Effective Fault Localization Using Weighted Test Cases

Yihan Li, Chao Liu
School of Computer Science and Engineering
Beihang University
Beijing, China
Email: {liyihannew, liuchao}@sei.buaa.edu.cn

Abstract— **Locating faults in a program is prohibitively time-consuming and tedious, and therefore, many automated fault localization techniques have been proposed to assist in the debugging process. Spectrum based fault localization are promising techniques that can guide developers to the possible locations of faults. These techniques make a summary on the number of passing and failing tests cases to prioritize suspicious statements according to likelihood of containing program bugs for each statement. Though results are encouraging, these techniques treat all test cases as equally important, which ignore individual error diagnosis ability for different test cases. In this paper, we present an approach to exploit varying weights for individual test cases in the computation of suspiciousness scores so as to improve the effectiveness of spectrum based fault localization techniques. To validate our method, experiments were performed on eight typical SFL techniques using two standard benchmarks. Results are suggestive that for the studied SFL techniques, our method can significantly improve the fault localization effectiveness in most situations and in other cases it does not introduce much adverse impact on the techniques original performance.**

Index Terms— **fault localization, program spectrum, debugging, passed tests, failed tests**

I. Introduction

It is a common phenomenon that no matter how much effort developers spend on testing a program, software faults are introduced and removed continually during software development processes. When faults are revealed during testing processes, software debugging is often used to remove as many faults in the program as possible so as to improve the quality of the program. Typically, software debugging involves locating faults, repairing faults and verifying repairs. Among the various debugging activities, locating faults has been recognized as one of the most expensive [1]. Therefore, over the last ten years there has been an explosion of work in automatic fault localization techniques [2]–[8] that assist developers in finding the locations of faults, thereby alleviating the work developers devoted in debugging.

In particular, Spectrum-base fault localization (SFL) is a low-cost and effective technique that tries to pinpoint the possible locations of faults. Essentially, SFL aims to correlate program entities with program failures via statistically analyzing coverage information. Specifically, SFL collects coverage information from a set of test cases together with their corresponding test results to form program spectrum, and then contrasts the coverage statistics of program entities between passed runs and failed runs using various statistical formulas. Finally, it yields a ranking list that rank all program entities in terms of suspiciousness. The program entities with higher suspicious value are more likely to contain bugs and thus are given higher priority for examination during software debugging. Based on coverage information, researchers have proposed various statistical formulas to measure the correlation between program entities and program failures, such as Tarantula [9], Ochiai [10] and Jaccard [10].

Although SFL approaches have brought encouraging results in locating faults, most of SFL approaches do not distinguish the contribution of test cases from each other. Basically, these approaches take information of the number of failed and passed tests into account and thus assume all test cases share equal importance, which ignore individual fault detection ability for different test cases and may limit effectiveness of fault localization. Take Tarantula as an example. Suppose that two statements are both exercised by the same number of passed tests and that of failed tests while are executed by different tests. In such case, the technique assigns same suspiciousness score to these two statements and thereby loses ability to distinguish these two statements in the ranking list, which may decrease its accuracy in predicting the locations of bugs. Furthermore, during testing, the distribution of passed tests and failed tests are always uneven. That is, the number of failed tests is often relatively smaller than that of passed tests. While SFL assigns same weights to all tests, the contribution of the faulty statement to failure will also be decreased largely in the case that the faulty statement is executed by relatively more passed tests, which may make the faulty statement ranked lower than some other non-faulty statements in the suspicious list.

In this paper, we extend our previous study in [11] and propose a weighting strategy to measure contribution of different test cases so as to mitigate noise induced by tests. Our method distinguishes the weight of one failed test case from anther or one passed test case from another. For each failed test, different weights are assigned with respect to each statement according to its coverage information and proportion between failed tests and passed

© 2014 ACADEMY PUBLISHER
doi:10.4304/jsw.9.8.2112-2119

tests. For each passed test, weights are assigned according to its average similarity to all failed tests. In such a way, our method can be generally applied to existing SFL techniques, such as Tarantula et al., to improve fault localization effectiveness. We use Siemens programs and three Unix programs as our subject programs to evaluate our strategy, and compare performance of several fault localization techniques using the conventional formulas and our refined formulas respectively. The empirical results show that our method is promising on the studies subject programs. Further analysis shows that for the studied metrics, our approach can significantly improve their performance in locating faults. The contributions of this paper can be summarized as:

1) We propose an approach to quantify weights for failed tests and passed tests respectively according to coverage information so that statistics of data from tests is enriched which can benefit fault localization. Our approach can be easily applied to existing statistical formulas without much modification.
2) We evaluate the effectiveness of our weighting approach systematically across two standard benchmarks and compare performance of several SFL techniques using the conventional method and our proposed method. Comparisons are also made with peer woks.

The rest of the paper is organized as follows. Section II introduces some related works in fault localization. Section III presents our techniques. We detail our weighting scheme in this section. Section IV providers our experiment and analysis. Finally, Section V concludes our work and presents future work.

II. Related Works

In this section, we briefly review previous studies related to fault localization.

Agrawal et al. [12] are the first to propose a coverage based fault localization technique called dicing, which subtracts the set of statements executed by a passed test case from those executed by a failed one and reports the result statements as the likely faulty statements. Renieris et al. [4] extended this idea and proposed Nearest Neighborhood Queries technique. The technique compares the spectra of the successful runs and failed runs, then select the nearest passed run as one of input to dicing.

The Tarantula system [9] colored the statements to highlight the particular statements that contain bugs. It used the number of successful tests and failed tests to locate buggy statements. The intuition is that the more failed tests and the less successful tests cover a statement, the more likelihood for the statement to be faulty. Different colors for different suspicious statements are then assigned to visualize program codes. Therefore a buggy statement is colored as red for highlighting so that developers can focus on it directly. Empirical evaluation [13] showed that Tarantula consistently outperforms four other techniquesSet union [4], Set intersection [4], Nearest Neighbor [4], and Cause Transitions [14].

Abreu et al. [10] proposed several spectra metrics to study the accuracy of prediction by fault localization. In their work, they found that Ochiai metric is more effective in bug localization performance than other metrics. Similar to Tarantula, these metrics only use binary information of test execution to rank statements.

Wong et al. [15] proposed some heuristic strategy to assign different weights for passed and failed tests respectively. Weights for passed/failed tests are assigned according to the number of passed/failed tests. The tests are grouped and tests in the different group are assigned different weights. As the total number of successful tests and the total number of failed tests are fixed, weights assigned to tests are fixed regardless of individual execution information. Moreover, their method does not distinguish weights for individual tests in the same group that contribute to statements.

Naish et al. [5] proposed a weighting strategy for failed tests. The rationale behind the idea is that failed tests that cover few statements provide more information than other failed tests. Thus, they assumed that weight of a failed test is inversely proportional to the number of statements exercised in the test. Inspired by their work, we will extends their studies in three aspects: 1) We use basic blocks rather than statements to calculate weight for each failed test since statements within a basic blocks are often not distinguishable from each others in terms of error diagnosis. 2) We additionally consider the imbalance property of tests with respect to each statement. The weight of a failed test is assigned according to proportion between the number of failed tests and that of passed tests. 3)Weights of passed tests are also quantified. In this study, we also compare our weighting strategy with method proposed by Naish et al.

Bandyopadhyay et al. [6] extended the idea of nearest neighbor queries [6] to incorporating the relative importance of different passing test cases in the calculation of suspiciousness scores. They stated that the importance of a passing test case is proportional to its average proximity to the failing test cases. They used different thresholds for their weighting function to control weights assigned to tests. However, in their study, they do not prescribe how to choose best threshold for weighting function. Different from their work, our work focuses on relative importance of failing tests and passing tests.

III. approach

A program spectrum is a collection of data that record the statements that are executed in each test case. To sum up such information, the current popular SFL techniques utilize four spectrum parameters. They are the number of passed/failed test cases in which statement was/wasnt executed. Following Abreu et al. [10], the notation $< a_{ef}(s), a_{nf}(s), a_{ep}(s), a_{np}(s) >$ is used to denote these four numbers for each statement s to calculate suspiciousness score based on statistical formula. The first part of

© 2014 ACADEMY PUBLISHER

the subscript in the notation for each parameter indicates whether the statement is executed (e) or not (n) and the second one indicates whether the test passed (p) or failed (f). In the context of clarity that these spectrum parameters are provided for the specific statement s, we sometimes omit the notation "(s)" in the spectrum parameters for simplicity. Similar to this, in this paper, four weighted spectrum parameters with respect to each statement are defined as follows:

$N_{ef}(s)$: the weights of failed test cases that cover statement s

$N_{nf}(s)$: the weights of failed test cases that do not cover statement s

$N_{ep}(s)$: the weights of successful test cases that cover statement s

$N_{np}(s)$: the weights of successful test cases that do not cover statement s

The contribution for different types of tests to failure is considered separately in terms of error diagnosis. The reason is that typically failed tests present definite information about program behavior that fault is activated and propagated to program failure, while passed tests do not guarantee that fault is activated. The computation of the above four weights for each statement will be described in the following sections. To facilitate the discussion in the rest of the paper, let us suppose that a program *P* consists of *n* executable statements, which are denoted as $P = \{s_1, s_2, s_3, \ldots, s_n\}$, and *m* basic blocks, which is denoted as $P = \{b_1, b_2, b_3, \ldots, b_m\}$. Also consider that the program *P* is tested against a test suite *T*, which comprises of *w* different test cases that are denoted as $T = \{t_1, t_2, t_3, \ldots, t_w\}$. The test suite *T* can be partitioned into two disjoint subsets T_p and T_f according to the passed/failed status of test cases.

A. Weights of Failed Tests

In this section, we describe how to compute $N_{ef}(s)$ and $N_{nf}(s)$ for each statement in detail. Two practical test scenarios that motivate us to distinguish weights for failed tests are discussed respectively, and some equations are defined accordingly to capture characteristic of weights induced by test scenarios. Then these equations are combined properly to measure different weights of failed tests with respect to each statement.

1) Weight of coverage of test with respect to each statement: Suppose two tests both trigger program failure while one of them executes less basic blocks than the other does. When measuring fault locating ability for each test, it seems that the test that exercises less basic blocks gives us more information to find fault than the other one, since less basic blocks narrow down the search space for possible locations of faults. In the extreme case that the test executes only one basic block, it is certain that the executed basic block contains faulty statements. The conventional SFL techniques only use the number of failed tests in which a statement is executed, which lose such information. Note that here basic blocks are used rather than statements to capture this characteristic since statements within the same basic block cannot be distinguished by tests.

Let a test case t_i be one element of T_f and cover statement s. Spectra of basic block information for t_i are collected, which is recorded as a binary vector $< b_{1i}, b_{2i}, b_{3i}, \ldots, b_{mi} >$. If block b_j is covered by test t_i, then the value for b_{ji} in the vector is 1 otherwise 0. After collecting basic block information, we compute weight of test case t_i about its execution that contributes to statement s using equation 1:

$$E_{ci}(s) = m/\sum_{r=1}^{m} b_{ri} \tag{1}$$

The weight is inversely proportional to the number of basic blocks executed in the failed test. The greater the E_{ci} is, the more information the test case provides for locating faults. The weight approaches maximum when the failed test executes only one basic block. In such scenario, developers can immediately find the location of fault with the aid of this failed test. When a failed test executes all the basic blocks in the program, it provides relatively less information to aid in locating faults.

The weight for a statement s that is not executed by a failed test t_i is also quantified in the similar way under the assumption that a large value implies that the statement may not be correlated with program failure. The following equation computes such contribution of tests to the statements.

$$E_{ni}(s) = m/(m - \sum_{r=1}^{m} b_{ri} + 0.01) \tag{2}$$

A small constant (0.01 in this study) is used in the denominator to avoid division by zero when all of the basic blocks are executed in a failing test. The weight quantifies the degree that the failed test subjects the statement to be a non-faulty one.

2) Weight of status of tests for failed tests: In testing process, the number of failed tests is often relatively smaller than that of passed tests. It indicates that failed tests are often rarer than passed tests in test sets. Furthermore, a failed test definitely tells us that there exist some faults in the program while a passed test does not. This implies that contribution of failed tests to fault localization should be distinguished from that of passed tests. The conventional SFL techniques do not make full use of this feature, which decrease the effect of failed tests for error diagnosis and may degrade effectiveness of SFL. In this paper, we propose an information quantity [16] based strategy to reduce imbalance between sizes of passed and failed tests, thus contribution of failed tests to failure is dynamically determined by proportion between failed and passed tests. Specifically, let an event be "The test set contains (h+k) tests, of which the number of failed tests is k". We assume that occurrence of test cases is stochastic and independent from each other. The probability for occurrence of this event equals to k/(h+k). Thus, the information quantity for the event can be represented as: $-log(k/(k+h)) = log((h+k)/k)$. Information quantity

© 2014 ACADEMY PUBLISHER

measures how much information is contained by an event. A smaller probability this event occurs with, a larger information quantity the event has. Motivated by this property, we use the information quantity of the event as weight for every k failing tests. Therefore, the weight of every failed test t_i can be computed by equation 3:

$$I = log((h+k)/k+1) \tag{3}$$

A constant (1 in this study) is added in the equation to ensure that weight of a failing test is always greater than 1. The formula dynamically determines weight for every failed test according to size relation between passed tests and failed tests. A great value indicates that the failed information is rare for fault localization and weight of failed tests are adjusted for better error diagnosis. Such case often happens when many passed test cases and a few failed test cases are contained in the test suite, which is common in test process. In such cases, passed test cases may be already redundant while failed test cases are lacking for locating faults, which implicitly indicate that we should pay relatively more attention to such failed test cases owing to redundancy of passing tests.

Next, we quantify the contribution of failed tests associated with statements. Suppose a statement s is executed by a_{ef} failed tests. The statement gets more suspicious as a_{ef} increases. Our intuition is that the contribution of a failed test to a statement should be increased as the statement is executed by more failed tests. To characterize such contribution, we use the following equation to compute: $-log((|T_f| - a_{ef})/|T_f|) = log(|T_f|/(|T_f| - a_{ef})) = log(a_{ef}/(|T_f| - a_{ef}) + 1)$. Therefore, we compute the weight of every failed test t_i that covers the statements using equation 4 by combining the weight from size proportion and the number of failed tests associated with the statement:

$$RC(s) = I \times log(a_{ef}/(|T_f| - a_{ef} + 0.1) + 1) \tag{4}$$

The constant 0.1 is used to avoid division by 0. Similarly, the weight of the failed test t_i for a statement s that is not executed by t_i is calculated using equation 5:

$$RN(s) = I \times log(a_{nf}/(|T_f| - a_{nf} + 0.1) + 1) \tag{5}$$

3) Combined to measure weights of failed tests: To compute $N_{ef}(s)$ and $N_{nf}(s)$ of failed tests with respect to every statement in the program, the above four equations are combined as follows.

For a statement s that is executed by a failed test t_i, the contribution of this test to the statement is computed as: $E_{ci}(s) * RC(s)$. Thus, to compute weights of failed test set *F* all of which covers the statement s, we sum over weight of each test case in *F* using equation 6.

$$N_{ef}(s) = \sum_{t_i \in F} E_{ci}(s) \times RC(s) \tag{6}$$

Similarly, for a failing test t_i that does not cover a statement s, the weight of the test to the statement is computed as: $E_{ni}(s) \times RN(s)$. Therefore, eq. 7 is used to compute weights of a failed test set *U* all of which do not cover the statement s by summing over weight of each test case in *U*.

$$N_{nf}(s) = \sum_{t_i \in U} E_{ni}(s) \times RN(s) \tag{7}$$

B. Weights of Passed Tests

Passed tests provide clues about whether the executed statements are innocent of program failures or not. However, passed tests may also execute faulty statements and recent studies revealed that fault localization is susceptible to such tests [10]. The extents to which passed tests subject statements not to be faulty ones may vary for various passed tests due to their execution information. For example, if statements executed by a specific passed test share little intersection with statements of other failed tests, we have high confident to conclude that the statements executed by the passed test may be free from program faults and thus give those statements lower examination priority. However, the current SFL does not distinguish contribution of passed tests. To mitigating negative impacts of such tests on fault localization, we proposed a method to quantify weights of passed tests according to how close the passing test is to failed tests.

Suppose a test case t_i is denoted as a binary vector $< e_{i1}, e_{i2}, \ldots, e_{in} >$, where $e_{ij} = 1$ indicates the statement s_j is executed by the test while $e_{ij} = 0$ indicates the statement is not executed. To quantify how similar two tests are, a modified $Ochiai$ similarity coefficient [17] is used:

$$Sim(t_i, t_j) = \frac{v(t_i \cap t_j)}{\sqrt{v(t_i) \times v(t_j)}}, where$$
$$v(t_k) = \sum_{m=0}^{n} I_{km} \; and \; I_{km} = \begin{cases} a_{ef}(m) & if \; e_{km} = 1 \\ 0 & others \end{cases} \tag{8}$$

The function of $v(t_k)$ measures the closeness of t_k to failed tests. We assign various contributions of statements to failure according to execution information. The greater the $sim(t_i, t_j)$ is, the more similar two tests are. If two tests executed the same statements, the similarity between them approximate maximum, that is 1. If two tests shared no same statements in their executions, the similarity between them is 0. Based on this, we assign a lower weight to a passed test that is more similar to the failed test. Thus, given a set of failed tests, the weight for a passed test t_p is computed as average similarity with failed tests using following equation:

$$W(t_p) = \frac{\sum_{t_i \in T_f}(1 - Sim(p, t_i))}{|T_f|} \tag{9}$$

For every statement s and a passed test set *C* all of which covers the statement s, the weights of test set *C* for statement s are recorded in N_{ep}, which is summed over each test case in *C* as equation 10:

$$N_{ep} = \sum_{t \in C} W(t) \tag{10}$$

© 2014 ACADEMY PUBLISHER

TABLE I.
STATISTICS OF SUBJECTS PROGRAMS

Subject	Faulty Versions	LOC	Size of Test Pool
print_tokens	4	565	4130
print_tokens2	10	510	4115
replace	27	563	5542
schedule	4	412	2650
schedule2	8	307	2650
tcas	35	173	1608
tot_info	23	406	1054
flex	27	5217	567
grep	23	12653	809
gzip	17	6573	213

Similarly, for every statement s and a passed test set *N* all of which does not covers the statement s, a variable N_{np} is recorded, which is summed over each test case in *N* as equation 11:

$$N_{np} = \sum_{t \in N} W(t) \tag{11}$$

IV. EXPERIEMENTS

In this section, we conduct two sets of experiments to evaluate the effectiveness of our algorithms. We compare the conventional method with our weighted method and peer works across subject programs.

A. Subject Programs

In this paper, Siemens suite and Unix programs are used as subject programs for the empirical studies, which are obtained from the Software artifact Infrastructure Repository [18]. These programs have been used to measure the effectiveness of fault localization techniques in previous studies [10], [13]. The Siemens suite contains seven small programs while Unix contains three relatively big programs. For each program, there are a variety of test cases and faulty versions available. Table I presents the detailed information on the subject programs. The *Faulty Versions* column lists the number of faulty versions for each subject program.The column *LOC* shows the lines of code for each program. The column *Size of Test Pool* represents the total number of available test cases in the test pool for each program.

Following previous work [10], [13], we excluded those faulty versions whose faults cannot be detected by any test case in the test suites, since failed runs are required for SFL techniques. Besides, we also remove the versions whose faults are introduced in non-executable statements, such as modifications in the header files, mutants in variable declaration statements, or modifications in a macro statement started with "#define". We use the standard coverage tool *gcov* in conjunction with *gcc* to collect coverage information of program executions, and hence we also excluded those versions that *gcov* cannot handle owing to segmentation faults. In summary, we used all the remaining 143 faulty versions in our data analysis.

TABLE II.
REFINED FAULT LOCALIZATION TECHNIQUES

Name	Formula
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$
SBI	$\frac{a_{ef}}{a_{ef}+a_{ep}}$
O	$\begin{cases}-1 & if\ a_{nf} > 0 \\ a_{np} & if\ a_{nf} < 0\end{cases}$
O^p	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
Wong2	$a_{ef} - a_{ep}$
Wong3	$a_{ef} - h, where\ \ h = \begin{cases}a_{ep} & if\ a_{ep} \le 2 \\ 2+0.1(a_{ep}-2) & if\ 2 < a_{ep} \le 10 \\ 2.8+0.001(a_{ep}-10) & if\ a_{ep} > 10\end{cases}$

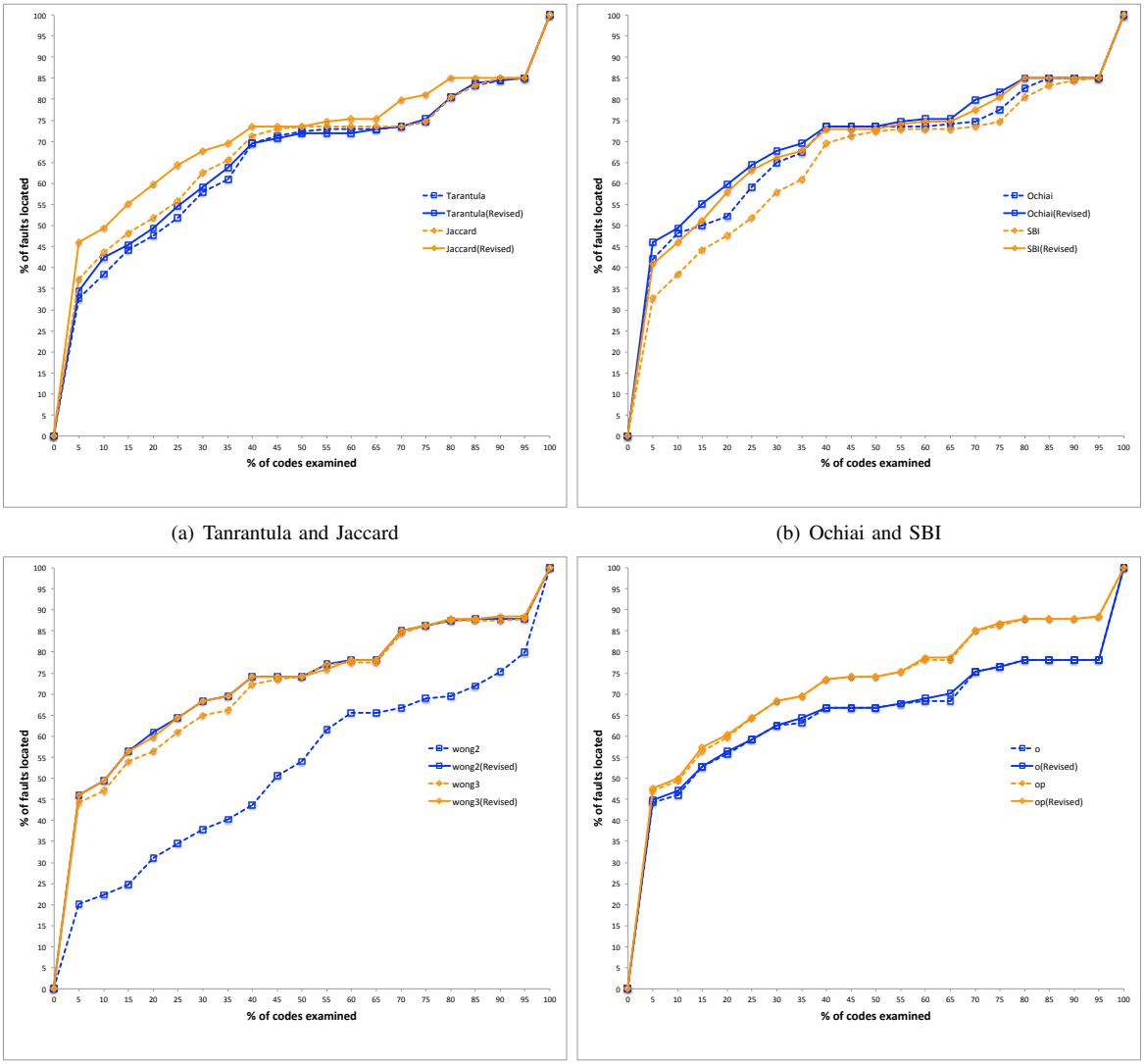
B. Formulas under investigation

After the above four weights are collected for each statement, the faulty statements are supposed to have relatively high N_{ef} values and relatively low N_{ep}. To quantify how each program statement is correlated with program failure, some function that maps the four weights to a single value (we call such number suspiciousness score) for each statement can be applied to rank the statements. The property of employed function must satisfy the condition that those statements with the highest values are most possible to be faulty. In this paper, we apply our method to some SFL techniques as a refinement to calculate suspiciousness score for every statement. Table II shows eight conventional SFL techniques. Of the studied techniques, none utilize integer-specific operations such as modulo. Thus, no adaptation is needed for use with non-integral values; The spectrum parameters $< a_{ef}, a_{nf}, a_{ep}, a_{np} >$ in conventional formulas can be simply substituted by $< N_{ef}, N_{nf}, N_{ep}, N_{np} >$ respectively to generate weighted based techniques. Of the investigated techniques, Tarantula is an old technique and locates faults based on the assumption that the faults are executed by relatively more failed tests and less passed tests [9]. Jaccard is used in the Pinpoint framework and evaluated very effective in previous studies [19]. Ochiai is often used for computing genetic similarity in molecular biology and is first used as fault localization technique in [10]. SBI is a variant of CBI that measure suspicious value in the statement-level [20]. Wong2 and Wong3 are techniques that distinguish the contribution of all the passed test cases to debugging [15]. O and O^p have been proved to be the best in certain cases among many formulas [21]. Because different techniques have different formulas, we would like to investigate whether all of them can benefit from our weighted refinement.

C. Evaluation Metric

To measure the effectiveness of fault localization techniques, we follow [20] to use *Expense* metric. It measures the percentage of the program that must be examined to find the fault following rank list from top down. The lower the measure is, the better the effectiveness is. It is defined

© 2014 ACADEMY PUBLISHER



(a) Tanrantula and Jaccard

(b) Ochiai and SBI

(c) Wong2 and Wong3

(d) O and O^p

Figure 1. effectiveness comparison between conventional SFL and our weighted approaches

as follow:

$$Expense = \frac{|V_{examined}|}{|V|} * 100\%$$

$|V|$ measures the size of executable codes in the program, and $|V_{examined}|$ measures the number of statements that has to be inspected so as to find the fault. In case of tie when two or more statements share same suspiciousness score, we adopt the worst cases. That is, developers have to examine all the tie statements to find the faulty statement. In our study, Expense reduction score $\Delta Expense = Expense - Expense^{'}$ is also used to measure relative effectiveness improvement, where $Expense$ and $Expense^{'}$ refers to Expense before and after applying our approach respectively. A positive value of $\Delta Expense$ indicates that the effectiveness of fault localization is improved after applying our approach. The greater the measure value is, the more improvement the effectiveness of fault localization gains.

D. Results for comparison with conventional techniques

In this section, we present experimental results to investigate effectiveness in fault localization by applying the refinement method to all the formulas listed in Table II. Fig. 1 illustrates effectiveness between conventional SFL techniques and corresponding refined techniques in all faulty versions. For all the listed figures, the x-axis represents the percentage of executable statements to be examined, and the y-axis denotes the percentage of faulty versions whose faults have been located by examining no more than corresponding percentage of executable statements in the x-axis.

As shown in Fig. 1, the curves for eight revised formulas are always higher than the corresponding techniques. That indicates that all the eight revised formulas using our approach achieve better performance than, or at least competitive with the original SFL techniques in terms of effectiveness of fault localization. For example, based on Figure 1. (c), we find that by examining less than 10% of the code, the revised Wong2 formula can locate faults for

© 2014 ACADEMY PUBLISHER

TABLE III.
MEAN EXPENSE REDUCTION SCORE FOR INDIVIDUAL PROGRAM

Subjects	Tarantula (Revised)	Jaccard (Revised)	Ochiai (Revised)	SBI (Revised)	Wong2 (Revised)	Wong3 (Revised)	O (Revised)	O^p (Revised)
print_tokens	3.85%	3.21%	0.00%	5.51%	29.92%	0.00%	0.00%	0.00%
print_tokens2	0.59%	12.42%	7.77%	12.42%	25.63%	0.39%	0.29%	0.29%
replace	0.29%	2.94%	0.70%	3.16%	26.78%	0.11%	-0.05%	-0.05%
schedule	0.67%	4.03%	1.51%	4.53%	9.24%	0.17%	0.17%	0.17%
schedule2	3.57%	11.82%	7.91%	12.22%	23.50%	0.05%	0.00%	0.00%
tcas	0.22%	3.16%	1.54%	2.68%	19.52%	0.66%	0.09%	0.09%
tot_info	1.50%	8.84%	5.88%	10.16%	36.10%	0.46%	0.29%	0.29%
flex	0.33%	0.21%	0.20%	0.11%	1.06%	0.03%	0.00%	0.02%
grep	-0.90%	0.00%	-0.21%	1.57%	11.28%	2.54%	-0.59%	-0.10%
gzip	0.17%	-0.50%	-0.29%	0.39%	9.35%	6.61%	0.00%	-0.17%

TABLE IV.
EXPENSE SCORES COMPARISON WITH PEER WORK FOR EIGHT TECHNIQUES

(a) Tarantula,Jaccard,Ochiai and SBI

Subjects	Tarantula -Revised	Tarantula -Naish	Jaccard -Revised	Jaccard -Naish	Ochiai -Revised	Ochiai -Naish	SBI -Revised	SBI -Naish
print_tokens	**5.00%**	8.08%	**3.34%**	4.75%	**3.34%**	**3.34%**	**3.34%**	8.08%
print_tokens2	**16.77%**	20.33%	**7.66%**	19.12%	**7.66%**	14.35%	**8.11%**	20.33%
replace	11.48%	**11.13%**	**8.49%**	10.48%	**8.49%**	8.94%	**8.61%**	11.13%
schedule	**6.55%**	7.22%	**2.69%**	6.21%	**2.69%**	3.53%	**2.69%**	7.22%
schedule2	57.69%	**56.53%**	**45.85%**	56.33%	**45.85%**	53.23%	**45.85%**	56.53%
tcas	**52.04%**	52.26%	**48.97%**	52.13%	**48.88%**	50.37%	**49.58%**	52.26%
tot_info	**34.43%**	35.78%	**25.55%**	33.86%	**25.55%**	30.65%	**25.77%**	35.78%
flex	**31.47%**	32.53%	**26.62%**	27.26%	**26.62%**	26.98%	**30.50%**	32.53%
grep	**50.11%**	53.24%	**44.05%**	49.11%	**44.05%**	48.61%	**47.64%**	53.23%
gzip	32.43%	**32.10%**	31.82%	**31.03%**	31.81%	**30.82%**	33.20%	**32.10%**

(b) Wong2,Wong3,O and O^p

Subjects	Wong2 -Revised	Wong2 -Naish	Wong3 -Revised	Wong3 -Naish	O -Revised	O -Naish	O^p -Revised	O^p -Naish
print_tokens	**3.34%**	33.26%	**3.34%**	**3.34%**	**3.34%**	**3.34%**	**3.34%**	**3.34%**
print_tokens2	**7.55%**	31.05%	**7.55%**	7.61%	**7.55%**	22.30%	**7.55%**	**7.55%**
replace	**8.15%**	34.91%	**8.15%**	8.26%	8.15%	**8.11%**	8.15%	**8.11%**
schedule	**2.69%**	11.92%	**2.69%**	2.85%	**2.69%**	2.85%	**2.69%**	2.85%
schedule2	**45.85%**	71.49%	**45.85%**	46.24%	**45.85%**	54.61%	**45.85%**	46.15%
tcas	**48.57%**	68.09%	**48.09%**	48.75%	**48.09%**	51.78%	**48.09%**	48.18%
tot_info	**25.48%**	61.58%	**25.48%**	25.94%	**25.48%**	29.94%	**25.48%**	25.77%
flex	**26.54%**	37.43%	**26.54%**	27.96%	**30.86%**	30.94%	**26.55%**	26.66%
grep	**38.41%**	50.62%	**38.92%**	46.03%	56.61%	**56.02%**	**39.59%**	43.20%
gzip	**23.26%**	33.04%	**23.79%**	30.82%	**62.41%**	**62.41%**	23.85%	**23.04%**

46% of the faulty versions. In contrast, the conventional Wong2 formula can only locate 20.1% of the faults.

Furthermore, we can observe that the improvement of effectiveness of fault localization varies for different SFL techniques using our refined method. For example, the performance of the refined $Wong2$ technique can be significantly improved while that of the refined O^p technique is improve slightly. One of the reason why our refinement method has little effect on the improvement for certain techniques may be that the faulty statements have already been ranked by these techniques at the very front position. For these techniques, our method does not introduce much inverse impact for fault localization. That is, these techniques can still performance competitively with original techniques.

For a more detailed comparison, Table III presents the mean expense reduction scores for eight refined techniques on each program. It can be seen from this table for most programs and techniques the reduction score is always greater than 0%. This indicates that the effectiveness of conventional SFL techniques get improved by our weighting method in most cases. Among all the improvement, the highest one is in program tot_info using formula Wong2. The relative improvement is as high as 36.10%. Besides, we can also observe that for some technique on certain programs, the relative improvement decrease. For example, the improvement on the program grep using refined Tarantula formula decreases by 0.90%. However, the decrease is very small and limited compared with increase on effectiveness of fault localization. In total, on average the effectiveness of fault localization for 98.6% faulty versions can be enhanced.

E. Results for comparison with peer work

In this section we compare our method with peer works. Naish et al. [5] proposed a weighting function for failed tests and considered weight of a failed test is inversely proportional to the number of statements exercised in the test. We implemented their weighting method on the eight techniques ourselves and carefully examined the

© 2014 ACADEMY PUBLISHER

weighting function to assure it is strictly consistent with that in [5]. Table IV shows the mean *Expense* scores of these techniques on each program. We refer to refined techniques by our method using "-revised" suffix and refined techniques by Naish et al. using "-Naish" suffix.

From table IV, we can find that for most of programs, the expense scores enhanced by our method are often greater than the corresponding scores enhanced by Naishs method, which means that when locating faults in programs, our method requires less code examination than the method proposed by Naish. Furthermore, our method can achieve much more improvement on fault localization than method proposed by Naish et al. Taking Ochiai in Table IV (a) as an example, the expense score is 7.66% using our method in print_tokens2 while 14.35% using Naishs method, which indicates that our approach can obtain nearly 50% saving in terms of examination on codes compared with Naish et al. method.

V. CONCLUSIONS AND FETURE WORKS

In this study, we proposed a weight-based refinement method for conventional SFL techniques to enhance error diagnosis. Our method distinguishes the weight of one failed test case from anther or one passed test case from another. For each failed test, different weights are assigned with respect to each statement according to its coverage information and proportion between failed tests and passed tests. For each passed test, weights are assigned according to its average similarity to all failed tests. In such a way, our method can be generally applied to existing SFL techniques to improve fault localization effectiveness. We conduct experiments to evaluate the refinement techniques on eight typical SFL formulas using two standard benchmarks. The experimental results suggested that the revised techniques can always achieved better performance than, or at least competitive with the techniques original performance.

In our future work, we plan to conduct more empirical studies by using larger scale programs and multiple-faults versions. We also wish to explore other factors that may impact on weights of test cases to develop more effective strategies for effectively locating faults.

REFERENCES

[1] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459 – 494, 1985.

[2] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, vol. 32, pp. 831–848, 2006.

[3] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux, "Dellis: A data mining process for fault localization," in *SEKE*, 2009, pp. 432–437.

[4] M. Renieres and S. Reiss, "Fault localization with nearest neighbor queries," oct. 2003, pp. 30 – 39.

[5] L. Naish, H. J. Lee, and K. Ramamohanarao, "Spectral debugging with weights and incremental ranking," in *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, dec. 2009, pp. 168 –175.

[6] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov. 2011, pp. 420 –423.

[7] X. Jiang, X. Gu, and Y. Lin, "Adaptive double-threshold joint spectrum sensing based on energy detection," *Journal of Computers*, vol. 8, no. 10, 2013. [Online]. Available: http://ojs.academypublisher.com/index.php/jcp/article/view/jcp081025652569

[8] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *SIGPLAN Not.*, vol. 40, pp. 15–26, June 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065014

[9] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 467–477.

[10] R. Abreu, P. Zoeteweij, and A. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, 2007, pp. 89 –98.

[11] Y. Li, C. Liu, and Y. Zi, "Exploiting weights of test cases to enhance fault localization," in *SEKE*, 2013, pp. 589–593.

[12] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," 1995.

[13] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101949

[14] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 342–351. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062522

[15] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, ser. COMPSAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 449–456.

[16] J. A. T. Thomas M. Cover, *Elements of Information Theory, 2nd Edition*. Hardcover, July 2006.

[17] O. A, "Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions," *Bull. Jpn. Soc. Sci. Fish*, vol. 22, p. Bull. Jpn. Soc. Sci. Fish, 1957.

[18] *http://sir.unl.edu/content/sir.php*.

[19] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 595–604. [Online]. Available: http://dl.acm.org/citation.cfm?id=647883.738238

[20] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 201–210. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368116

[21] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

© 2014 ACADEMY PUBLISHER