Dichotomy method in testing-based fault localization

Sun Ji-Rong, Ni Jian-Cheng, and Li Bao-Lin

***Abstract*—** In practice, testing-based fault localization (TBFL), which uses test information to locate faults, has become a research focus in recent years. Dichotomy method is presented to perform on TBFL. First we optimize the test information itself from three aspects: searching scope localization using slice technique, redundant test case removal, and test suite reduction with nearest series. Secondly, the diagnosis matrix is set up according to the optimized test information and each code in failed slice is prioritized accordingly. Thirdly, the dichotomy method is iteratively applied to an interactive process for seeking the bug: the searching scope is cut in two by the checking point *cp*, which is of highest priority in searching scope; If *cp* is wrong, the bug is found; else we will ignore the codes before/after it according to the result of *cp*. Finally, we conduct three studies with Siemens suite of 132 program mutants. Our method scores 0.85 on average, which means we only need to check less than 15% of the program before finding out the bug.

***Keywords*—**diagnosis matrix, dichotomy method, execution slice, testing-based fault localization (TBFL), test suite optimization

I. Introduction

TO improve the quality of a program, we have to remove as many defects as possible in it without introducing new bugs at the same time. However, localizing a fault is a complex and time-consuming process. To reduce the cost on debugging, it is natural to automate fault localization using information acquired from testing, which is referred to as testing-based fault localization (called TBFL in this paper).

In practice, no clear continuity exists between the testing task and diagnosis one, i.e., locating faults in the program code. While the former aims at generating a minimal test suite with a high fault-revealing power, the latter uses, when possible, all available symptoms (e.g. traces) coming from testing to locate and correct the detected faults. The richer the information coming from testing, the more precise the diagnosis may be.

To reduce the human effort, many approaches have been proposed in recent years to automate fault localization based on the analysis of execution traces, such as ①Dicing[1],[2], ②TARTANTULA[3],[4], ③Interactive approach[5],[6], ④Nearest Neighbor Queries approach[9], ⑤SAFL[7],[8], ⑥Control-flow based Difference Metric approach [10], and ⑦our previous incremental approach[11] etc.

All approaches except ③ are automatic carried out to give a report of the most suspicious codes or the possibility of each code containing bug and check the report one by one. While approach ③ is an interactive process, the information gathered from previous steps can be used to provide the ranking of suspicious statements for the current interaction step.

In approach ④ and ⑥, only one successful test case, most similar to the failed one, is selected out according to some metrics. Then the difference between these two execution traces will determine the report. In approach ① and ⑦, several successful traces will be picked up to help to prioritize the code. For each code, more times to appear in the successful slice, less impossible to contain any bug. In approach ②, ③ and⑤, an entire test suite is used to color the codes in the failed program, different color with different brightness stands for different possibility of containing bug.

Most of the research in this topic has focused on how to compare the successful and failing execution traces. But the effectiveness of a fault localizer is largely depends on the quality of the given test suite or selected test case(s) themselves. Suppose we have access to the failed test case and its execution trace (called *wt* and S_{wt} in this paper), how to maximizing the utility of test information is above all the first and most important question in TBFL technique. Approach ① to ④ will be compared with ours in this paper.

II. Preliminary Work

Typically, the TBFL problem can be formalized as follows [5]. Given a program, which is composed of a set of statements (denoted as $P=\{s_1,s_2,\ldots,s_m\}$), and a set of test cases (denoted as $T=\{t_1,t_2,\ldots,t_n\}$), the information acquired when running these test cases against the target program can be represented as a n*(m+1) boolean execution matrix(called a **diagnosis matrix** in this paper), denoted as $E=(e_{ij})$ $(1\leqslant i\leqslant n,\ 1\leqslant j\leqslant m)$, where

$$e_{ij}=\begin{cases}1 & \text{statement } s_j \text{ is executed by test } t_i \quad (1\le j\le m)\\ 1 & \text{test case } t_i \text{ is successful} \quad (j=m+1)\\ 0 & \end{cases} \tag{1}$$

Thus, the TBFL problem can be viewed as the problem of calculating which statements are most suspicious based on the diagnosis matrix.

S. Ji-Rong is with Sichuan Radio and TV University, Chengdu, CO 610073, China (phone: +86-13378115339; e-mail: sunjirong@126.com).

N. Jian-Cheng is with School of computer science, Sichuan University, Chendu, CO 610065, China. (e-mail: nijch@163.com).

L. Bao-Lin is with School of computer science, Sichuan University, Chendu, CO 610065, China (e-mail: libaolin_2000@163.com)

The techniques described in this paper are based on the following observations [12],[13]:

1) If a statement is not executed under a test case, it cannot affect the program output for that test case.
2) Even if a statement is executed under a test case, it does not necessarily affect the particular output.
3) The likelihood of a piece of code containing a specific fault is proportional to the number of failed tests that execute it.
4) The likelihood of a piece of code containing a specific fault is inversely proportional to the number of successful tests that execute it.

A. Sample Program

Let's have a sample program *P* in Fig. 1 and its diagnosis matrix in Table 1 to see how to use the test information; it will be further used to show how to prioritize the code and how to locate the bug in the following sections.

```
s1: read(a,b,c);
s2: class:=scalene;
s3: if a=b or b=c
s4:   class:=isosceles;
s5: if a=b and b=c
s6:   class:=equilateral;
s7: if a*a=b*b+c*c
s8:    class:=right;
s9: case class of
s10:  right    : area:=b*c/2;
s11:  equilateral : area:=a*2*sqrt(3)/4
s12:  otherwise :  s:=(a+b+c)/2;
s13:            area:=sqrt(s*(s-a)(s-b)(s-c));
      end;
s14: write(class,area);
```

Fig. 1 example program P

The program in Fig. 1 reads the lengths of three sides of a triangle, classifies the triangle, computes its area, and outputs the class and the area computed.

Table 1 diagnosis matrix of program P

	t1 (2,2,2)	t2 (4,4,3)	t3 (5,4,3)	t4 (6,5,4)	t5 (3,3,3)	t6 (4,3,3)
s1	1	1	1	1	1	1
~~s2~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
~~s3~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
~~s4~~	~~1~~	~~1~~	~~0~~	~~0~~	~~1~~	~~1~~
s5	1	1	1	1	1	1
s6	1	0	0	0	1	0
~~s7~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~
~~s8~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~0~~
s9	1	1	1	1	1	1
~~s10~~	~~0~~	~~0~~	~~1~~	~~0~~	~~0~~	~~0~~
s11	1	0	0	0	1	0
~~s12~~	~~0~~	~~1~~	~~0~~	~~1~~	~~0~~	~~1~~
~~s13~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~	~~0~~
s14	1	1	1	1	1	1
S/F	1	1	1	1	0	1

Table 1 gives the test cases in T and its corresponding execution path. The program produces correct outputs on all test cases except t5, which is marked grey in Table1. Because s11 uses the expression a*2 instead of a*a.

B. Using slice to cross out irrespective statements

An **execution slice** with respect to *wt* is the set of code executed by *wt*. Based on Observation **1)**, control didn't reach the statements 8,10,12,13 as shown in Fig.2 during the execution of t5, we can be sure that the error could not be brought by those statements. Thus the row 8,10,12,13 will be first crossed off with single-lines illustrated in table 1 during the diagnosis process.

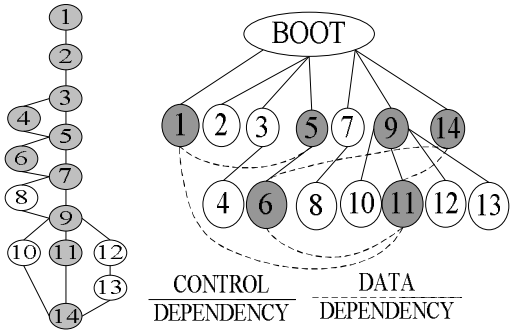


Fig. 2 program *P*'s DFD and CFG with respect to test case t5

A **dynamic slice** uses dynamic analysis to identify all and only the statements that contribute to the selected variables of interest on the particular anomalous execution trace. In this way, the dynamic slice technique prune away unrelated computation and the size of slice can be considerably reduced, thus allowing an easier location of the bugs [14]. *P*'s CFG is shown on the right in Fig. 2 and t5's dynamic slice is marked grey, which can be obtained according to the dynamic control dependency and data dependency. According to observation 2), even statements 2,3,4,7 are executed but do not affect variable *area*. So the bug must exist in *area*'s dynamic slice with respect to t5. Thus the row 2,3,4,7 will be further crossed off with double-line illustrated in table 1 during the diagnosis process.

Let's use S_{wt} to stand for the slice of *wt*, maybe execution or dynamic. We only need to focus on the statements in S_{wt}.

III. Test Suite Optimization

Given such a failing run, fault localization often proceeds by comparing the failed run with successful run or runs, that is, a run which does not exhibit the unexpected behavior. Because only one fault-revealing execution trace is considered in the manual fault localization process, then how to exploit multiple execution traces becomes the starting point of TBFL.

A. Wrong and Right test suite

However, TBFL approaches have an obvious shortcoming. The effectiveness of these approaches depends on the distribution of the test cases in the test suite. Intuitively, the more test cases in test suite, the more accuracy of the TBFL report. But if several pieces of code in different places are executed by the same test cases (called indistinguishing statements), those codes will be given the same rank whatever approaches taken. Experiments in [7],[15] show more indistinguishing statements, less accurate diagnosis.

Let us use *Req(t,wt)* to stand for the common statements both executed by the test case *t* and *wt*, i.e.,

$$Req(t,wt)=S_t \cap S_{wt} \quad (2)$$

Bigger $|Req(t,wt)|$ is, more codes in execution of *t* and *wt* are

common, i.e, that means more "nearest" to *wt*[9].

We will create two subsets of *T*: ***Right*** and ***Wrong***. Each failed test case will be allocated to *Wrong*. All successful test cases will be allocated to *Right* according to the following rules:

(1)If several test cases' executions are the same in *Right*, only one will be included in *Right* set.

(2)If its execution is the same as anyone in *Wrong*, ignore it.

(3)At last the test cases in *Right* will be ordered according to the number of $|Req(t,wt)|$, the maximum is the first, the minimum is the last.

Because all the failed test cases need fault localization, we only need to exclude the extra successful test cases. They are redundant, useless and even harmful to TBFL. Thus test case t1,t6 will be ignored in diagnosis process.

B. Test suite reduction toward TBFL

Reducing the testing effort implies generating a minimal test suite for reaching the given criterion. While an accurate bug diagnosis requires maximizing the information coming from testing process for a precise cross-checking and fault localization. However, experiments in [7],[15] have shown that redundant test cases may bias the distribution of the test suite and harm TBFL. We propose to use test suite reduction techniques to remove some harmful redundancy to boost TBFL.

Suppose we aim at finding one faulty statement each time. As described in previous section, we only need to check the code in S_{wt}. Given the diagnosis matrix *E* as input and the codes in S_{wt}, reduce the test suite T to T':

(1)$T'=\Phi$.

(2) The test cases are allocated to *Right* or *Wrong* based on *E* and $S_{wt.}$ according to section Ⅲ.A.

(3)Select the test case from *Right* in-sequence which means the nearest to *wt* into T' until all statements in S_{wt} are covered.

(4)The test cases in *Wrong* must be existed in diagnosis process, so T'=T'+*Wrong*.

C. Optimized diagnosis matrix

Since some irrespective statements have been ruled out and some harmful redundant test cases are got rid off from T, so we need to build up a new diagnosis matrix E' aiding to rank the statements of S_{wt} toward TBFL. The test cases in T' and the statements in S_{wt} will only be evolved in E'.

For example, if we take up the dynamic slice technique with t5, then we can obtain a new optimized diagnosis matrix as shown in the left of Table 2.

Table 2 optimized diagnosis matrix and priority(s) of S_{t5}

Si	optimized diagnosis matrix				%S	%F	Prior(si)
	t2 (4,4,3)	t3 (5,4,3)	t4 (6,5,4)	t5 (3,3,3)			
1:s1	1	1	1	1	100%	100%	0.5
2:s5	1	1	1	1	100%	100%	0.5
3:s6	0	0	0	1	0	100%	1
4:s9	1	1	1	1	100%	100%	0.5
5:s11	0	0	0	1	0	100%	1
6:s14	1	1	1	1	100%	100%	0.5
S/F	1	1	1	0			

IV. Dichotomy Method in TBFL

Experimental results show that the developer needs to examine up to 20% of the total statements for less than 60% of faults and examine more statements for other faults [3],[9],[10]. That is, even the automatic TBFL report is with a high score, for a middle-sized program composed of several thousand statements, the developer has to examine several hundred statements to find the location of one fault. Therefore, current TBFL approaches still can hardly serve as an effective substitute of manual fault localization. We focus on how to combine the merit of manual fault localization and TBFL. To achieve this, we proposed a **Dichotomy Method** in TBFL.

A. Code prioritization in S_{wt}

Let's use F(s) to be a subset of *Wrong* and any test case in F(s) executes the statement s. We use %Failed(s) to stand for the ratio of failed test cases that execute s. So according to observation 3), it means the error possibility of s.

$$\%\,Failed(s) = \frac{|F(s)|}{|Wrong|} \times 100\% \tag{3}$$

Let's use R(s) be a subset of *T'* and any test case in R(s) is successful and executes the statement s. We use %Successful(s) to represent the ratio of successful test cases in T' that execute s. According to observation 4), it means the correct possibility.

$$\%\,Successful\ (s) = \frac{|R(s)|}{|T'|-|Wrong|} \times 100\% \tag{4}$$

Let's use *Priority(s)* stand for the possibility of statement s containing a bug. It can be obtained by combining the equation (3) and (4) as follows:

$$Priority(s) = \frac{\%\,Failed(s)}{\%\,Failed(s) + \%\,Successful\ (s)} \tag{5}$$

Thus all the statements in S_{wt} are to be ranked with a priority. Intuitively, the higher *Priority(s)* is, the more suspicious a statement s is, and thus it should be examined earlier. Any TBFL tool described above rank the statement directly without optimizing the testing information and will be ceased at this step, a report with a list of code's priority turned in. Then the code will be checked one by one from the highest priority to the least. While our TBFL tool is a semi-automatic, each time which statement is to be checked depends on the checking result last time.

The priority of each code in S_{wt} with respect to t5 is listed in the right of Table 2.

B. Dichotomizing search method in TBFL

We call our TBFL technique as ***dichotomy method*** for it is similar with finding for an item in an ordered list using dichotomizing search. The architecture of dichotomizing search method in TBFL is illustrated in Fig. 3.

The whole process can be elaborated as follows:

① Acquiring diagnosis matrix E: Given the traces and results of each test case obtained from testing process as input, it will out put the diagnosis matrix E;

②Acquiring slice of *wt*: Given a failed test case *wt*, its output variable *v* is not correspondence with the expected. Then we try

to find out what makes *v* abnormal and fault localization is needed. Suppose we aim at one default at a time. If use execution slice technique, no other resource needed and the trace of *wt* is $S_{wt.}$; If use dynamic slice, the dynamic slice S_{wt} will be acquired with the help of slicing tools. We can preliminary conclude that the bug exists in S_{wt}. As described in section Ⅲ, it will be further used in test suite reduction and diagnosis matrix optimization, i.e. the covering criterion R. In section Ⅴ, we will compare the effectiveness of different technique.

③Acquiring *Right* and *Wrong*: Allocating each test case in the initial test suite T to *Right* and *Wrong* individually according to the rules elaborated in section Ⅲ.A.

④Test suite reduction: It has been proved in [7],[15] that redundant test case will harm the accuracy of TBFL. We use the greedy algorithm to select out nearest test cases from T to T'.

⑤Acquiring optimized diagnosis matrix E': After some redundant information from testing process, a new diagnosis matrix E' is required to rank the code in S_{wt}. The statements are listed in the same order as the program execution and then to be re-numbered for convenience.

⑥ Prioritizing code: In this step, given an initial or optimized diagnosis matrix *E* as input, each code in *E* will be evaluated with a priority. Then a report will be turned in with the ranked code list. We let a=0, b=|*E*|. The index a,b are used to present the searching range. The bug must exist in the statements between a and b. If the input is an optimized diagnosis matrix, then we only need to check the code in S_{wt}, otherwise in whole program.

⑦Setting the checking point *cp*: Looking up the report, find a statement *cp* with maximum *Priority(cp)* from a to b and make it a checking point. We use ***pred(cp)*** and ***succ(cp)*** to stand for the variables before and after *cp* separately. We are sure that *cp* is the most possible of containing fault. The variables that are accessed by the checking point *cp* are examined carefully, especially those contribute to the error output *v*. If there are several statements ranked with the same highest priority, then we pick the last one as checking point *cp*. Because we think the code in the earlier has been checked more times, it is less to be faulty. Variable *k* is used to count the iteration times, which means how many statements have been examined before the bug found. In table2, s6 and s14 are ranked the same, so first *cp* is set to s14. And it is faulty, the algorithm terminates.

⑧ Dichotomizing the searching range: The different conditions are to be analyzed: **1)**If these values are already incorrect before executing *cp*, i.e. *pred(cp)* is error, we can assume that there should be a faulty statement before *cp*. Thus the codes after *cp* are innocent to the bug, b=*cp*-1; **2)** If both *pred(cp)*and *succ(cp)* are correct, we would assume that no faulty statement has been executed yet, a=*cp*+1. **3)** If *pred(cp)* is correct and *succ(cp)* is incorrect, we may determine that the statement *cp* is a faulty one, thus the algorithm terminates and the bug is found. Otherwise go to ⑦ for next iteration.

From the beginning, we have supposed to find out one bug a time. We can use the *dichotomy method* to gradually narrow down the searching range. Steps ② to ⑤ are marked within dashed rectangle in Fig.3. Those steps aim to maximize the utility of testing information and to give an accurate report. Steps ⑥ to ⑧ are used to dichotomize the searching range. Using the statement with highest rank (i.e. *cp*) to split the suspect range in two, we look into the variables accessed by *cp* to see whether abnormal occurs. Thus the next binary search range is determined.

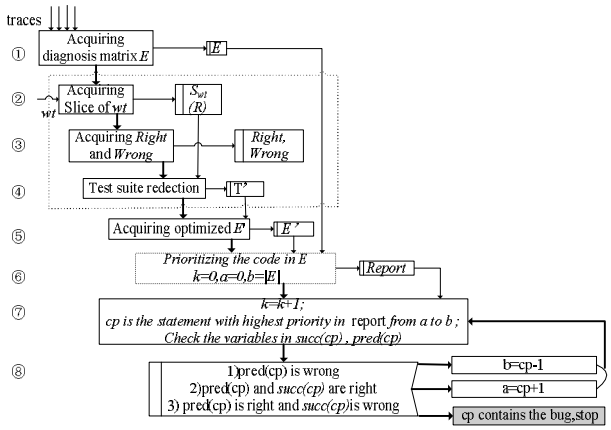


Fig.3 architecture of dichotomizing search method in TBFL

V. EXPERIMENT

We apply our *dichotomy method* together with Approach ① to ④. Among them, ①, ②and ④ are automatic and end with a report of ranking list, while ③and ours are semi-automatic to continue with the report for further localization.

To investigate the effectiveness of our technique and guide our future work, we considered three questions:

1. How many check points has been set before finding out the faulty code with different slice technique? Does dynamic technique really outperform the execution one?
2. Does diagnosis matrix optimization really work? How does the quality of test suite itself influence the effect of TBFL tool?
3. Does our method really overwhelm others in quality?

To investigate the effectiveness of our ***dichotomy method*** in finding out the faulty statements, we have designed a number of studies. The first set of studies investigates the effectiveness of *dichotomy* technique when using execution slice and dynamic slice separately, at this study, the initial test suite is used. The second set of studies evaluates the effectiveness of diagnosis optimization. The third set of studies compares the accuracy of our method with other methods.

A. Programs under test

The target programs used in our experiment are the Siemens programs, which were first collected and used by Hutchins et al.[13] and were also used in the experiments of some other fault localizer [5],[9],[10],[15].

The Siemens programs consist of 132 versions of seven programs in C with injected faults, as shown in Table 3.Their Lines of Code(LOC) range from 138 to 516. Each faulty version has exactly one injected fault. Some of the faults are code omissions; some are stricter or looser conditions in the statements; some focus on the type faults of variables, and so on. The Siemens programs not only have various faulty versions, but also have the correct version for each faulty program. Because all those 5 approaches in our experiment are testing-based fault localization, we try to imitate the true testing process. For each program, its initial test suite T is first created by the black-box test suite and then more test cases are manually added in to ensure that each executable statement, edge, and definition-use pair in the is covered by at least 30 different test cases. The overview of the Siemens program suite is shown in Table 3. The last four columns show LOC, the number of test cases in T and failed test cases in *Wrong*, the number of faulty versions, respectively.

Table 3 relative data of the experimental program

No.	Program	Descption	LOC	|T|	|*Wrong*|	NoV
P1	Replace	Pattern replacement	516	5,542	3–309	32
P2	Printtokens	Lexical analyzer	402	4,130	6–186	7
P3	Printtokens2	Lexical analyzer	483	4,115	33–518	20
P4	Schedule1	Priority scheduler	299	2,650	7–293	9
P5	Schedule2	Priority scheduler	297	2,710	2–65	10
P6	Tot_info	Information measure	346	1,052	3–251	23
P7	Tcas	Altitude separation4	138	1,608	1–131	41

In our experiment, we use the initial test suite T of each Siemens programs to execute its faulty versions. For each faulty version, record the corresponding execution matrix *E*.

B. Evaluation Framework

Based on the execution matrices, in our method, we first reduce the test suite with respect to *wt* (i.e. a faulty version) and then in the optimized diagnosis matrix E', a report of ranked statement list can be obtained; While the other four above methods directly analyze the execution matrices to obtain a report of ranked statements list. Because the faulty statement is injected manually, we can know where it is before hand. Thus from the report, we can directly know how many codes will be examined or how many check points will be set before the bug has been found.

To evaluate the performance of these different TBFL tools, we need a quantitative measure of a fault localizer's report quality. Formally, given a report of a mutant, that is, a list of program features the localizer indicates as possible locations of the bug, we want to compute and assign a score to it. We adapt the evaluation technique that was originally proposed by Renieris and Reiss [9] and later adopted by Hao et al [5] and Guo et al [10]. For each fault, this measure defines a score that indicates how much portion of code does not need to be examined for finding its location. For example, for a faulty program consisting of 100 statements, if we need to examine 5 statements before we find the faulty one, the score is 0.95. The higher this score is, more accurate the report is and the better the performance is.

We use *Score*(*wt*) to stand for the effectiveness of a method with respect to *wt* when checking the bug in a given version. Using all mutants of a program, the average diagnosis accuracy is computed, it estimates the quality of the diagnosis method. Thus the score of a method for *P* is the average of scores of all versions, which can be calculated by equation (6).

$$Score(P) = \frac{\sum Score(wt)}{|\ NoV\ |} \tag{6}$$

C. Study 1: execution slice vs. dynamic slice

We conduct the experiment in this study with the initial test suite, aiming at comparing the effectiveness of slice technique in finding the bug.

First Steps ②,③,④ and ⑤ are ignored, we only use *dichotomy method* to seek the bug in the whole program from initial diagnosis matrix. The score's distribution of the report is listed in Table 4 column Program(T). And then step ② is added, execution slice and dynamic slice are applied on *wt* individually and the codes in S_{wt} only need be examined for each version. The results are shown in last two column of Table 4.

Table 4 Versions(%) of average Score distribution

Score	Program(T)	Execution(T)	Dynamic (T)
0-0.1	3.03	2.27	2.27
0.1-0.2	9.09	1.52	1.52
0.2-0.3	1.52	0.00	0.00
0.3-0.4	1.52	0.00	0.00
0.4-0.5	0.76	0.00	0.00
0.5-0.6	4.55	0.76	0.76
0.6-0.7	6.06	5.30	3.79
0.7-0.8	11.36	12.88	7.58
0.8-0.9	19.70	25.00	28.79
0.9-1.0	42.42	52.27	55.30

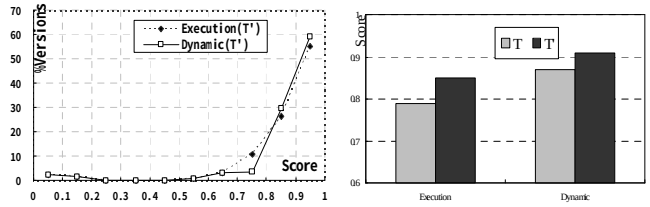
Table 4 compares the distribution of scores with non-slice, execution slice and dynamic slice over the percentage of total versions. We found that slice technique really does work in comparing with non-slice and dynamic slice technique do performs better than execution one. When examining less than 20% statements of the program, only 62.12% fault can be found with non-slice technique, while 77.27% faults with execution slice, and 84.09% with dynamic slice.

For a given failed test case *wt* in relation to one version, there maybe more than one failed test case at the same time. The codes in *P* but out of S_{wt} can also be executed by other failed test cases, thus can be ranked with a higher priority, which may mislead the diagnosis when the searching range is not limited to S_{wt}. So first to local the searching range in S_{wt} is very essential.

D. Study 2: T vs. T'

From steps ③ to ⑤, we have tried to get rid of the harmful information and furthest utilize the information from testing process. In this study, we try to estimate how those steps contribute to the scores of our *dichotomy method*. Based on study 1, steps ③ to ⑤ are added in to acquire T' of each version. When optimizing the diagnosis matrix in step ⑤, we adopt the execution and dynamic slice separately.

Thus the score of each version can be calculated with optimized E' and its distribution is shown in Fig. 4(a), where execution and dynamic slice technique is in comparison. With either slice technique, more than 50% bugs can be found in less than 10% of the program and more than 80% bugs can be found in less than 20% of the program.



(a) Execution vs. Dynamic (b) T vs. T'

Fig.4 effectiveness of test suite reduction in dichotomy method.

Combine with the result in study 1, the scores of *dichotomy method* regarding T and T' can be acquired respectively, which are illustrated in Fig.4(b). The test suite optimization really works, which promote about 4% diagnosis accuracy.

E. Study 3: Dichotomy method vs. other TBFL tool

As accquiring dynamic slice needs more resources and the execution trace (slice) of each test case has already existed and also be used in other TBFL tools, to be fair, we will take the execution slice of *wt* in study 3 in contrast.

The scores of Approach ① to ④ can be obtained from reference [5],[9],[10]. Then the effectiveness of those 5 TBFL tools is illustrated in Fig 5. The score of our *dichotomy method* outperforms the others at least by 30% including the interactive approach ③.

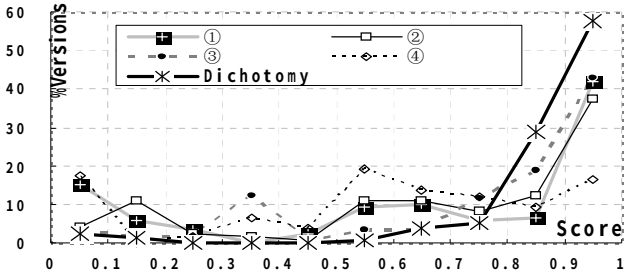


Fig. 5 effectiveness of different TBFL tool

We have also found that there exist some bugs which can be located nearly whole program having been examined, whatever approach adopted.

VI. CONCLUSION AND FUTURE WORK

Testing-based fault localization (TBFL) has become a research focus in recent years. In this paper, we present the details of our *dichotomy method* in TBFL.

First we limit the searching scope to the slice S_{wt} of a given failed test case *wt*. Second we optimize the initial test suite to exclude some redundant and harmful test cases. Then only nearest test case which contributes most to a given fault in S_{wt} is selected out to cover the statement in S_{wt}. Third the diagnosis matrix is accordingly optimized to rule out the useless information from the testing process. And each code in S_{wt} will be evaluated with a priority of containing a fault based on diagnosis matrix. At last, we extend the TBFL tool to use an interactive process to handle the information gathered from the previous interaction step to decide which suspicious statement, i.e., checking point *cp*, next to be examined. In each iteration, *dichotomy method* is applied to decrease the searching scope: Looking into the diagnosis matrix, select out a statement as *cp* with highest priority within the searching scope, and inspect the variables before and after *cp* carefully. If the variables before *cp* is right and after *cp* is wrong, then *cp* is the faulty code, algorithm terminates; If the variables before *cp* is wrong, then checking scope will be set before *cp*; If the variables before and after *cp* both are right, then checking scope will be set after *cp*.

Based on the results of three studies that evaluate the effectiveness of *dichotomy method*, we find that each step in our technique does contribute to helping locate suspicious statements and our technique does outperform the others at least by 30% in score.

The studies also suggest some directions for our future work: (1) what kind of fault in a program is always with a low score nearly to 0? (2) When several faults are injected to a version at the same time, could our method preserve to be superior to others?

REFERENCES

[1] H.Agrawal, J.Horgan, S.London, and W.Wong. Fault Localization using Execution Slices and Dataflow Tests. Proceedings of ISSRE'95(Int. Symposium on Software Reliability Engineering), Toulouse, France, October 1995.

[2] W. E. Wong, Y. Qi, An execution slice and inter-block data dependency-based approach for fault localization. Proceedings of the 11th Asia-Pacific Software Engineering Conference 2004 (APSEC'04), pp.366-373.

[3] J.A.Jones, M.J.Harrold, and J.Stasko. Visualization of Test Information to Assist Fault Localization. In Proc. of the 24th International Conference on Software Engineering, May 2002, pp.467-477,

[4] J.A.Jones and M.J.Harrold. Empirical Evaluation of the Tarantula Automatic Fault Localization Technique. Proceedings of ASE'05(Automated Software Engineering), Long Beach, California, USA, November 2005.

[5] D.Hao, L.Zhang, H.Zhong, H.Mei and J.Sun. Towards Interactive Fault Localization Using Test Information. In Proc. of the 13th Asia Pacific Software Engineering Conference, 2006. APSEC 2006. Dec. pp.277-284

[6] D.Hao. Testing-Based Interactive Fault Localization. In Proc. of the the 28th international conference on Software engineering 2006, pp.957-960

[7] D.Hao, L.Zhang, H.Zhong, H.Mei and J.Sun. Eliminating Harmful Redundancy for Test-Based Fault Localization using Test Suite Reduction: An Experimental Study. In Proc. of the International Conference on Software Maintenance,Sep.25-30,2005.

[8] D.Hao, Y.Pan, L.Zhang, W.Zhao, H.Mei and J. Sun. A Similarity-Aware Approach to Testing Based Fault Localization. In Proc. of the 20th IEEE International Conference on Automated Software Engineering, November,pp.291-294,2005.

[9] M.Renieris and S.P.Reiss. Fault Localization with Nearest Neighbor Queries. In Proc. of International Conference on Automated Software Engineering, pp.30-39,2003.

[10] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In CC, 2006.

[11] J. Sun, Z. Li, J. Ni, F.Yin. Priority Strategy of Software Fault Localization. In Proc. of the 6th WSEAS International Conference on APPLIED COMPUTER SCIENCE. (ACOS '07), pp. 500-506

[12] H. Agrawal, J.R. Horgan, E.W. Krauser, Incremental regression testing. Proceeding of the Conference on Software Maintaiance, 1993,pp.299-308.
[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and control flow-based test adequacy criterions. In Proc. of the 16th Int'l. Conf. on software. Eng., pp. 191-200, May 1994.
[14] H. Agrawal, R. A. DeMillo, and E. H. Spafford, Dubugging with Dynamic Slicing and Backtracking, Software-Practice & Experience, 23(6):589-616, June, 1996
[15] B. Baudry, F. Fleurey, and Y. Le Traon. Improving Test Suites for Efficient Fault Localization. In Proc. of International Conference on Software Engineering, *(ICSE '06),* May 2006, pp. 82-91