version in a subject program were then averaged and plotted as shown in Fig. 10.

As can be seen in the figure, the curves for most subject programs become nearly horizontal over time as more test cases are considered in the value profile. One notable exception is for program totinfo, which has a curve that increases much higher than that of all the other programs. This is because totinfo uses many floating-point values, which are highly likely to be different from test case to test case.

4.3 Larger Subject Programs

We conducted some additional experiments to see whether IVMPs can still be computed in reasonable time for larger subject programs. These programs are shown in Table 3. We selected one fault from each program to study. We selected these particular faults because we were able to find test cases to expose them using execution traces that were not too long. Program space (obtained from [7]) contains a known bug in which a condition ($error\ != 0$) should instead be ($error == 17$). The grep program contains a known bug in which using parameters $-i$ and $-o$ simultaneously may lead to incorrect output. Programs sed, flex, and gzip contain seeded faults [7] of the following respective types: a "-1" term is missing from an expression; command-line parameters are incorrectly processed; input files with improper file names are incorrectly processed.

Prog. Name	LOC	Fault Type	Program Description
space	6199	real	ADL interpreter
grep-2.5	5812	real	pattern matcher
sed-4.1.5	12972	seeded	stream editor
flex-2.5.1	10013	seeded	lexical analyzer generator
gzip-1.3	5166	seeded	file compressor

Table 3: Larger subject programs.

For these experiments, we followed a similar setup as for the Siemens benchmark programs, except here we did not create branch-coverage adequate test suites since the programs are much larger. Instead, each suite consists of a few failing runs and several (5 – 6) successful runs. Our experimental results using the larger programs are given in Table 4.

Program Name	Faulty Stmt Rank	IVMP Search Time	# Re-executions Done/Possible for IVMPs
space	Tarantula: 106 IVMP: 5	79.5 min	35841/1061154 (3.4%)
grep-2.5	Tarantula: 213 IVMP: 3	0.8 min	241/588 (41.0%)
sed-4.1.5	Tarantula: 35 IVMP: 3	1.8 min	881/5816 (15.1%)
flex-2.5.1	Tarantula: 45 IVMP: 1	0.5 min	87/228 (38.2%)
gzip-1.3	Tarantula: 96 IVMP: 1	215.6 min	126845/6918816 (1.8%)

Table 4: Experimental results using the larger programs (one fault and test suite per program).

For each program, the rank of the faulty statement is shown for each of the Tarantula and IVMP approaches (recall: the rank of a statement is its position – worst-case position if there are ties – in the ordered list of executed statements). The timing and required re-execution counts for computing IVMPs in the IVMP approach is also provided in the table.

From these results, it can be seen that the IVMP approach was able to take advantage of IVMPs to demonstrate significant improvements in fault localization effectiveness over Tarantula. Moreover, for the `grep`, `sed`, and `flex` subjects, IVMP search times were quite low due to the fact that we were able to find failing runs with very short execution traces. For the `space` and `gzip` programs, IVMP search times were comparatively longer due to longer failing traces. However, these timing results still seem reasonable in an automated debugging context especially considering the significant improvement in rank of the faulty statements using the IVMP approach.

We observe that even though the above programs themselves are significantly larger than the Siemens programs, the IVMP search times are still very similar for both sets of programs (from a few minutes to several hours). This illustrates that it is not the program size that determines the runtime of our approach, but rather, it is the length of the shortest failing trace. If failing traces can be reduced to be very small, then IVMP search times will be relatively low regardless of the original program size.

4.4 Other Points of Discussion

Scalability. One of the major questions about the IVMP approach is whether it can scale to large programs. We have begun addressing this issue by reducing the search for IVMPs, which is the step of our approach that requires the most computation time. We have also argued that it is the execution trace sizes that determine the runtime of our approach, not the program sizes. Thus, our approach can be useful on large programs in certain cases. However, there is clearly more work to do to improve the scalability of our approach. One path we are pursuing is to combine the IVMP approach with other techniques that can further limit the IVMP search. For example, we can use *program slicing* to find the statements that can influence the incorrect output of a failing run, and only search for IVMPs in those statements. Another option is to find ways to improve the efficiency of program re-executions when searching for IVMPs. For instance, rather than completely executing a program from scratch every time an alternate value mapping is applied, we may be able to use a checkpointing and logging scheme to effectively "skip" the initial parts of certain executions.

Multiple Simultaneous Faults. Some of the Siemens faulty versions contain multiple faults. In these cases, we were still able to find IVMPs for at least one of the faults and achieve statement ranking results that localize the fault. However, in general it may be difficult to find IVMPs in programs with multiple faults that influence each other or that have different effects on program output. Thus, the presence of multiple faults can diminish the effectiveness of our approach.

Address Values. Address values are currently ignored in our approach. This is because address values from different test case executions cannot simply be blindly substituted into a particular failing run, since address values are execution-specific and have no meaning outside of a given execution. As a result, our current approach may have limited effectiveness for memory-related faults.

5. RELATED WORK

Delta debugging [2, 19, 20] is a debugging framework that focuses on studying the differences between successful and failing runs to aid in debugging. The approach identifies failure-inducing input and isolates the differences between a successful and failing test case [20]. Cause-effect chains of failures in a program run can then be computed by identifying the chain of variables and values that caused the failure [19]. *Cause transitions* in these chains, which are points in time in which a variable ceases to be a cause for a failure and another such variable begins, can then be identified to provide potential links to the defects in the code that caused the failure [2]. This work is similar to ours because it involves altering program state (the values of variables) to try to isolate the variables that cause a failure. Our approach also alters program state, but at a finer granularity: only the values used in a particular statement instance in a failing execution are altered at any given time in our approach.

The work of Alex Groce [3], partly inspired by *delta debugging*, describes an approach for helping to automate the process of locating and explaining faults using *distance metrics*. This approach is similar to the *nearest neighbor* approach described by Renieris and Reiss [15] that searches for a correct execution that is most similar to an incorrect execution, compares the spectra for these two executions, and identifies the most suspicious parts of the program. *Nearest neighbor* was previously shown [2] to perform more poorly than *cause transitions* at fault localization.

Program slicing [16, 18] was proposed to identify a subset of program statements that can influence the value of a variable at some program location. While early work proposed construction of static slices, later work developed dynamic slices [1, 10] that are constructed with respect to a particular program run. To take potential influence into account, relevant slicing was proposed [11]. Construction of smaller dynamic slices by intersecting multiple slices was studied [4, 21]. To further order the statements in a dynamic slice according to their likelihood of being faulty, confidence analysis was proposed [22]. Our approach does not just consider which statements can affect incorrect program output, but it actually searches for ways to show how the values used at these statements can be altered so that the incorrect output becomes correct. Our work therefore goes beyond slicing as IVMPs may better pinpoint faulty statements and have potential to be more helpful to the user in understanding the cause of a bug and how to fix it.

A *statistical* approach to isolating bugs in programs has been described [12, 13] that uses sampling during program execution to collect data and identify *bug-predicting predicates* that are associated with individual bugs. Another statistical approach [14] incorporates the use of models to analyze the evaluation patterns of predicates in successful and failing runs; a predicate is deemed to be relevant to a bug if its evaluation pattern in failing runs is significantly different from that in successful runs. An *invariant-based* approach was developed [5] that formulates program invariants while a program is running and dynamically monitors for violations of these invariants that can isolate the root cause of a fault. Unlike statistical approaches that may take limited information into account when narrowing the search for faulty code, our approach considers much more information in the form of alternate value mappings that can be applied at multiple statement instances in a failing run.

Work on *predicate switching* [21] suggests that analyzing "critical predicates" in software, which when forced to take an alternate outcome will cause a failing run to produce correct output, can help in locating faulty code. A similar work [17] automates the construction of a successful run using a failing run by trying to alter the outcomes of some conditional branches in the failing run. While the above techniques merely alter branch outcomes, the approach we have proposed in this paper alters values of a subset of program variables. Switching the outcome of a predicate may not be able to produce the correct output while a change in the values of a subset of variables may be able to do so. In this sense, *predicate switching* is subsumed by our IVMP detection approach. Also, since faults may be present in statements that do not involve predicates, then *predicate switching* requires some way to link "critical predicates" to the faulty code. The IVMP approach makes no distinction between predicate and non-predicate statements, so IVMPs can be directly found at faulty statements that may not involve predicates. Moreover, our approach can present a developer with a set of identified IVMPs which can make it easier to understand a bug – a simple predicate switch provides a limited amount of information.

6. CONCLUSIONS

In this paper, we presented a value profile based approach for fault localization to assist developers in the task of software debugging. The approach involves computing IVMPs that show how values used at particular program statements can be altered so that failing runs instead produce correct output. Using these IVMPs, executed statements can then be ranked according to their likelihood of being faulty. Our experimental results have shown that for the benchmark programs we studied, our IVMP approach can produce ranked lists of statements that are generally very effective at quickly pointing to a faulty statement. Moreover, our approach was seen to be more effective than a prior fault localization approach that, to our knowledge, had yielded the best results known up until now for the benchmark programs in our study. We have also shown that the time required to perform our approach is reasonable in a debugging context.

Acknowledgements. We would like to thank the anonymous reviewers for their valuable feedback. This research is supported by NSF grants CNS-0751961, CNS-0751949, CNS-0810906, and CCF-0753470, to UC Riverside.

7. REFERENCES

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. *ACM 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 Conference on Software Engineering*, pages 342–351, May 2005.

[3] A. Groce. Error explanation and fault localization with distance metrics. *Ph.D. Thesis, CMU*, 2005.

[4] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. *IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, November 2005.

[5] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *International Conference on Software Engineering*, pages 291–301, May 2002.

[6] http://valgrind.org.

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

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and controlflow-based test adequacy criteria. *International Conference on Software Engineering*, pages 191–200, May 1994.

[9] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[10] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, Oct. 1988.

[11] B. Korel and J. Laski. Algorithmic software fault localization. *Annual Hawaii International Conference on System Sciences*, pages 246–252, January 1991.

[12] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. *Conference on Programming Language Design and Implementation*, pages 141–154, June 2003.

[13] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. *Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.

[14] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: Statistical model-based bug localization. *European Software Engineering Conference held jointly with the International Symposium on Foundations of Software Engineering*, pages 286–295, Sept. 2005.

[15] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *18th IEEE International Conference on Automated Software Engineering*, pages 30–39, Oct. 2003.

[16] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[17] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. *International Conference on Automated Software Engineering*, pages 347–351, Nov. 2005.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[19] A. Zeller. Isolating cause-effect chains from computer programs. *International Symposium on the Foundations of Software Engineering*, pages 1–10, Nov. 2002.

[20] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[21] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *28th International Conference on Software Engineering*, pages 272–281, May 2006.

[22] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.