Computing and Informatics, Vol. 34, 2015, 823–857

STATISTICAL BASED SLICING METHOD FOR PRIORITIZING PROGRAM FAULT RELEVANT STATEMENTS

Saeed Parsa, Mojtaba Vahidi-Asl, Farzaneh Zareie

Faculty of Computer Engineering
Iran University of Science and Technology
Narmak, Tehran, Iran
e-mail: {parsa, m_vahidi_asl}@iust.ac.ir,
farzaneh_zareie@comp.iust.ac.ir

Abstract. The aim of this paper is to integrate the strong points of statistical debugging and program slicing techniques for efficient program fault localization. The dynamic slices could be inappropriately large including redundant information considering that the statements are not ranked according to their fault relevance in the computed slice. On the other hand, the conventional statistical debugging techniques do not consider the dependence relationships between the faulty code (i.e., the cause) and the erroneous output (i.e., the effect). This information is very useful during the debugging. In this paper, a new method Stat-Slice for locating the latent bugs in programs is presented which could find a wider range of bugs, e.g. the omitted code faults, the header file faults, and etc., comparing with other techniques. Unlike the traditional program slicing techniques, the proposed method computes the backward dynamic slices of several failing and passing runs. Using K-means clustering in addition to a new ranking and pruning technique, we prioritize statements according to their likelihood to be the cause for failure. Our experiments on Siemens, grep, gzip, and flex test suites manifest that ranking statements according to their suspiciousness has considerably reduced the effort for fault localization.

Keywords: Fault localization, dynamic slicing, statistical debugging, clustering, Pearson, fault

Mathematics Subject Classification 2010: 68-N99

1 INTRODUCTION

Producing a bug free program is the ideal goal of software companies. Even sophisticated testing teams with best efforts cannot test all feasible execution paths of a program due to its large size and complexity. Therefore, some latent bugs go undetected which may cause crashes or unexpected results. The existence of undetected bugs in the released software programs necessitates the process of fault localization to make the software more robust and reliable. Due to the intricacy and inaccuracy of manual fault localization, a vast amount of research has been done to develop automated techniques which assist developers in finding bugs [1, 3, 4, 5, 6, 7, 8].

The question is how to find the origin of failure by considering the program incorrect output(s). If there is a control/data dependence between the faulty statement(s) and the output statement, the backward dynamic slice, namely BDS, of the output in a single failing execution may contain an execution instance of the faulty statement [1, 14, 15]. Therefore, a backward dynamic slicing technique could be helpful to find the origin of failure if it is located in the slice. However, since the size of BDS could be large and the statements in the slice are not ranked according to their relevance with the failure, a programmer should spend a considerable effort to examine the statements in the slice with an equal chance to find the origin of failure [9, 14, 16, 17]. In order to rank the statements of a slice, in [15] a pruning based approach is introduced which gives a confidence value to the statements included in BDS according to their likelihood of being faulty. However, it is applicable when a program has multiple output values where, prior to an incorrect output value, some other outputs produce correct results. In some situations, the faulty statement(s) is not included in the BDS [2]. Therefore, some other statements which are not included in the BDS but have influence on the unexpected result should also be considered by the programmer in locating the fault. Although relying on a single failing run is the main advantage of the slicing techniques, they can merely find a fault that is related to the corresponding incorrect output, and hence they are incapable to find other existing fault(s) in a program.

To address the mentioned deficiencies, the traditional statistical debugging methods collect runtime data from a big number of runs and contrast the behavior of the correct and incorrect executions to locate faults [6, 7, 8, 12]. Typically, the runtime behavior is determined by evaluating simple Boolean expressions called predicates (e.g. the directions of branches, the results of function calls, assignment statements, etc.) at various program points [13]. A main advantage of the statistical debugging techniques is their ability to rank the predicates according to the likelihood of being relevant to the faulty code [6]. However, they may require a large number of passing and failing executions for their analysis [15]. Furthermore, since ranking is performed on predicates, in many cases the exact location of faults are not pinpointed by a statistical technique and only the fault relevant predicates, those predicates that manifest the effect of faults, are identified. Therefore, the programmer should scrutinize the code manually to find the origin of failure [21]. On the other hand, the conventional statistical debugging techniques do not consider the

dependence relationships between the faulty code (i.e., the cause) and the erroneous output (i.e., the effect). This information is very useful during the debugging.

In this paper, we propose the following novel approach, namely Stat-Slice. First, the backward dynamic slicing technique is applied on a small number of failing and passing executions. Then, each program run is converted into an execution vector, also called program spectrum, where each element of the vector corresponds to a specific statement of a given program. In each run, a particular statement could have three different states:

1. it is included in the BDS,
2. executed but not included in the BDS and
3. non-executed.

We expand the BDS by computing two proposed conditional probabilities to identify whether an executed statement that is not in the BDS, has the potential to impact the failing/passing result of the program. The result is an expanded backward dynamic slice, EBDS. These execution vectors are clustered according to their Euclidean distances to identify different failing and passing execution paths with the aim of pruning irrelevant data, reducing manual code scrutinization for pinpointing the failure origin, and finding multiple faults in the program. Finally, we compute the binary correlation of each statement with the failing and passing state of the program and rank them according to the computed fault relevance score. Stat-Slice is evaluated on faulty versions of Siemens, gzip, grep, and flex programs and the results manifest the high performance of the technique in localizing bugs. In summary the following contributions are made in this paper:

1. We rely on a fewer passing runs compared with traditional statistical techniques.
2. Stat-Slice considers those statements which have full data and control dependence with the program output result. Therefore, it can find a wider range of faults comparing with other fault localization techniques.
3. Each program execution is converted to an execution vector according to the computed BDS. The clustering of execution vectors helps to identify different execution paths. It helps locating multiple bugs in the program and reduces manual code inspection.
4. Stat-Slice detects 80 faults in Siemens suite with less than 10 percent manual code inspection. It also finds bugs with a very small amount of code examination in larger programs, gzip, grep, and flex.
5. The proposed expanded backward dynamic slice, EBDS, enables us to identify the faulty statements which are not included in the backward dynamic slice of the program output.

The remaining part of this paper is organized as follows. In Section 2, a motivating example is presented. An overview of the method including some definitions is described in Section 3. The experiments and results are shown in Section 4.

Some related works are presented in Section 5. Finally, the concluding remarks are mentioned in Section 6.

2 EXAMPLE

In this section, a motivating example is presented to demonstrate the capability of the proposed statistical fault localization technique along with the power and precision of backward dynamic slicing to pinpoint faults with high accuracy. Stat-Slice takes advantage of a statistical method to rank statements by contrasting the backward dynamic slices of failing and passing runs. The example provides an overall step by step description of Stat-Slice and demonstrates all stages of our proposed approach.

The program in Figure 1 a) receives two $2 * 2$ matrices, $x1$ and $x2$, and two flags, *flag*1 and *flag*2, as its input parameters. If the value of *flag*1 is positive, the program checks whether $x1$ and $x2$ are invertible. If they are both invertible, the program also checks whether one of them is the inverse of the other. If the value of *flag*2 is positive, it checks whether the multiplication matrix, $x3$, is invertible. We have seeded two different faults in the program. As shown, the first faulty statement is located in line 18 of the program in which the programmer by mistake has written $x1[0][1]$ instead of $x2[0][1]$. The second fault is in line 19 where $x1[0][0]! = 1$ is written instead of $x3[0][0]! = 1$.

These are the instances of semantic faults which cause failure (i.e. incorrect output) for specific input parameters. The output statement is in line 34. Figure 1 b) contains ten different test cases, specified by $TC\#1$ to $TC\#10$, and their corresponding backward dynamic slices computed from the output statement at line 34 and the statements included in the slice are marked by '$*$'. Because the output statement in line 34 is included in the BDS of all test cases, by default, it is not shown in the Figure 1.

Note that the passing test case $TC\#10$ is a coincidental correct test case [27] for which the execution of the faulty statement does not cause any program failure. Generally, coincidental correct test cases mislead the statistical fault localization techniques and should be removed from further analysis as mentioned below. As shown in the Figure 1, the BDS's for the test cases $TC\#7$ and $TC\#10$ are identical although they have different termination states. Since the BDS may contain the faulty statement(s), we mask the last passing test case with the failing one and exclude it from our further analysis.

It can be shown that the size of a backward dynamic slice is typically large according to the program size. In our example, for the faulty statements in lines 18 and 19, the backward dynamic slices for the failing test cases contain 23 lines of code, in average. Therefore, the programmer should manually examine a considerable portion of the code to locate the failure origin(s). The Stat-Slice ranking method assigns the highest score, i.e. 1, to the faulty statements and locates the failure origins without any need for manual inspections. In the remaining parts of this

```
      main()
      {
        int  i,j,k,x1[2][2],x2[2][2],x3[2][2],det1, flag1=0,flag2=0;

1-    for (i=0;i<2;i++)
2-      for (j=0;j<2;j++)
        {
3-        scanf("%d",&x1[i][j]);
4-        scanf("%d",&x2[i][j]);
        }
5-    scanf("%d",&flag1);
6-    scanf("%d",&flag2);

7-    if(!flag1 && !flag2)
8-      { print("no request from user!")
9-        return; }
10-   det1= x1[0][0]*x1[1][1]- x1[0][1]*x1[1][0];
11-   if (det1!=0) {
12-     for (i=0;i<2;i++)
13-       for (j=0;j<2;j++)
14-       {  x3[i][j] = 0;
15-          for (k=0;k<2;k++)
16-            x3[i][j]+=x1[i][k]*x2[k][j]; }

17-   if(flag1>0)
18-     if((x2[0][0]*x2[1][1]!=x1[0][1]*x2[1][0])  //it should be  x2[0][1] instead of x1[0][1]
19-       if(x1[0][0]!=1||x3[1][1]!=1||(x3[0][1]!=0||x3[1][0]!=0))  //it should be  x3[0][0]
                                                                    // instead of x1[0][0]
20-         print("Both matrices are invertible but x1 is not the inverse of x2");
21-       else
22-         print(("x1 is the inverse of x2");
23-     else
24-       print("x2 is not invertible");

25-   if(flag2>0)
26-     if(x3[0][0]*x3[1][1]!= x3[0][1]*x3[1][0])
27-       print("the multiplication result of x1 and x2 is invertible");
28-     else
29-       print("the multiplication result of x1 and x2 is not invertible");
      }
30-   else   \\ if determinant of x1 is zero
31-   print("x1 is not invertible");

32-   void print(char *str)
33-   {   // output statement
34-       printf("%s",str);
      }
```

a)

# line	TC # 1	TC # 2	TC # 3	TC # 4	TC # 5	TC # 6	TC # 7	TC # 8	TC # 9	TC # 10
	$X1=\begin{bmatrix}1&1\\1&0\end{bmatrix}$ $X2=\begin{bmatrix}1&1\\1&1\end{bmatrix}$ Flag1=0 Flag2>0	$X1=\begin{bmatrix}1&0\\-1&1\end{bmatrix}$ $X2=\begin{bmatrix}1&2\\1&1\end{bmatrix}$ Flag1=0 Flag2>0	$X1=\begin{bmatrix}1&0\\1&1\end{bmatrix}$ $X2=\begin{bmatrix}1&0\\2&0\end{bmatrix}$ Flag1>0 Flag2>0	$X1=\begin{bmatrix}1&0\\1&1\end{bmatrix}$ $X2=\begin{bmatrix}1&-1\\1&-1\end{bmatrix}$ Flag1>0 Flag2>0	$X1=\begin{bmatrix}-1&1\\0&-1\end{bmatrix}$ $X2=\begin{bmatrix}-1&1\\0&-1\end{bmatrix}$ Flag1>0 Flag2>0	$X1=\begin{bmatrix}1&1\\1&2\end{bmatrix}$ $X2=\begin{bmatrix}1&0\\1&1\end{bmatrix}$ Flag1>0 Flag2>0	$X1=\begin{bmatrix}1&-1\\0&1\end{bmatrix}$ $X2=\begin{bmatrix}-1&1\\0&1\end{bmatrix}$ Flag1>0 Flag2=0	$X1=\begin{bmatrix}1&1\\-1&0\end{bmatrix}$ $X2=\begin{bmatrix}1&1\\0&0\end{bmatrix}$ Flag1>0 Flag2=0	$X1=\begin{bmatrix}1&1\\1&1\end{bmatrix}$ $X2=\begin{bmatrix}1&0\\0&1\end{bmatrix}$ Flag1>0 Flag2=0	$X1=\begin{bmatrix}1&0\\1&-1\end{bmatrix}$ $X2=\begin{bmatrix}1&0\\1&-1\end{bmatrix}$ Flag1>0 Flag2=0
1	*	*	*	*	*	*	*	*	*	*
2	*	*	*	*	*	*	*	*	*	*
3	*	*	*	*	*	*	*	*	*	*
4	*	*	*	*	*	*	*	*	*	*
5	*	*	*	*	*	*	*	*	*	*
6	*	*	*	*	*	*	*	*	*	*
10	*	*	*	*	*	*	*	*	*	*
11	*	*	*	*	*	*	*	*		*
12	*	*	*	*	*	*	*	*		*
13	*	*	*	*	*	*	*	*		*
14	*	*	*	*	*	*	*	*		*
15	*	*	*	*	*	*	*	*		*
16	*	*	*	*	*	*	*	*		*
17			*	*	*	*	*	*		*
18				*	*		*			*
19				*	*					
20				*	*					
21							*			*
22							*			*
23			*			*		*		
24			*			*		*		
25	*	*	*	*	*	*				
26		*			*	*				
27		*			*	*				
28	*		*	*						
29	*		*	*						
30									*	
31									*	
T.S.	pass	pass	pass	fail	fail	fail	fail	pass	pass	pass

b)

Figure 1. a) A sample program code with two seeded faults. The faults are located in statement 18 and 19 and the result statement is in statement 34. b) Ten test cases in addition to the computed backward dynamic slices for each test case.

paper, we describe how Stat-Slice ranks the statements in Figure 1 and gives the highest score to the faulty statements.

3 OVERVIEW OF THE METHOD

In this section the main idea of the proposed Stat-Slice technique is described in detail. First some basic definitions are depicted in Section 3.1 and the detail of the proposed technique is indicated in Section 3.2.

3.1 Basic Definitions

In order to combine the idea of the backward slicing technique and the statistical fault localization methods, we assume that for each single failure in the program, there exist at least one failing run and a number of passing runs. Providing test cases is not a difficult task, since we could collect runtime data from end users or generate random test cases. To illustrate details of the proposed Stat-Slice technique, some basic definitions should be described at the first place.

Definition 1. Given program P and a set of test cases Tc_P: each test case tc_j consists of input parameters, I_j, and the corresponding desired output, O_j. After executing P on tc_j, $Exe_P(tc_j)$, the output result would be O'_j, $Exe_P(tc_j) = O'_j$. Test case tc_j is called failing test case, tc_j^{fail}, if: $O_j \neq O'_j$ and otherwise it is named passing test case, tc_j^{pass}. Therefore, the set of test cases in Tc_P is split into two disjoint categories: Tc_P^{pass} for the passing test cases and Tc_P^{fail} for the failing test cases.

Definition 2. Running program P with test case tc_k generates a dynamic dependence graph, $G_P^k(N, E)$, in which N is the set of nodes and E is the set of directed edges: Each node $S_l \in N$ represents the l^{th} execution instance of the statement S in the program and an edge from the node T_γ to S_τ stands for a dynamic data or control dependence from the τ^{th} execution instance of the statement S to the γ^{th} execution instance of the statement T.

In other words, each program statement may be executed several times in a single run. For instance, the number of executions of a statement in a loop body depends on the number of loop iterations. Therefore, it is important to know whether an execution instance of a statement relates to the failure. If an execution instance of the statement S uses the data provided from an execution instance of the statement T, there is a directed edge from T to S and we say that S is data dependent to T. Similarly, if the execution of S is a direct result of outcome of the statement T, we say S is control dependent to T and there is a directed edge from T to S. The nodes of $G_P^k(N, E)$ are also known as the code coverage for the test case tc_k, since they are source-code statements that are executed when P is run with tc_k.

Definition 3. Let $G_P^k(N, E)$ be a dynamic dependence graph for the program P running tc_k. The backward dynamic slice of S_l, $\text{BDS}_P^k(S_l)$, contains the nodes (i.e. statements) in $G_P^k(N; E)$ which are captured by backward traversal of the graph from S_l to the last reachable node. The remaining part of $G_P^k(N, E)$ contains the nodes which are executed but are not included in $\text{BDS}_P^k(S_l)$.

In other words, $\text{BDS}_P^k(S_l)$, includes the node S_l, all the execution instances having control or data dependence with S_l, as well as the union of the backward dynamic slices of the control and data dependent elements. Since, the failure in a failing run of a given program is mostly manifested as a wrong output value, the

backward dynamic slice of the value frequently captures the faulty code responsible for producing the incorrect value [15]. Therefore, for each failing program run, the execution instances corresponding to the erroneous output statements are identified and the backward slices of such instances are computed.

3.2 Method

The aim of Stat-Slice is to analyze the control and data dependence between the program output and the other parts of a given program in failing and passing runs to pinpoint the faulty statement(s). An appropriate technique for extracting the mentioned data and control dependence is computing the backward dynamic slices, BDS, starting from the output statement.

To find and rank the statement(s) according to their likelihood of being faulty, it is required to contrast the statements in the failing versus passing runs. Since, we want to consider the presence or absence of a given statement in the corresponding BDS when contrasting the passing and failing runs, we convert each execution to a vector where each element of the vector corresponds to a program statement. We assign values to the elements of the vector according to the presence or absence of the element in the BDS of the corresponding execution. Computing the BDS and constructing the vectors is described in Section 3.2.1. Converting a given program run into a vector has some advantages. Presenting the executions in an Euclidean space enables us to find the similar failing and passing executions according to their corresponding BDS and the executed statements. Therefore, we can cluster the vectors according to their similarities in the Euclidean space and identify the different execution paths in the program, as described in Section 3.2.2. Clustering the execution vectors helps us to compute the expanded backward dynamic slice (EBDS), prune the irrelevant data including the coincidental correct test cases, find multiple bugs, provide a visual presentation of the program runs and reduce the amount of manual code inspection. By computing EBDS, described in Section 3.2.3, we can find those faults which are not located in BDS. To compute EBDS, we calculate two conditional probabilities to find out whether the given statements, which are executed but are not included in BDS, have a potential to affect the program result. By clustering the execution vectors, we also prune the irrelevant data which may mislead Stat-Slice in accurate ranking of the statements, as described in Section 3.2.4. Then, as presented in Section 3.2.5, for each statement, the Pearson correlation between two binary variables is computed and the statements are scored and ranked according to their likelihood of being faulty. The first binary variable specifies the observation of the statements in BDS of different runs. The second variable specifies the termination state of the program, failing/passing. Finally, we assign the high scored statements to each cluster and report them to the programmer, as described in Section 3.2.6.

3.2.1 Computing BDS and Constructing the Execution Vectors

The first phase of Stat-Slice includes analyzing a given program and identifying the output statement(s) to find out which points generate the undesired results. Some programs may have multiple outputs and the debugger should identify the one producing the incorrect values. The main concern is how to consider the dynamic data and control dependence between the program output point and the faulty code when analyzing the different failing and passing runs. Since, we do not know the location of the faulty code, we compute the backward full dynamic slice, BDS, starting from the output statement and later we expand the BDS to include more suspicious statements.

Assume that program P has an output statement, OP, with k execution instances, $OP = \{op_1, op_2, \ldots, op_k\}$, and it generates incorrect values for some specific test cases. Here, for simplicity we choose a single execution instance of OP that has an erroneous value in at least one single execution of P. For such op_j, the backward dynamic slice is computed for the failing test case tc_m^{fail}, $\text{BDS}_P^m(op_j)$, and the passing test case tc_n^{pass}, $\text{BDS}_P^n(op_j)$. For programs with more than one incorrect output, simply the union of backward slices for all incorrect outputs is computed.

At this point, a preprocessing is done on our data set to remove the redundant and irrelevant data. As mentioned in Section 2, if there are passing test cases which have identical BDS with the existing failing test cases, we exclude the passing BDS from our further analysis.

Assume $\text{BDS}_P^{All}(op_j)$ contains the union of all slices computed with the existing passing and failing test cases. Therefore, $\text{BDS}_P^{All}(op_j)$ contains the statements which have been observed in at least one backward dynamic slice. With these assumptions, we define the Euclidean execution space as follows.

Definition 4. Given $\text{BDS}_P^{All}(op_j)$, a set of all distinct statements extracted from the failing and passing backward dynamic slices of the program P, we define Prog-Space(P) as the Euclidean space of n-tuples $\{x_1, x_2, \ldots, x_n, \}$ where each tuple represents a dimension (also called feature) in the space. Each x_j stands for one and only one specific element in $\text{BDS}_P^{All}(op_j)$ and can only have three values $\{-1, 0, 1\}$. In other words, each statement in $\text{BDS}_P^{All}(op_j)$ constructs a dimension in Prog-Space(P).

For simplicity, in the remaining part of the paper we assume there is a single output statement and the existence of multiple output statements does not change the algorithm.

Definition 5. Each program run executed by a given test case tc_k, regardless of its failing or passing state, can be represented as the execution vector $V_P^k = \{X_1, X_2, \ldots, X_n\}$ in Prog-Space(P) space, where k is an index for the corresponding test case tc_k and X_i is the value of the dimension x_i that might be:

$$X_i = \begin{cases} 1 & \text{if } \left(x_i \in \mathrm{BDS}_P^k(op_j)\right) \\ 0 & \text{if } \left(x_i \in \left\{G_P^k(N,E) - \mathrm{BDS}_P^k(op_j)\right\}\right) \\ -1 & \text{if } \left(x_i \in \left\{All.St_P - G_P^k(N,E)\right\}\right), \end{cases}$$

where $All.St_P$ represents all the statements in the program P. $X_i = 1$ means that the statement x_i is included in the backward dynamic slice of the corresponding run. In contrast, $X_i = 0$ specifies that the statement is executed but not included in the BDS and $X_i = -1$ means that the statement is not executed in that particular run. Table 1 shows the execution vectors for the nine test cases of the example shown in Figure 1. It is clear that all the statements which marked by '$*$' in Figure 1 have the value of 1 in the Table. Note that the statements in lines 1 to 10 of the program code are executed by all the nine test cases, hence have the value of 1, and for simplicity we do not show them in the Table.

#	TC#1	TC#2	TC#3	TC#4	TC#5	TC#6	TC#7	TC#8	TC#9
11	1	1	1	1	1	1	1	1	0
12	1	1	1	1	1	1	1	1	−1
13	1	1	1	1	1	1	1	1	−1
14	1	1	1	1	1	1	1	1	−1
15	1	1	1	1	1	1	1	1	−1
16	1	1	1	1	1	1	1	1	−1
17	0	0	1	1	1	1	1	1	−1
18	−1	−1	0	1	1	0	1	0	−1
19	−1	−1	−1	1	1	−1	0	−1	−1
20	−1	−1	−1	1	1	−1	−1	−1	−1
21	−1	−1	−1	−1	−1	−1	1	−1	−1
22	−1	−1	−1	−1	−1	−1	1	−1	−1
23	−1	−1	1	−1	−1	1	−1	1	−1
24	−1	−1	1	−1	−1	1	−1	1	−1
25	1	1	1	1	1	1	0	0	−1
26	0	1	0	0	1	1	−1	−1	−1
27	−1	1	−1	−1	1	1	−1	−1	−1
28	1	−1	1	1	−1	−1	−1	−1	−1
29	1	−1	1	1	−1	−1	−1	−1	−1
30	−1	−1	−1	−1	−1	−1	−1	−1	1
31	−1	−1	−1	−1	−1	−1	−1	−1	1
T.S.	**pass**	**pass**	**pass**	**fail**	**fail**	**fail**	**fail**	**pass**	**pass**
	Cluster #1						**Cluster #2**		

Table 1. The execution vectors for the example in Figure 1

Given program P, we call two program runs executed by the test cases tc_m and tc_n similar if their corresponding execution vectors, V_P^m and V_P^n, are close together according to a predefined threshold in the Euclidean space, Prog-Space(P).

3.2.2 Clustering the Execution Vectors

Assuming that the existing passing and failing executions are presented as two failing and passing labeled execution vectors in Prog-Space(P), we have applied k-means clustering method to identify the different execution paths for the effective ranking of the program statements. There are different reasons for clustering the execution vectors:

1. Identifying the different execution paths to find multiple faults in the program. This capability is evaluated on multiple faulty versions of *gzip* and *grep* programs in Section 4.4.1.
2. Computing the expanded backward dynamic slice (EBDS) according to the vectors in each cluster.
3. Pruning the irrelevant data by integrating all failing vectors in each cluster into a single failing vector.
4. Reducing the amount of manual code inspection by assigning high scored statements to each cluster.

In the following definition, we describe the way that the execution vectors are clustered using the k-means technique [25].

Definition 6. Given a set of m execution vectors for program P, without regarding the failing or passing label of the vectors, k-means aims to partition the m vectors into k sets ($k \leq m$) $CL = \{cl_1, cl_2, \ldots, cl_k\}$ according to the following within-cluster function which minimizes the sum of squared Euclidean distances:

$$\arg\min_{CL} \sum_{i=1}^{k} \sum_{V_P^k \in cl_i} \|V_P^k - c_i\|^2, \tag{1}$$

where c_i is the center of the execution vectors in cluster cl_i and V_P^k is an execution vector corresponding to tc_k in Prog-Space(P). The execution points in each cluster share the common executed and slice included statements and therefore they are approximately related to a particular program execution path. The clusters which contain at least one failing execution point are considered for further analysis and we ignore the remaining clusters. After clustering the execution vectors, we prioritize the clusters according to the number of including failing vectors. The more failing vectors in a given cluster the more priority the cluster may have. The priority assignment to the clusters is done in the final stage of Stat-Slice, described in Section 3.2.6, when we allocate the high scored faulty statements to the clusters to reduce the amount of manual code inspection.

As could be seen in Table 1, two clusters can be identified in the execution space of our example, presented in Section 2. Each cluster is surrounded by a grey box in the Table, where the first cluster contains the execution vectors corresponding to

test cases $TC\#1$ to $TC\#6$ and the second cluster contains the vectors of test cases $TC\#7$ to $TC\#10$. Since Cluster#1 contains three failing vectors while Cluster#2 has a single failing vector, the first cluster achieves higher priority in comparison with the second cluster.

3.2.3 Computing the Expanded Backward Dynamic Slice (EBDS)

As mentioned in Definition 5, the value of the statements (i.e. features) which are executed in a specific run but are not included in the corresponding backward dynamic slice is zero. Although, such statements do not appear in the backward slice of an incorrect output statement, in some cases they could be responsible for the program failure. Those statements might be included in the relevant slice of an incorrect output [2] and therefore if they found to be suspicious, it is required to include them in the ranking model. A good example is the faulty statement, line 18, which is executed by test case $TC\#6$ but is not included in the corresponding BDS. This is because the *if* condition in line 18 is not evaluated as *true*, by mistake, and therefore the *else* part, line 23, is included in the BDS, as shown in Table 1. To have precise ranking model, this statement should be included in the backward dynamic slice of the output and a mechanism is required to consider the statement in our analysis. To decide whether a statement should be included in a slice, we consider two conditional probabilities for identifying the suspicious statements.

The aim is to convert the existing zero values to either 1 (have potential to be faulty statement) or -1 (no potential to be faulty statement) by computing two conditional probability functions for the execution vectors in each cluster. With this conversion, we do not miss statements which have the potential to be faulty. It is worth noting that the conversion occurs after the clustering and the real value of features in the execution space does not actually change. In other words, if the zero value of a given vector $(1, 0, 1)$ is converted to -1, the position of the vector in the Euclidean space is not altered at all. This conversion is performed as a preprocessing step for the ranking stage, described in Section 3.2.5.

For each cluster in the execution space, first we identify those statements that have at least one zero value for some execution vectors. Let λ be such statement. We compute the probability of λ being 1 for the failing test cases, $P(\lambda = 1|fail)$, and the passing test cases, $P(\lambda = 1|pass)$, separately. If the computed probability values are less than 0.5, we assume that the corresponding statement has no significant effect on the output statement(s) and its zero value is converted to -1. Otherwise, if the probability is more than 0.5, it may affect the output statement(s) and hence the zero value is converted to $+1$. If the computed probability is exactly 0.5, for failing vectors the zero value is converted to 1 and for passing vectors it will be converted to -1. With this technique, we expand BDS to the suspicious statements of the relevant slice without computing the relevant slices, directly, to reduce the computational cost and decrease the number of irrelevant statements.

When the value of a statement is zero in all passing or failing execution vectors in a given cluster, we act conservatively: if the value of the statement is zero in all

passing vectors, the zero value is converted to -1; if the value is zero in all failing vectors, we convert the value to 1.

To better understand the computation of EBDS, consider the execution vectors of the running example in Table 1. The converted vectors are shown in Table 2 where due to simplicity, only statements having zero values in some test cases are shown. The previous zero value is shown by grey color with an arrow leading to the converted value. The statements with the new value 1 are added to the BDS of the corresponding test case and make EBDS. As could be seen in Table 2, statement 17 in cluster#1 has zero value for test cases $TC\#1$ and $TC\#2$. Since, in one out of the three passing execution vectors in the same cluster, the value of the statement in a single test case, $TC\#3$, is 1, the passing conditional probability would be $P(\lambda = 1|pass) = 1/3$ and thereby the zero values are converted to -1. A similar passing conditional probability is computed for statement 26 and the zero values of test cases $TC\#1$ and $TC\#3$ are converted to -1. Another example is the zero value of statement 18 for failing test case $TC\#6$ in cluster#1. Because the value of the statement in two other failing test cases is 1, the failing conditional probability would be $P(\lambda = 1|fail) = 2/3$ and consequently the zero value becomes 1. Similar to statement 18, the zero value of statement 26 in $TC\#4$ is converted to 1 because the statement in two other failing test cases $TC\#5$ and $TC\#6$ has the value of 1.

#	TC#1	TC#2	TC#3	TC#4	TC#5	TC#6	TC#7	TC#8	TC#9
11	1	1	1	1	1	1	1	1	$0 \rightarrow -1$
17	$0 \rightarrow -1$	$0 \rightarrow -1$	1	1	1	1	1	1	-1
18	-1	-1	$0 \rightarrow -1$	1	1	$0 \rightarrow 1$	1	$0 \rightarrow -1$	-1
19	-1	-1	-1	1	1	-1	$0 \rightarrow 1$	-1	-1
25	1	1	1	1	1	1	$0 \rightarrow 1$	$0 \rightarrow -1$	-1
26	$0 \rightarrow -1$	1	$0 \rightarrow -1$	$0 \rightarrow 1$	1	1	-1	-1	-1
T.S.	**pass**	**pass**	**pass**	**fail**	**fail**	**fail**	**fail**	**pass**	**pass**
	Cluster #1						**Cluster #2**		

Table 2. The converted vectors resulted from converting the zero valued statements in the execution vectors of Table 1. The previous zero value is shown by grey color. The statements with the new value 1 are added to the BDS of the corresponding test case

In cluster#2, there is a single failing vector. Therefore, the zero value of statements 19 and 25 is converted to 1. For statements 18 and 25 of $TC\#8$, the computed passing probability is exactly 0.5 and hence the converted value is -1. A similar conversion is done for statement 11 of $TC\#9$.

In Section 4.5.1, we compare the fault localization performance of Stat-Slice using both EBDS and the relevant slices on a subject program. In the same section, we also discuss the privileges of using EBDS for fault localization in comparison with using the relevant slices.

3.2.4 Pruning the Irrelevant Data in Each Cluster

In order to increase the precision and accuracy of the technique, it is better to prune useless test cases as much as possible and prevent Stat-Slice to be misled by the irrelevant and noise data. To this end, we first integrate all the failing test cases of each cluster into a single failing execution vector. This process simplifies the ranking stage, described in the subsequent section. Note that the pruning is performed on the vectors in EBDS, computed in the previous stage, for which the corresponding elements are either $+1$ or -1.

To integrate the failing vectors of a cluster into a single failing vector, we act as follows. Given cluster_P^γ as the γ^{th} cluster in Prog-Space(P), if there are only two failing execution vectors in $\text{cluster}_P^\gamma : V_P^m = (X_1, \ldots, X_{size})$ and $V_P^n = (Y_1, \ldots, Y_{size})$ where X_i and Y_i are the values of feature x_i and y_i in vector V_P^m and V_P^n, respectively; and *size* is the size of the vectors; If $X_i \neq Y_i$, meaning that one of them is -1 and another is $+1$, the one with value $+1$ also becomes -1. In other words, the elements with value $+1$ are masked by their corresponding -1 elements in another vector. We do this because if a particular statement is not executed in one failing run while it is executed in a similar failing run, it is very unlikely that this statement be a cause of failure. If there are more than two failing vectors in the cluster, we use majority voting technique. For a given statement s, if its value in the majority (or half) of the failing vectors is 1, the value of s in the integrated vector would be 1; otherwise, it would be -1. By applying this process on all failing vectors in each cluster, we achieve a representative failing vector which has the overall characteristics of all existing failing vectors in that cluster.

Consider the running example in this paper. In the second cluster there is a single failing vector and hence it is the representative failing vector in the cluster and there is no need for the integration. But for cluster#1 we have three failing vectors as shown in Table 1. The integrated failing vector, considering the converted values in Table 2, is shown in left hand side of Table 3.

Having a certain number of clusters, each has one specific failing vector; the aim is to select those passing vectors in each cluster which could help us to rank the statements, effectively. In order to consider the diversity in our ranking model, for each failing vector, we find Λ passing execution vectors which are the nearest and Λ passing vectors which are the furthest to the failing vector of each cluster. The parameter Λ is a pre-specified number specified by the test pool characteristics, more described at the end of this section.

The reason for selecting the close passing test cases to a failing execution is straightforward. It is based on the idea of the nearest neighbor [5]. With two similar passing and failing test cases, the differences in the executed statements are very likely to be the faulty code. However, in some cases, it is common that a failing and a passing test case are completely different in terms of their computed backward dynamic slices but both of them provide useful data for statistical debugging. A good example is the passing vector corresponding to $TC\#9$ in which the faulty statement(s) is not included in the corresponding BDS. Considering the slices of

the failing vector in cluster#1 and the mentioned passing vectors, the two execution vectors are far from each other in the execution space. However, both of them should be considered in the ranking phase, to achieve precise results. This is why we also consider far distance vectors in each cluster, for better performance of the fault localization algorithm.

To identify the appropriate vectors, we define a set for candidate vectors, S-Set, which is initially empty. Now for each selected vector in the cluster, its corresponding altered vector is inserted into the set. For cluster_P^γ in the execution space, firstly the failing vector is inserted into the set, and then the altered vectors corresponding to the Λ nearest and the Λ furthest passing vectors in terms of their Euclidean distance with the failing point are inserted into the set. After the process, the S-Set contains δ number of failing vectors and $2 * \Lambda * \delta$ number of passing points where δ is the number of clusters including a failing vector in the space. Note that in our running example there are few passing vectors in each cluster. Therefore, we consider all of them in the ranking model.

3.2.5 Ranking the Statements According to Their Fault Relevance

Stat-Slice can be categorized as a spectrum-based fault localization technique. A program spectrum is an execution profile, reflecting the dynamic behavior of the program, that specifies which elements are executed within a specific execution.

The spectrum-based fault localization techniques capture the testing coverage data (i.e. program spectra). They analyze them using an appropriate similarity coefficient, as a fault suspiciousness metric, to identify those elements of the program which are more correlated to the failure. Similar to [30], we concentrate on block hit spectra which contain a binary flag for each block of code (i.e. an individual program statement) indicating whether a statement is executed in a specific run. The hit spectra of M test cases and N statements can be summarized in a $M * (N + 1)$ matrix where each row represents a particular test case and the first N columns correspond to N different blocks (i.e. statements). The last column indicates the passing or failing state of each test case. Note that in our algorithm, M and N indicate the sizes of S-Set and $\text{BDS}_P^{All}(op_j)$ (defined in Definition 4), respectively. Through the analysis of the spectra matrix, the spectrum based fault localization techniques attempt to find the program blocks which are more likely to be the cause of failure.

In recent years, there has been a variety of fault metrics applied in the spectrum based fault localization techniques. Some of the well-known metrics are Tarantula [8], *Jaccard* and *Ochiai* [30]. For a given statement s the fault metric of Tarantula ($\phi^s_{Tarantula}$), *Jaccard* ($\phi^s_{Jaccard}$) and *Ochiai* (ϕ^s_{Ochiai}) are shown in relations (2), (3) and (4), respectively.

$$\phi^s_{Tarantula} = \frac{\frac{n_{E.F.}}{n_{NE.F.}+n_{E.F.}}}{\frac{n_{E.F.}}{n_{NE.F.}+n_{E.F.}} + \frac{n_{E.P.}}{n_{E.P.}+n_{NE.P.}}} \tag{2}$$

$$\phi^s_{Jaccard} = \frac{n_{E.F.}}{n_{E.F.} + n_{NE.F.} + n_{E.P.}} \tag{3}$$

$$\phi^s_{Ochiai} = \frac{n_{E.F.}}{\sqrt{(n_{E.F.} + n_{NE.F.}) * (n_{E.F.} + n_{E.P.})}} \tag{4}$$

In these relations, $n_{E.F.}$ and $n_{E.P.}$ are the number of failing and passing runs in which the statement s is executed, respectively. Similarly, $n_{NE.F.}$ and $n_{NE.P.}$ are the number of failing and passing runs in which the statement s is not executed, respectively.

As could be seen, the three mentioned metrics compute the suspiciousness of a given statement s based on a simple assumption: the more failing runs in which s is executed the more likelihood of s being faulty. The mentioned metrics have achieved desirable results in many cases. However, apart from Tarantula, the rest do not consider the passing ratio of the executed statement $\left(\text{i.e. } \frac{n_{E.P.}}{n_{E.P.}+n_{NE.P.}}\right)$ in their ranking model. Although Tarantula considers the passing ratio of the executed statement in its denumerator, it cannot assign an appropriate suspiciousness score to the faulty statements in some situations. Therefore, as shown in [30], it has got a lower performance comparing with Ochiai and Jaccard metrics. Note that for an identical number of failing and passing test cases, the Tarantula formula becomes $\frac{n_{E.F.}}{n_{E.F.}+n_{E.P.}}$ which is similar with Ochiai and Jaccard in the numerator and the difference between the three relations would be in their denumerators.

To improve the accuracy of the fault localization process, we have applied the Pearson correlation as our fault suspiciousness metric, shown in relation (5):

$$\phi^s_{X,Y} = \frac{(n_{O.F.} * n_{NO.P.}) - (n_{NO.F.} * n_{O.P.})}{\sqrt{(n_{O.F.} + n_{O.P.}) * (n_{O.F.} + n_{NO.F.}) * (n_{O.P.} + n_{NO.P.}) * (n_{NO.F.} + n_{NO.P.})}} \tag{5}$$

where $n_{O.F.}$ and $n_{O.P.}$ specify the number of times that s is observed in the slice of the failing and passing executions, respectively. Similarly $n_{NO.F.}$ and $n_{NO.P.}$ depict the number of times that s is not observed in the slice of the failing and passing executions, respectively.

The denominator in (5) is used for normalization which causes the scores to be in the range of -1 to 1. If the denominator becomes zero, by default the numerator value is considered as the correlation value.

The Pearson correlation [26] is an appropriate metric to compute the discriminative power of a statement between the failing and passing runs.

Unlike the mentioned metrics, Pearson considers two factors when assigning the fault suspiciousness score to different statements:

1. The number of times in which the statement is observed in the EBDS of the failing runs and the number of times in which the statement is not observed in the EBDS of the passing runs.

2. The number of times in which the statement is observed in the EBDS of the passing runs and the number of times that the statement is not observed in the EBDS of the failing runs.

A given statement is identified as fault suspicious if the amount of the first factor is high whilst the second factor has got a small value. In other words, the larger difference between the two factors, the more likelihood of the statement being faulty. This is because the statement with the highest score is the one for which its presence in the EBDS of a run increases the chance of the termination state to be failing. In contrast, when the statement is not presented in a slice, the corresponding program termination state is more likely to be passing. Considering both factors at the same time is the key power of Stat-Slice which results in high accuracy of the technique.

All eligible statements in the S-Set are assigned scores according to the (5) and ranked in a descendent order. To show how the scores are computed for different statements, consider the example in Figure 1. The left hand side of the Table shows the EBDS for nine test cases and the right hand side shows the value of four items, $n_{O.F.}$, $n_{O.P.}$, $n_{NO.F.}$, $n_{NO.P.}$, required to compute the Pearson correlation, for each statement. The last column demonstrates the computed value as the score of the corresponding statement. As shown in the Table, the faulty statements 18 and 19 have got the highest score (i.e. 1).

3.2.6 Assigning the High Ranked Statements to the Clusters

As mentioned earlier, we have clustered the execution vectors for several reasons. One significant reason is reducing the amount of manual code inspection. By clustering the execution vectors, we are able to distinguish different execution paths in a given program.

As mentioned in Section 3.2.2, after performing the clustering technique on the program execution vectors, we prioritize clusters according to their relevance to the program failure. To this end, we assign priority to a cluster including a larger number of failing execution vectors comparing with other clusters (we are talking about the initial clusters before integrating the failing vectors in a single vector). Now we are ready to assign the scores to the clusters in the execution space. At this point, we want to know which fault suspicious statement, identified in the previous stage, should be examined first to find out whether it is the actual cause of the failure. The process is done hierarchically such that we first investigate the cluster with the highest priority and examine whether the statement with the highest score in that cluster is the actual cause of failure. If it is the cause of failure, we report it to the user. Otherwise, we go to the second rank cluster to examine the highest score statement, not seen previously, in that cluster and so on. The process continues until we find the cause of failure or all the clusters are searched with their highest scored statements. In the latter situation, in a hierarchical process the second high scored statement of the clusters are examined according to the priority of the cluster and so on. With this level based strategy, we try not to lose any suspicious statement

#	**TC#1**	**TC#2**	**TC#3**	**TC#4, 5, 6**	**TC#7**	**TC#8**	**TC#9**
11	1	1	1	1	1	1	−1
12	1	1	1	1	1	1	−1
13	1	1	1	1	1	1	−1
14	1	1	1	1	1	1	−1
15	1	1	1	1	1	1	−1
16	1	1	1	1	1	1	−1
17	−1	−1	1	1	1	1	−1
18	−1	−1	−1	1	1	−1	−1
19	−1	−1	−1	1	1	−1	−1
20	−1	−1	−1	1	−1	−1	−1
21	−1	−1	−1	−1	1	−1	−1
22	−1	−1	−1	−1	1	−1	−1
23	−1	−1	1	−1	−1	1	−1
24	−1	−1	1	−1	−1	1	−1
25	1	1	1	1	1	−1	−1
26	−1	1	−1	1	1	−1	−1
27	−1	1	−1	1	−1	−1	−1
28	1	−1	1	−1	−1	−1	−1
29	1	−1	1	−1	−1	−1	−1
30	−1	−1	−1	−1	1	−1	1
31	−1	−1	−1	−1	−1	−1	1
T.S.	**pass**	**pass**	**pass**	**fail**	**fail**	**pass**	**pass**

$n_{O.F.}$	$n_{NO.P.}$	$n_{NO.F.}$	$n_{O.P.}$	*Score*
2	1	0	4	0.26
2	1	0	4	0.26
2	1	0	4	0.26
2	1	0	4	0.26
2	1	0	4	0.26
2	1	0	4	0.26
2	3	0	2	0.55
2	5	0	0	1.00
2	5	0	0	1.00
1	5	1	0	0.65
1	5	1	0	0.65
1	5	1	0	0.65
0	3	2	2	-0.40
0	3	2	2	-0.40
2	2	0	3	0.40
1	4	1	1	0.30
1	4	1	1	0.30
0	3	2	2	-0.40
0	3	2	2	-0.40
0	4	2	1	-0.26
0	4	2	1	-0.26

Table 3. Assigning scores to each statement according to given test cases of the example in Figure 1. The left hand side shows EBDS for nine test cases. The right hand side shows the values of Pearson factors and the last column shows the computed Pearson correlation as the statement score.

related to a particular execution path (i.e. cluster) and simultaneously reduce the manual code scrutinization.

4 EXPERIMENTAL RESULTS

In this section, the effectiveness of Stat-Slice is empirically evaluated. To this end, we compare Stat-Slice with some outstanding statistical debugging techniques in the context of software fault localization. We have also compared our performance with 'Pruning with confidence' slicing technique [15] which has compared its capability with other slicing techniques and thereby we do not include other slicing methods, here. The evaluation is performed on Siemens, gzip, grep, and flex [23] programs. The test pool for each program contains a number of faulty versions and each version includes at least one type of fault. A brief description about each benchmark is presented in Table 4. For each program, a number of failing and passing test cases are available. However, for our experiments we consider a few number of test cases to simulate conditions in which we are not provided with a large number of executions.

Application	Versions	Procedures	Lines	Test Cases	Description
print-tokens	7	20	472	4 056	*lexical analyzer*
print-tokens2	10	21	399	4 071	*lexical aralyzer*
replace	32	21	512	5 542	*pattern replacement*
schedule	9	18	292	2 650	*priority scheduler*
schedule2	10	16	301	2 680	*priority scheduler*
tcas	41	8	141	1 578	*altitude separatton*
tot-info	23	16	440	1 054	*information*
gzip	6	146	6 582	217	*measure*
grep	6	104	15 633	809	*file compressor*
flex	5	162	12 418	525	*lexical analyzer*

Table 4. A brief description of the subject programs

To compute the backward dynamic slices, we used the dynamic slicing framework introduced in [22, 24]. The framework instruments a given program and executes a gcc compiler to generate binaries and collects the program dynamic data to produce its dynamic dependence graph. The framework contains two main tools: Valgrind [20] and Diablo [19]. The instrumentation is done by Valgrind memory debugger and profiler which also identifies the data dependence among the statement execution instances. The Diablo tool is capable to produce program's control flow graph from the generated binaries. Finally we used the WET tool to compute the backward dynamic slices from an incorrect output value. To perform clustering, we used WEKA machine learning and data mining tool [22] which is an open source software. The experiments are conducted on 2.4 GHz Intel Core 2 Quad CPU with 3.50 GB RAM running linux UBUNTU 6.06 with gcc 2.3. The test pools are obtained from Software Repository Infrastructure (SIR) [18]. Before presenting the results, some important issues should be mentioned:

1. An important factor to measure the effectiveness of a ranking algorithm is the number of faults that could be located according to the amount of code inspection by the programmer. The manual code inspection is measured using the well-known PDG-based T-score evaluation framework which is used in previous works [5, 6, 8]. We have also presented our results according to a conventional slicing evaluation framework.
2. The dynamic slicing framework does not work properly on tot_info program and it only produces a limited data dependence graph and no control dependence data. Therefore, the results for tot_info have been partially collected considering this limitation.
3. Due to the segmentation faults we could not run some versions of the Siemens programs and hence we could not detect their faults.
4. For statements with the same fault relevance score, the priority is with the statement which is closer to the output statement.

4.1 Experiments on the Siemens Programs

In this section, we first present our results on six programs of Siemens according to the T-score evaluation framework in Figure 2. In the subsequent parts of the section, we compare the performance of Stat-Slice with some fault localization techniques. With T-score less than one percent, Stat-Slice finds 48 faults which outperforms other fault localization techniques.

The dynamic slicing techniques for fault localization use some measurements other than the T-score, to evaluate their performance. These measurements are: the size of the program (i.e. LOC), the executed statements in average (i.e. Avg(Exec)), the maximum amount of code explored to reach the faulty code (Max(EFS)), the minimum amount of code explored to reach the faulty code (i.e. Min(EFS)), the average amount of code explored to reach the faulty code (i.e. Avg(EFS)). The performance of Stat-Slice according to these measurements for six programs of Siemens suite is presented in Table 5.

Program	LOC	Avg (Exec)	Max (EFS)	Min (EFS)	Avg (EFS)	Avg(EFS)/ LOC	Avg(EFS)/ Avg(Exec)
print-tokens	726	142	11	2	8	0.011	0.056
print-tokens2	570	170	27	1	9	0.015	0.052
schedule	412	137	5	1	4	0.009	0.029
schdeule2	374	122	42	3	13	0.034	0.10
replace	564	110	40	1	8	0.014	0.07
tcas	173	53	28	1	3	0.017	0.053

Table 5. The performance of Stat-Slice according to the slicing measurements on Siemens suite

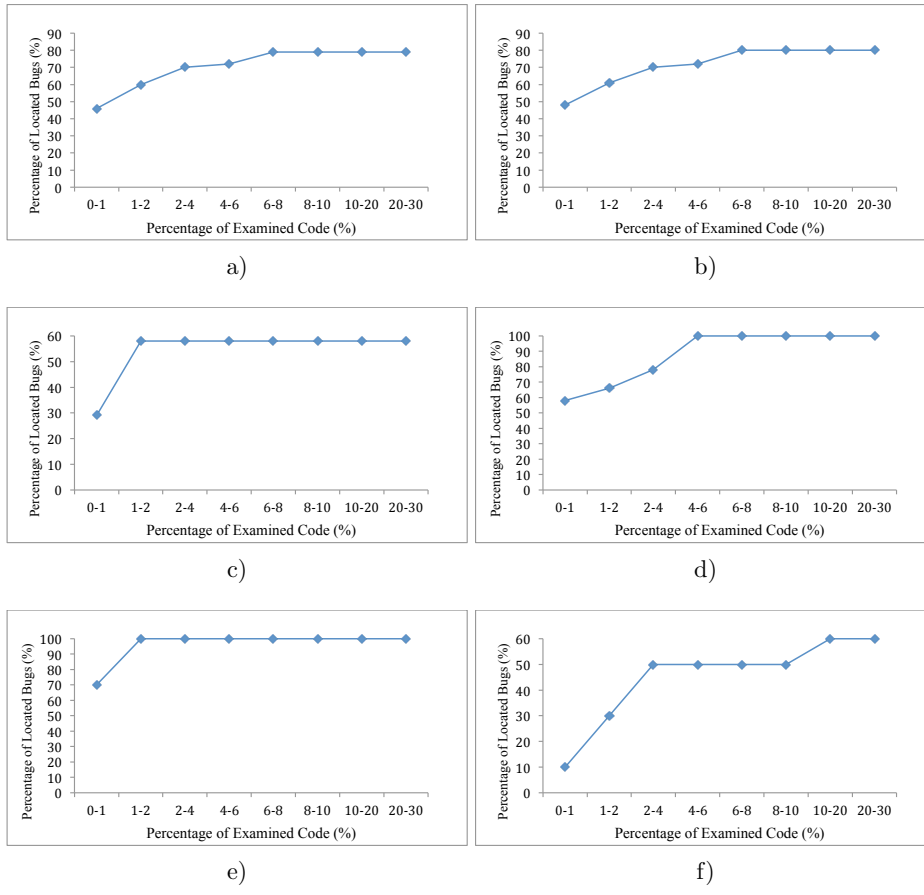

Figure 2. The performance results of Stat-Slice on Siemens programs: a) tcas, b) replace c) printtokens d) printtokens2 e) schedule f) schedule2

Figure 3 shows two examples for the omitted code fault and the macro statement fault which are located in Figures 3 a) and 3 b), respectively, and Stat-Slice has detected both of them precisely. The omitted code fault simulates a semantic fault for which a programmer by mistake omits the required code or does not include it in its right place. In Figure 3 a) the red code, in line 177, represents the omitted code in function get_token(tp) which is commented out. Stat-Slice reports line 176 as the fault suspicious statement with the highest score. Since the omitted code is not included in the dynamic program slice, it cannot be captured by conventional slicing techniques. For the fault in Figure 3 b), Stat-Slice reports the exact location of the fault in the macro statement, line 18.

As mentioned earlier, the full slicing techniques are incapable to locate faults which are included in the relevant slices of the output. As an example, consider the

```
    ....
174. if(is_eof_token(ch1)==TRUE)
175.   {ch=unget_char(ch,tp);
176. if(ch==EOF)unget_error(tp);
177. /*return(buffer);*/  //omitted code fault
178. }
    .....
```

(a)omitted code fault in version 3 of program printtokens2, function *token get_token(tp)*

```
...
14. #include "chisq.h"
15.#include  "gamma.h"
  ...
18. #define MAXLINE 56 // Fault-it should be 256 instead of 56
  ...
21.    #ifndef MAXTBL
....
```

(b)Version 6 of program tot_info,line 18: A fault in macro statement.

Figure 3. An example of the omitted code fault and the macro statement fault

version 4 of the program *Schedule*, where the fault is located in line 207 as shown in Figure 4 a). The corresponding output statement has 13 instances, integer values, in each single run as shown in Figure 4 b). The test case No. 34, in the test pool provided from [18] is a failing test case for which the program output statement does not produce a correct value in the instance 11 and 12 of its output as shown in Figure 4 b). The correct value is also shown in the Figure. While the faulty statement, line 207, is not included in the BDS of the output point for some test cases, the EBDS does it contains and Stat-Slice captures the fault with the highest score.

```
…
205.   count = src_queue->mem_count;

207.   if (count > 1) /* off by one */ {        //it should be "if(count>0)"
208.     n = (int) (count*ratio + 1);
209.     proc = find_nth(src_queue, n);
…
```

(a)

Ins 1	Ins 2	Ins 3	Ins 4	Ins 5	Ins 6	Ins 7	Ins 8	Ins 9	Ins 10	Ins 11	Ins 12	Ins 13	Output instances
2	1	12	10	6	7	8	9	11	13	5	0	3	Correct result
2	1	12	10	6	7	8	9	11	13	0	5	3	Incorrect result

(b)

Figure 4. An example from version 4 of *Schedule* program where the faulty statement is not in BDS of a specific failing test case

4.2 Comparison with the Spectrum Based Fault Localization Techniques

As described in Section 3.2.5, Stat-Slice can be categorized as a spectrum based fault localization technique. In this section, we empirically compare the performance of our technique with Tarantula, Jaccard and Ochiai. The experimental results on the Siemens suite show that, with less than one percent manual code inspection, Ochiai, Jaccard, and Tarantula have detected 28, 17, and 13 percent of faults, respectively. With the same T-score, Stat-Slice has detected 36 percent of faults which outper-

forms other spectrum based fault localisation techniques. For T-score less than 10 %, Stat-Slice has detected 67 percent of faults comparing with Tarantula, Jaccard and Ochiai which have found 52, 55 and 59 percent of faults, respectively. The result of this comparison is shown in Figure 5.

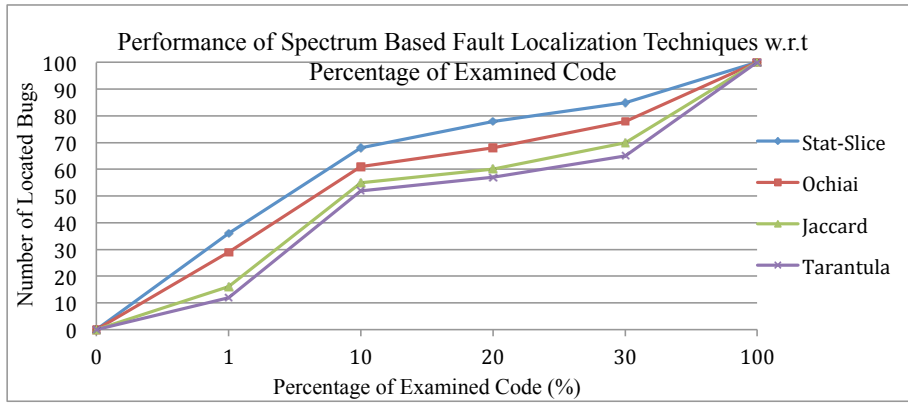

Figure 5. Performance comparison between Stat-Slice and three other spectrum based fault localization techniques on Siemens suite

There are some reasons which justify the outperformance of Stat-Slice comparing with other spectrum based fault localization techniques. While the spectrum based fault localization techniques consider all program statements as the program spectra, Stat-Slice limits the program spectra to the statements which are observed in EBDS of the program output result. Therefore, many irrelevant and misleading statements are excluded for the further analysis, improving the performance of Stat-Slice. The second reason is pruning the irrelevant passing test cases (e.g. coincidental correct test cases) by clustering the execution vectors and finding the nearest and furthest passing vectors to the representative failing vector of each cluster. The third reason is our fault metric, Pearson, which outperforms the previous fault localization metrics, on our subject programs. To study the effect of Pearson on the performance of Stat-Slice, we have conducted an experiment in Section 4.5.2, in which Pearson ranking metric of Stat-Slice is substituted by three mentioned metrics, Ochiai, Jaccard and Tarantula and the fault localisation performance of Stat-Slice is analysed using all four mentioned metrics, so called Stat-Slice(Pearson), Stat-Slice(Ochiai), Stat-Slice(Jaccard), and Stat-Slice(Tarantula).

4.3 Comparison with Other Fault Localization Techniques

A full comparison between the four fault localization techniques, shown in Figure 6, reveals that Stat-Slice outperforms Ochiai, Liblit, and Sober.

As mentioned before, 'Pruning with Confidence Value' (CV) technique [15] ranks the statements in a single failing execution based on their relevance to the failure. It

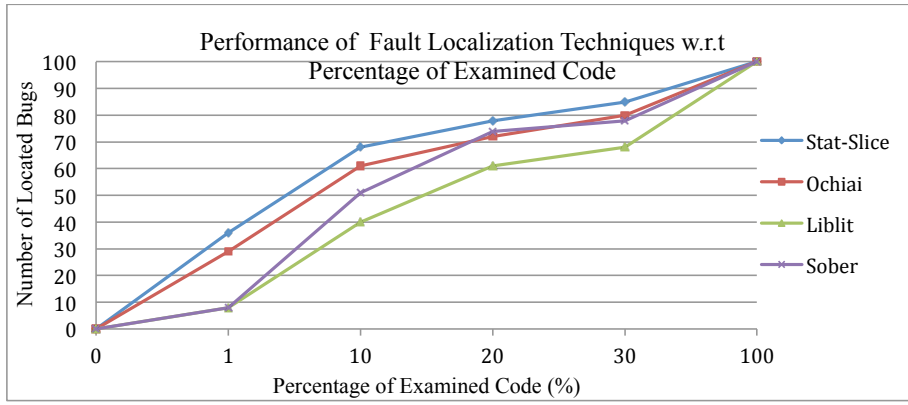

Figure 6. A full comparison between Stat-Slice and Ochiai, Liblit (2005), Sober on Siemens suite

assumes that the given program has multiple output values and prior to an incorrect output value there exist some correct output values in the same run. Therefore, it is applicable in particular programs. Since all programs of Siemens do not have this characteristic, the evaluation of CV technique is performed on some versions (i.e. only 24 out of 132 versions) of five programs. The comparison results for those incomplete five programs are presented in Figure 7.

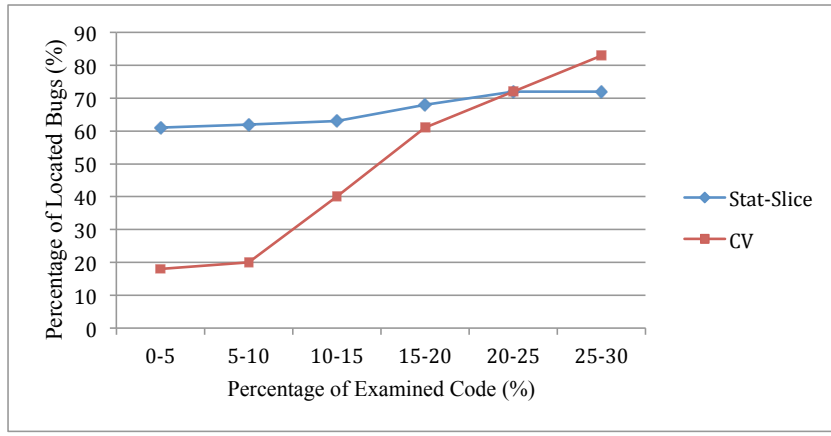

Figure 7. The result of CV and Stat-Slice on some versions of 5 programs in Siemens suite: *printtokens*, *printtokens2*, *schedule*, *schedule2* and *replace*

4.4 Experiments on gzip and grep Programs

In this section we evaluate the scalability of Stat-Slice on reasonably large programs. To this end, we have chosen gzip and grep programs with 6 582 and 15 633 lines of code, respectively [23, 24]. In each test pool, provided from software infrastructure repository [18], there is a folder called 'versions.seeded' containing five faulty versions with different seeded faults. Each faulty version has a header file 'FaultSeeds.h' containing some fault macros. By enabling each macro in the program code, the corresponding fault is expanded in the code. To specify which fault is seeded into the program we use the notation 'prog-name.v_x.y', in which prog-name, x and y specify the program name (i.e. grep or gzip), the faulty version number and the order of the macro statement in 'FaultSeeds.h', respectively.

The faulty versions we select for our experiments are specified in Table 6. Table 6 also presents the performance of Stat-Slice according to the slicing techniques measurements for fault localization. As shown in the Table, with very small amount of manual code inspection, we could locate all the existing faults in two programs. In some versions, the actual location of faulty statement is very close to the statement reported by Stat-Slice.

	Program	LOC	Max (EFS)	Min (EFS)	Avg (EFS)	Avg(EFS)/LOC
grep	preg_v1.11	15 633	10	1	4	0.0003
	gvep_r3.2					
	grep_v4.10					
	grep_v4.12					
gzip	gzip_v1.5	6 582	37	2	13	0.002
	gzip_v4.6					
	gzip_v5.6					
	gzip_v5.13					

Table 6. The performance of Stat-Slice according to the slicing measurements on gzip and grep

4.4.1 Finding Multiple Faults in gzip and grep Programs

As described in Section 3.2.2, one advantage of clustering the execution points is to find multiple faults in programs. To evaluate the performance of Stat-Slice in finding multiple faults, we have conducted an experiment on multiple faulty versions of gzip and grep programs. To build the multiple faulty versions in these programs, we have enabled more than a single fault macro in a version at the same time. For example to make a 2-fault version of grep, we have enabled the first and 11^{th} fault macro of the first version (i.e. grep.v_1.1 + grep.v_1.11). By clustering the execution vectors, the passing and failing test cases are partitioned according to different individual faults in a program. Therefore, for each individual fault the corresponding failure origin may achieve the highest fault relevance score. Without clustering, it is unclear

which failing test case is relevant to a specific fault in the program. Hence, due to the mixture of failing test cases correspond to different faults in the program, the failure origins cannot be detected easily.

For more convincing, we have compared the performance of Stat-Slice and two well-known multiple fault localization techniques in finding multiple faults on 2-fault and 3-fault versions of gzip and grep programs. We name these two techniques Parallel Debugging [33] and MulBug Finder [34]. The Parallel Debugging technique, proposed by Jones et al. applies clustering technique on failing test cases based on behavior models and fault localization information of program spectra. Each cluster relates to a specific fault and the including failing test cases are combined with the passing test cases to build a specialized test suite relevant to the corresponding fault. Then it applies the Tarantula ranking model [8] on the specialized test cases of each cluster to find the corresponding single fault in that cluster. The MulBug Finder algorithm proposed by Zheng et al. uses a bi-clustering technique to partition predicates according to their presence in different failing and passing runs. The correlation among predicates are computed using some conditional probabilities. It also uses a voting system which assigns a vote to a predicate according to its presence in failing runs. After computing the votes for each predicate, it ranks the predicates based on their impact on the failure outcome of the program.

Te results shown in Table 7 reveal that due to the clustering and pruning the irrelevant data, Stat-Slice locates multiple faults with much smaller T-score values in comparison with two other multiple fault localization techniques. Our experiments also show that the use of backward dynamic slicing information in addition to the proposed ranking model are the reasons for outperformance of Stat-Slice in finding multiple faults in programs.

4.5 Evaluating the Different Aspects of Stat-Slice

As described in Section 3, Stat-Slice has different phases. Each phase has a significant effect on the performance of Stat-Slice, as mentioned in Sections 3.2.1, 3.2.2, 3.2.3, 3.2.4, 3.2.5 and 3.2.6. Some important aspects of Stat-Slice are evaluated in the previous sub-sections. In this section we experimentally evaluate some other key aspects of Stat-Slice.

The experiments are done on flex program. We have activated 30 faults in the program and made 30 faulty versions. We have also randomly generated several passing and failing test cases in addition to the existing test cases in the test pool provided from [18]. Flex is a lexical analyser with 12418 lines of code. The organization of this section is as follows.

In Section 4.5.1 we have compared the performance of Stat-Slice using both EBDS and backward relevant slices. In Section 4.5.2 we have studied the effect of Pearson metric on fault localization in comparison with other fault metrics. In Section 4.5.3 the impact of the cluster numbers on the performance of Stat-Slice is evaluated.

	grep				gzip			
2 faults	grep.1.1 grep.1.11	grep.1.11 grep.1.15	grep.1.1 **grep.1.15**	grep.1.11 grep.1.12	gzip.1.5 gzip.1.13	gzip.1.13 gzip.1.14	gzip.5.10 gzip.5.5	gzip.5.6 gzip.5.13
Stat-Slice (T-Score) %	0.22	0.15	0.07	0.22	0.02	0.05	0.24	0.03
MulBug Finder [47] (T-Socre) %	0.31	0.29	0.21	0.41	0.14	0.16	0.25	0.05
Parallel Debugging [46] (T-Score) %	0.37	0.21	0.11	0.33	0.25	0.11	0.24	0.05
3 faults	**grep.v1.1** **grep.v1.12** **grep.v1.15**		**grep.v1.11** **grep.v1.12** **grep.v1.15**		**gzip.v1.5** **gzip.v1.13** **gzip.v1.14**		**gzip.v1.3** **gzip.v1.5** **gzip.v1.13**	
Stlt-Saice (e-ScorT) %	0.41		0.26		0.05		0.15	
MulBug Finder [47] (T-Score) %	0.53		0.41		0.08		0.26	
Parallel Debugging [46] (T-Score) %	0.58		0.32		0.15		0.21	
3 fluats	**grep.v4.5** **grep.v4.7** **grep.v4.10**		**grep.v4.4** **grep.v4.10** **grep.v4.12**		**gzip.v5.6** **gzip.v5.10** **gzip.v5.13**		**gzip.v5.5** **gzip.v5.6** **gzip.v5.13**	
Stat-Slice (T-Score) %	0.30		0.07		0.32		0.04	
MulBug Finder [47] (T-Score) %	0.33		0.10		0.45		0.28	
Parallel Debugging [46] (T-Score) %	0.31		0.12		0.35		0.20	

Table 7. The performance of Stat-Slice, MulBug Finder and Parallel Debugging in finding multiple faults in gzip and grep programs

4.5.1 Computing EBDS Versus Backward Relevant Slice

According to our experiments on flex, gzip and grep programs, the time overhead of computing the relevant slices, in average, is between 5 to 15 percent more than the full dynamic slicing. The time overhead for computing the relevant slice of a single failing test case could be easily neglected. However, since Stat-Slice must compute the backward relevant slices for larger number of passing and failing test cases, the imposed time overhead is not trivial.

For small and middle size programs, the difference between the timing of dynamic and relevant slicing is insignificant and can be easily ignored. But for large scale programs considering numerous failing and passing test cases, the difference could be considerable.

But the more important issue is the size of the dynamic and relevant slices. As mentioned in [4] for the predicate faults compared to full dynamic slices, the relevant slices are 2–181 % larger in size. Again for large number of failing and passing test cases, this difference could be significant and considerable. Note that we are trying to reduce the number of fault candidate statements for our analysis

as much as possible. The large number of statements increases the dimensionality of the execution space affecting size and the contents of the execution vectors, the identified clusters, the nearest and furthest passing to failing vectors, scoring the statements according to the Pearson ranking model and prioritizing the statements. Therefore, by using EBDS we attempt to avoid introducing redundant and irrelevant data into the slices.

For more convenience, to show that the existence of the irrelevant statements in the relevant slices may mislead our ranking model, we have evaluated Stat-Slice based on both expanded dynamic slices (Stat-Slice (EBDS)) and the relevant slices (Stat-Slice (RelSl)) to find out which one achieves more accurate results.

The results in Figure 8 show the amount of code inspection to find all faults in flex program. As could be seen, both Stat-Slice(EBDS) and Stat-Slice(RelSl) achieve promising results on flex program. However, the results of Stat-Slice(EBDS) is much better than Stat-Slice(RelSl). With T-score less than 1 %, Stat-Slice(EBDS) finds 9 faults, while Stat-Slice(RelSlice) locates only 6 of them.

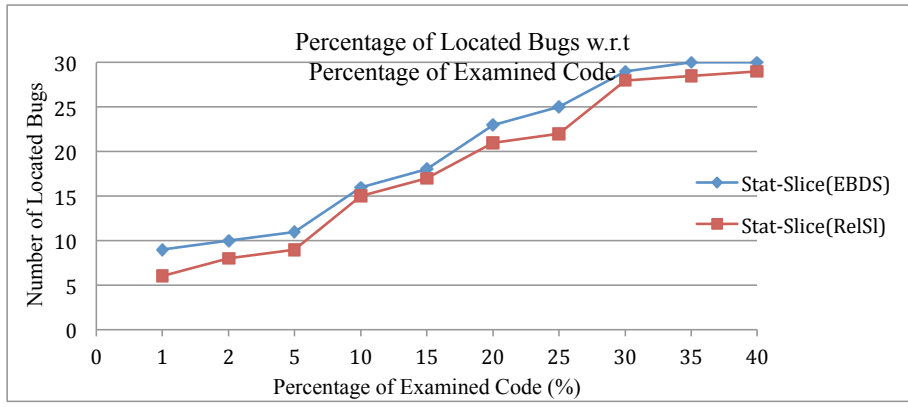

Figure 8. The performance comparison of Stat-Slice(EBDS) and Stat-Slice(RelSl) on flex program

4.5.2 The Effect of Pearson Metric on the Fault Localization Performance

In Section 4.2 we have compared the performance of Stat-Slice and three other spectrum based fault localization techniques, Tarantula, Ochiai, and Jaccard. However, the focus of that experiment is not the effectiveness of Pearson in comparison with other mentioned metrics. In this section, we evaluate the performance of Stat-Slice using different fault metrics. In other words, we substitute the Pearson metric with other mentioned metrics to evaluate the impact of Pearson on the performance of Stat-Slice. The resultant techniques are named Stat-Slice(Tarantula), Stat-

Slice(Jaccard), and Stat-Slice(Ochiai) which are compared with Stat-Slice(Pearson). The result of this comparison on flex program is shown in Figure 9.

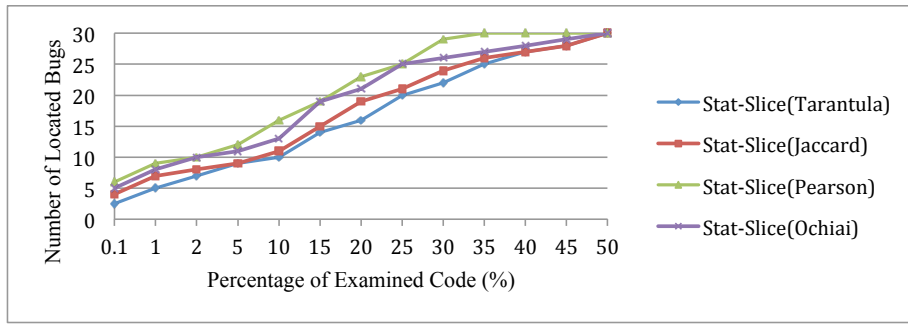

Figure 9. The performance comparison of Stat-Slice(Tarantula), Stat-Slice(Jaccard), Stat-Slice(Ochai) and Stat-Slice(Pearson)

As shown in the Figure, Stat-Slice(Pearson) outperforms other techniques. Some reasons for this outperformance are described in Section 3.2.5.

4.5.3 The Effect of the Cluster Numbers on the Fault Localization Performance

As described in Section 3.2.2, the choice of k (i.e. the cluster numbers) highly depends on the data distribution in the Euclidean distance. In this section, we show the impact of choosing k on the performance of Stat-Slice.

Our experiments on the flex program show that the best result is achieved when k is 15 as shown in Figure 10, $k = 13$ and $k = 17$ are in the next stages, respectively.

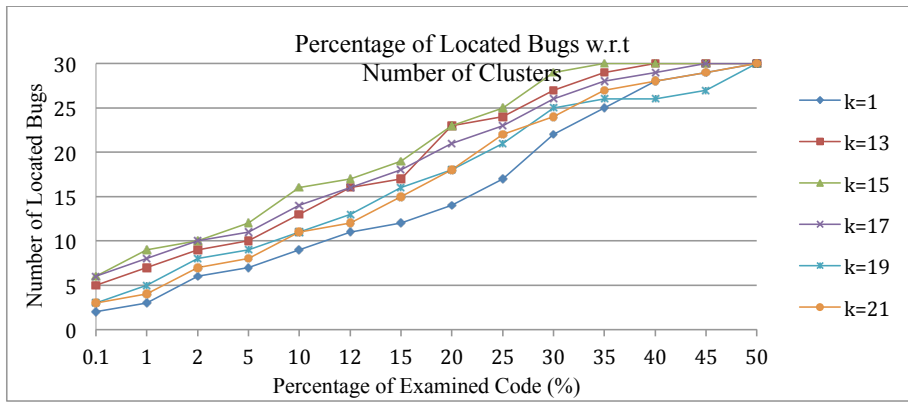

Figure 10. The performance comparison of Stat-Slice according to different k values

Although, the larger number of clusters may decrease the mean squared error in typical data sets, in our application the larger number of clusters may introduce undesired passing and failing test cases into the ranking model decreasing the performance of the fault localization algorithm. The small number of the clusters, on the other hand, may throw away the desired failing and passing test cases, result in poor ranking model of the method. Therefore, the exact number of clusters may highly affect the performance of our fault localization technique.

This experiment also shows the impact of the clustering on the performance of Stat-Slice. As could be seen in Figure 10, for $k = 1$, when there is a single cluster (i.e. equivalent to no clustering), the performance is considerably poorer comparing with other values of k.

5 RELATED WORK

Some related work has been described earlier. However, we consider some other major works in this section. Firstly, we discuss about the dynamic slicing techniques and some variants in Section 5.1 and in the second part of this section, Section 5.2, we consider other related work in this regard.

5.1 Slicing Techniques

Although it has been shown that dynamic slicing substantially reduces the number of program statements to be examined, the absolute number of statements might still be large and many of the statements in the slice are unlikely to be fault relevant [14, 16, 17].

To reduce the size of the dynamic slices, Zhang et al. [14] integrates the dynamic backward slicing with the idea of delta debugging [10]. To reduce the fault candidate set using the delta debugging method, a minimal failure-inducing input is identified. A minimal failure-inducing input is a part of the input that is found to be responsible for failure. Then the forward slice starting from the failure-inducing input is computed and intersected with the backward dynamic slice of an incorrect output.

In their later work [9], a bidirectional dynamic slice is computed for a specific identified decision making statement known as a critical predicate. The critical predicate is a conditional branch statement which is likely to be responsible for the failure execution and if an execution instance of that predicate is switched from one outcome to another (e.g. a *true* outcome is reversed to *false* or vice versa), the program produces a desirable output. The bidirectional dynamic slice contains statements in both forward and backward slices of a critical predicate.

To produce even smaller slices, in [17] the intersection of slices from three different points is computed; these points are: critical predicate, failure-inducing input and incorrect output. In a similar work [16] a set of values which have been used in an execution instance of a statement in a failing execution is replaced with some

other values to analyze whether the program result changes from incorrect to correct output. If that happens, the statement is marked as a faulty statement.

Nonetheless, none of the mentioned techniques are able to rank the statements according to their fault relevance and thereby a human debugger should examine all the reported statements with the same chance to find the failure origin. Moreover, a minimal failure-inducing input cannot be computed for all types of programs. In many situations, all input parameters may have some impact on the program result (i.e. cause failure) and cannot be simplified and isolated any further. On the other hand, the techniques that rely on critical predicate switching [9] or value replacement [16] may have scalability problems for large programs with huge amount of data values. In the predicate switching case, there might be many predicates in a program and a failure output may be the cause of more than a single critical predicate [14]. Thereby, identifying the critical predicates among a large number of predicates does not seem to be trivial and straightforward.

Another limitation of the slicing techniques is their dependence on a single failing execution. Hence, they only identify a fault that is related to that failure and cannot detect other unknown faults of a program. They are also incapable to find multiple bugs. Furthermore, due to the nature of the slicing, some types of bugs (e.g. omitted code) are hardly to capture [24].

5.2 Other Fault Localization Techniques

Zeller has proposed Delta Debugging [10] which analyzes the differences in the program state between the successful and failing executions to isolate bugs. The result is simplified minimal failure-inducing input. An extended approach in [11] checks the program memory states and generates a cause transition chain to find the responsible variables for the program failure [21].

In [7] a logistic regression technique is applied to find the bug predictors. In [13] a combination of predicates is analyzed to check whether a combined set of predicates are relevant to the failure. The mentioned regression methods are applied on the predicates rather than the statements, as performed in Stat-Slice, and they require reasonably large number of failing and passing test cases to build accurate models. Furthermore, Stat-Slice takes advantage of the dependence relationships between the faulty code (i.e., the cause) and the erroneous output (i.e., the effect) by computing the backward dynamic slices which is not considered in the previous statistical fault localization techniques.

In [31] the fault localization is done using neural network models. The program spectrum for each test case (i.e. the executed statements) as well as the fail/pass termination state of the program are fed to the neural network to build a classifier which models the relationship between the statement coverage data and the execution result. After training the network, for each single statement, a virtual test case is designed and is given to the network to assess the output value of the network as a fault suspiciousness indicator of the statement. The statements are then ranked according to the likelihood of containing a bug.

Wong et al. [32] have proposed a heuristic based statistical analysis technique which uses the statement coverage data for each test case and the corresponding termination state (i.e. pass or fail) to find the location of faults. To this end, a crosstab technique is built for each executable statement and the calculated statistic is used to score the statement according to its fault suspiciousness. The crosstab is used to analyze the relationship between two categorical variables, covered/not covered, and two other categorical variables, passed/failed. To find the dependence between the execution of a statement and the program termination state, a null hypothesis test based on chi-square is applied.

Debugging parallel programs is discussed in [28]. It shows that the problems arise from the nature of parallel programs such as communication and synchronization limit the applicability of sequential debuggers. It proposes a three phase record & replay mechanism to achieve the required debugging data. The three phases are tracing the order of occurred events, increasing the amount of tracing to build an event graph display, and obtaining the required data for sequential re-execution of target process.

Zhang et al. in [29] study the impact of using the non-parametric techniques on fault localization in comparison to parametric hypothesis testing methods. Their predicate based fault localization framework compares the differences in program spectra in failing and passing runs. They use the $Wilcox$ on signed-rank test and the Mann-Whitney test as their non-parametric testing methods and the Student's t-test and the F-test as two parametric hypothesis testing methods. Their experiments conducted on Siemens and Space test suite reveal that the non-parametric tests outperform the parametric hypothesis testing methods.

In [35] the topic of testing and debugging of distributed programs is investigated. It motivates the need for testing and debugging activities by depicting the problems involved in developing distributed applications.

6 CONCLUSIONS

In this paper, a new ranking technique for program fault localization is presented. Stat-Slice combines the features of backward dynamic slicing and statistical fault localization techniques to locate a wider range of faults with less amount of manual code inspection. Unlike the statistical fault localization techniques which rely on a large number of passing and failing runs, our proposed technique uses fewer numbers of test cases. Stat-Slice converts each program run to a vector according to three possible states of including statements,

1. included in the backward dynamic slice,
2. executed but not included in the backward dynamic slice, and
3. non executed.

To identify different program execution paths, we cluster the vectors according to their similarity in Euclidean space. By pruning the vectors and using the proposed

ranking model, all eligible statements are given score based on their likelihood to be faulty. In different stages of the technique, we try to exclude the irrelevant statements from the ranking model while increasing the chance of pinpointing the exact origin of failure. The experiments are conducted on Siemens, flex, gzip and grep suites. We have compared our results on Siemens with different fault localization techniques and show that due to the nature of Stat-Slice and using a small number of runs, we could find bugs more accurate than other techniques. We have also shown that, Stat-Slice is scalable in terms of the program size and supports our claim by presenting our results on large size programs. We have evaluated the effects of different methods within our technique on fault localization performance of Stat-Slice. We have also conducted experiments on multiple fault versions and the results demonstrate the significant superiority of Stat-Slice compared with similar techniques.

Acknowledgements

The authors highly appreciate the insightful and constructive questions, comments, and suggestions from anonymous referees, which proved invaluable during the preparation of the paper. The authors would like to thank Iran Telecommunication Research Centre (ITRC) for the financial support.

REFERENCES

[1] Agrawal, H.—DeMillo, R. A.—Spafford, E. H.: Debugging with Dynamic Slicing and Backtracking. Softw. Pract. Exper., Vol. 23, 1993, No. 6, pp. 589–616.

[2] Gyimóthy, T.—Beszédes, Á.—Forgács, I.: An Efficient Relevant Slicing Method for Debugging. Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Toulouse, France, 1999, pp. 303–321.

[3] Gorla, A.—Pezzè, M.—Wuttke, J.—Mariani, L.—Pastore, F.: Achieving Cost-Effective Software Reliability through Self-Healing. Computing and Informatics, Slovak Academy of Science, Vol. 29, 2010, No. 1. pp. 93–115.

[4] Zhang, X.—He, H.—Gupta, N.—Gupta, R.: Experimental Evaluation of Using Dynamic Slices for Fault Location. Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging (AADEBUG '05), Monterey, CA, USA, 2005, pp. 33–42.

[5] Renieris, M.—Reiss, S. P.: Fault Localization with Nearest Neighbor Queries. Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03), Montreal, Canada, 2003, pp. 30–39.

[6] Liu, C.—Yan, X.—Fei, L.—Han, J.—Midkiff, S. P.: SOBER: Statistical Model-Based Bug Localization. Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium

on Foundations of Software Engineering (ESEC/FSE-13), Lisbon, Portugal, 2005, pp. 286–295.

[7] Liblit, B.—Aiken, A.—Zheng, A. X.—Jordan, M. I.: Bug Isolation via Remote Program Sampling. Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03), New York, USA, 2003, pp. 141–154.

[8] Jones, J. A.—Harrold, M. J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), Long Beach, CA, USA, 2005, pp. 273–282.

[9] Zhang, X.—Gupta, N.—Gupta, R.: Locating Faults through Automated Predicate Switching. Proceedings of the 28th International Conference on Software Engineering (ICSE '06), Shanghai, China, 2006, pp. 272–281.

[10] Zeller, A.—Hildebrandt, R.: Simplifying and Isolating Failure-Inducing Input. IEEE Trans. Softw. Eng., IEEE Press, Vol. 28, 2002, No. 2, pp. 183–200.

[11] Zeller, A.: Isolating Cause-Effect Chains from Computer Programs. Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software (SIGSOFT '02/FSE-10), Charleston, South Carolina, USA, 2002, pp. 1–10.

[12] Liblit, B.—Naik, M.—Zheng, A. X.—Aiken, A.—Jordan, M. I.: Scalable Statistical Bug Isolation. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05), Chicago, USA, 2005, pp. 15–26.

[13] Arumuga Nainar, P.—Chen, T.—Rosin, J.—Liblit, B.: Statistical Debugging Using Compound Boolean Predicates. Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07), London, UK, 2007, pp. 5–15.

[14] Gupta, N.—He, H.—Zhang, X.—Gupta, R.: Locating Faulty Code Using Failure-Inducing Chops. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), Long Beach, CA, USA, 2005, pp. 263–272.

[15] Zhang, X.—Gupta, N.—Gupta, R.: Pruning Dynamic Slices with Confidence. SIGPLAN Not., ACM Press, Vol. 41, 2006, No. 6, pp. 169–180.

[16] Jeffrey, D.—Gupta, N.—Gupta, R.: Fault Localization Using Value Replacement. Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08), Seattle, USA, 2008, pp. 167–178.

[17] Zhang, X.—Gupta, N.—Gupta, R.: Locating Faulty Code by Multiple Points Slicing. Softw. Pract. Exper., John Wiley & Sons Press, Vol. 37, 2007, No. 9, pp. 935–961.

[18] Software Infrastructure Repository (SIR). Availaible on: `http://www.cse.unl.edu/_galileo/sir`.

[19] Diablo. Availaible on: `http://www.elis.ugent.be/diablo/`.

[20] Valgrind. Availaible on: `http://valgrind.org/`.

[21] Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann Press, 2005.

[22] Weka Machine Learning Software. Availaible on: http://cs.waikato.ac.nz/ml/weka/.

[23] Do, H.—Elbaum, S.—Rothermel, G.: Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. Empirical Softw. Eng., Kluwer Academic Publishers, Vol. 10, 2005, No. 4, pp. 405–435.

[24] Zhang, X.—Tallam, S.—Gupta, N.—Gupta, R.: Towards Locating Execution Omission Errors. Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), San Diego, USA, 2007, pp. 415–424.

[25] Xu, R.—Wunsch, D.: Clustering. Wiley-IEEE Press, 2009.

[26] Wilcox, R. R.: Introduction to Robust Estimation and Hypothesis Testing. Acad. Press, 1997.

[27] Wang, X.—Cheung, S. C.—Chan, W. K.—Zhang, Z.: Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization. Proceedings of the 31st International Conference on Software Engineering (ICSE '09), Vancouver, Canada, 2009, pp. 45–55.

[28] Kranzlmüller, D.: Incremental Tracing and Process Isolation for Debugging Parallel Programs. Computing and Informatics, Slovak Academy of Sciences, Vol. 19, 2000, No. 6, pp. 569–585.

[29] Zhang, Z.—Chan, W. K.—Tse, T. H.—Yu, Y. T.—Hu, P.: Non-Parametric Statistical Fault Localization. J. Syst. Softw., Elsevier Science Press, Vol. 84, 2011, No. 6, pp. 885–905.

[30] Abreu, R.—Zoeteweij, P.—Golsteijn, R.—van Gemund, A. J. C.: A Practical Evaluation of Spectrum-Based Fault Localization. J. Syst. Softw., Elsevier Science Press, Vol. 82, 2009, No. 11, pp. 1780–1792.

[31] Wong, W. E.—Debroy, V.—Golden, R.—Xu, X.—Thuraisingham, B.: Effective Software Fault Localization Using an RBF Neural Network. IEEE Transactions on Reliability, IEEE Press, Vol. 61, 2012, No. 1, pp. 149–169.

[32] Wong, W. E.—Debroy, V.—Xu, D.: Towards Better Fault Localization: A Crosstab-Based Statistical Approach. Systems, Man, and Cybernetics – Part C: Applications and Reviews, IEEE Press, Vol. 42, 2012, No. 3, pp. 378–396.

[33] Jones, J. A.—Bowring, J. F.—Harrold, M. J.: Debugging in Parallel. Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07), New York, USA, 2007, pp. 16–26.

[34] Zheng, A. X.—Jordan, M. I.—Liblit, B.—Naik, M.—Aiken, A.: Statistical Debugging: Simultaneous Identification of Multiple Bug. Proceedings of the 23rd International Conference on Machine Learning (ICML '06), Pittsburgh, USA, 2006, pp. 1105–1112.

[35] Cunha, J. C.—Krawczyk, H.: Testing and Debugging of Distributed Software. Computing and Informatics, Slovak Academy of Sciences, Vol. 19, 2000, No. 6, pp. 495–510.

Saeed Parsa received his B.Sc. in mathematics and computer science from Sharif University of Technology, Iran, his M.Sc. degree in computer science from the University of Salford in England, and his Ph.D. in computer science from the University of Salford, England. He is Associate Professor of computer science at the Iran University of Science and Technology. His research interests include software engineering, soft computing and algorithms.

Mojtaba Vahidi-Asl received his B.Sc. degree in software engineering from Tehran Polytechnic in 2005, and the M.Sc. degree in software engineering from Iran University of Science and Technology in 2008. He is currently a Ph.D. student at the Department of Computer Engineering at Iran University of Science and Technology. His research focus is on developing statistical algorithms to improve software quality with an emphasis on statistical fault localization and automated test case generation. His other interests are scam filtering, text and graph mining.

Farzaneh Zareie received her B.Sc. degree in computer engineering (software) from Bu-Ali Sina University in 2008. Her research was supervised by Mrs. Narges Bathaeian and had focused on design and development of an educational software for compiler concepts. She continued her study toward the Master's degree at the Iran University of Science and Technology (IUST) under the supervision of Prof. Saeed Parsa in 2011. She was a member of software testing research group in Prof. Parsa's Parallel Processing Lab Science 2009 and her research field is accurate bug localization techniques using program slicing.