Automated Software Engineering manuscript No.
(will be inserted by the editor)

Diversity Maximization Speedup for Localizing Faults in Single-Fault and Multi-Fault Programs

Xin Xia[*†] · **Liang Gong**[†] · **Tien-Duy B. Le** · **David Lo** · **Lingxiao Jiang** · **Hongyu Zhang**

Received: date / Accepted: date

Abstract Fault localization is useful for reducing debugging effort. Such techniques require test cases *with oracles*, which can determine whether a program behaves correctly for every test input. Although most fault localization techniques can localize faults relatively accurately even with a small number of test cases, choosing the right test cases and creating oracles for them are not easy. Test oracle creation is expensive because it can take much manual labeling effort (i.e., effort needed to decide whether the test cases pass or fail). Given a number of test cases to be executed, it is challenging to minimize the number of test cases requiring manual labeling and in the meantime achieve good fault localization accuracy.

[*]This work was done while the author was visiting Singapore Management University.

[†]The two authors contribute equally for this work.

Xin Xia
College of Computer Science and Technology, Zhejiang University, Hangzhou, China
E-mail: xxkidd@zju.edu.cn

Liang Gong
Electrical Engineering and Computer Sciences, University of California, Berkeley, USA
E-mail: gongliang13@eecs.berkeley.edu

Tien-Duy B. Le
School of Information Systems, Singapore Management University, Singapore
E-mail: btdle.2012@phdis.smu.edu.sg

David Lo
School of Information Systems, Singapore Management University, Singapore
E-mail: davidlo@smu.edu.sg

Lingxiao Jiang
School of Information Systems, Singapore Management University, Singapore
E-mail: lxjiang@smu.edu.sg

Hongyu Zhang
School of Software, Tsinghua University, Beijing, China
E-mail: hongyu@tsinghua.edu.cn

To address this challenge, this paper presents a novel test case selection strategy based on *Diversity Maximization Speedup* (Dms). Dms orders a set of unlabeled test cases in a way that maximizes the effectiveness of a fault localization technique. Developers are only expected to label a much smaller number of test cases along this ordering to achieve good fault localization results. We evaluate the performance of Dms on 2 different types of programs, single-fault and multi-fault programs. Our experiments with 411 faults from the Software-artifact Infrastructure Repository show (1) that Dms can help existing fault localization techniques to achieve comparable accuracy with on average 67% and 6% fewer labeled test cases than previously best test case prioritization techniques for single-fault and multi-fault programs, and (2) that given a labeling budget (i.e., a fixed number of labeled test cases), Dms can help existing fault localization techniques reduce their debugging cost (in terms of the amount of code needed to be inspected to locate faults). We conduct hypothesis test and show that the saving of the debugging cost we achieve for the real *C* programs are statistically significant.

Keywords Fault Localization · Test Case Prioritization

1 Introduction

Software testing and debugging activities are often labor-intensive, accounting for 30% to 90% of labor spent for a project (Beizer, 1990). Establishing sufficient testing and debugging infrastructure can help reduce software errors that cost the US economy 59.5 billion dollars (0.6% of 2002's GDP) (National Institute of Standards and Technology (NIST), 2002). Many automated testing and debugging techniques have been proposed to reduce the high cost in such activities.

Spectrum-based fault localization (e.g. Jones et al, 2002; Abreu et al, 2009; Campos et al, 2013) is an automated debugging technique that can narrow down the possible locations of software faults and help save developers' debugging time. Many spectrum-based fault localization techniques take as input a set of executions with labels (i.e., indicating whether an execution passes or fails), compare between failed and passed executions, and statistically locate faulty program entities. Such techniques require each execution to be labeled as a failure or a success, which often needs human interpretation of an execution result and may not be easy to determine when a failure is not as obvious as a program crash or invalid output formats. Labeling all executions or test cases as passing or failing for a program may require much manual effort and is often tedious, and thus, the effectiveness of existing spectrum-based fault localization techniques may be potentially hampered due to the unavailability of labeled test cases.

With test case *generation* techniques (Godefroid et al, 2005; Sen et al, 2005), we may be less concerned with lacking test cases. However, we still face the same problem of lacking test *oracles* that can determine whether a program behaves correctly for an input. Note that many software failures do

not have obvious symptoms, such as crashes or violation of predefined specifications; they may simply produce a wrong number or display a widget in an inappropriate place, and they still need a human to decide whether the results are good or not, which could be a laborious and error-prone activity. Recently, Artzi et al (2010) propose a directed test case generation approach for fault localization. They however only handle two kinds of errors in web applications that automated test oracles can be constructed: program crashes and invalid HTML documents. Campos et al (2013) use probability concepts to generate new test cases that could minimize the entropy of fault localization results. Although their approach reduces the diagnosis costs of fault localization results, it does not directly aim to minimize the number of test cases generated that may require manual labelling. In general programs, constructing automated test oracles is much more complicated and still requires much manual effort.

The key research question for this paper is as follows:

How can we minimize the number of test cases requiring human labeling while achieving comparable fault localization effectiveness as when all test cases are labeled, for both single-fault and multi-fault programs?

In this paper, we propose the concept of *diversity maximization speedup* (Dms) and an associated test case prioritization strategy to minimize the human effort needed to label test cases while maintaining the effectiveness of existing spectrum-based fault localization techniques. The concept is based on our observation that when given a sufficient number of suitable test cases, an effective fault localization technique would assign a unique suspiciousness score to most program elements (e.g., a function, a statement, a branch, or a predicate), and high scores to faulty elements and low scores to non-faulty ones. We thus design Dms to *speedup the changing process of the suspiciousness scores* generated by a fault localization technique *by using as few test cases as possible*.

This concept can be applied to both single-fault and multi-fault programs to reduce human effort required for labelling test cases. On the other hand, the amount of reduction achieved by the concept can be different for single-fault and multi-fault scenarios. We present detailed evaluation and comparison in Section 5. When we describe the intuition of this concept and the algorithmic details for realizing the concept, we do not explicitly distinguish these two scenarios from each other.

1.1 Running Example

Figure 1(a) and 1(b) illustrate how our concept helps reduce the number of test cases while maintaining the effectiveness of fault localization techniques.

There are 11 statements $s_1, ..., s_{11}$ in the program in Figure 1(a) (adapted from previous papers (González-Sanchez et al, 2011b; Jiang et al, 2011)), where s_7 is faulty. Suppose the program has 12 test cases $t_1, ..., t_{12}$. A dot for a statement under a test case means the corresponding statement is executed (or hit) in the corresponding test case. The collection of such dots (or represented

Statement		Test case												Suspiciousness Metrics						
		t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	N_{ef}	N_{ep}	N_{nf}	N_{np}	*Ochiai*	*Tarantula*	*Jaccard*
`main(){`	s1																			
`int let, dig, c;`	s2	•	•	•	•	•	•	•	•	•	•	•	•	3	9	0	0	0.500	0.500	0.250
`let = dig = 0;`	s3																			
`while(c=getchar()){`	s4																			
`if('A'<=c && 'Z'>=c)`	s5	•	•	•	•	•	•	•		•	•	•	•	3	8	0	1	0.522	0.529	0.273
`let += 1;`	s6		•	•	•	•	•				•	•	•	2	6	1	3	0.408	0.500	0.222
`else if('a'<=c && 'z'>c) /*FAULT*/`	s7	•	•		•	•		•		•		•		3	4	0	5	0.655	0.692	0.429
`let += 1;`	s8	•	•		•	•		•						2	3	1	6	0.516	0.667	0.333
`else if('0'<=c && '9'>=c)`	s9	•	•			•		•		•		•		2	4	1	5	0.471	0.600	0.286
`dig += 1;`	s10	•	•			•				•		•		2	3	1	6	0.516	0.667	0.333
`printf("%d %d\n",let,dig);}`	s11	•	•	•	•	•	•	•	•	•	•	•	•	3	9	0	0	0.500	0.500	0.250
pass/fail		P	F	P	F	P	P	P	P	F	P	P	P							

(a) Fault Localization with All Test Cases

Group(s) of the same score (the groups are ordered according to their suspiciousness)	Selected Test Case	Program Spectra												Normalized *Ochiai* Scores									
		s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	p/f	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11
{s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11}	t2	1	1	1	1	1	1	1	1	1	1	1	F	0.0909	0.0909	0.0909	0.0909	0.0909	0.0909	**0.0909**	0.0909	0.0909	0.0909	0.0909
{s5,s6,s7,s8,s9,s10},{s1,s2,s3,s4,s11}	t8	1	1	1	1	0	0	0	0	0	0	1	P	0.0742	0.0742	0.0742	0.0742	0.1049	0.1049	**0.1049**	0.1049	0.1049	0.1049	0.0742
{s7,s8,s9,s10},{s5,s6},{s1,s2,s3,s4,s11}	t6	1	1	1	1	1	1	0	0	0	0	1	P	0.0696	0.0696	0.0696	0.0696	0.0852	0.0852	**0.1205**	0.1205	0.1205	0.1205	0.0696
{s7,s8},{s5,s6},{s1,s2,s3,s4,s9,s10,s11}	t4	1	1	1	1	1	1	1	1	0	0	1	F	0.0824	0.0824	0.0824	0.0824	0.0951	0.0951	**0.1165**	0.1165	0.0824	0.0824	0.0824
{s7},{s5},{s8,s9,s10},{s1,s2,s3,s4,s11},{s6}	t9	1	1	1	1	1	0	1	0	1	1	1	F	0.0875	0.0875	0.0875	0.0875	0.0978	0.0753	**0.1129**	0.0922	0.0922	0.0922	0.0875

(b) Evolution of Suspiciousness Scores with Test Cases Selected by our approach

Fig. 1 Running Example

as sequences of `1` and `0` as shown in Figure 1(b)) are called *program spectra*. With the spectra for all of the test cases and their pass/fail information, fault localization techniques may calculate various suspiciousness scores for each of the statements and rank them differently. In this case, three well-known techniques, *Ochiai* (Abreu et al, 2009), *Tarantula* (Jones and Harrold, 2005), and *Jaccard* (Abreu et al, 2009) all rank s_7 as the most suspicious statement (the last three columns in the highlighted row for s_7 in Figure 1(a)). However, the fault localization techniques can in fact achieve the same effectiveness (i.e., ranking s_7 as the top suspicious one) with much fewer test cases when our concept is applied.

Use *Ochiai* as an example. First, we select an initial small number of test cases (t_2 in the example). After a programmer labels the execution result of t_2, *Ochiai* can already assign a suspiciousness score to each statement, although the ranks are not accurate (as in the last 11 columns of the row for t_2 in Figure 1(b)). Then, our approach calculates the potential rank changes that may be caused if a new test case is used by *Ochiai*, and selects the next test case with the maximal change potential (t_8 in our case) for manual labeling. With a label for t_8, *Ochiai* updates the suspiciousness scores for the statements (as in the last 11 columns of the row for t_8). Repeating such a process three more times, test cases t_6, t_4 and t_9 are added, and *Ochiai* can already rank s_7 as the most suspicious statement. Thus, our approach helps *Ochiai* to effectively locate the fault in this case with only five test cases, instead of 12. Section 3 and 4 present more details about our approach.

1.2 Contributions

We have evaluated our approach on five real *C* programs and seven Siemens test programs from the Software-artifact Infrastructure Repository (SIR (Do et al, 2005)). In total, we analyze 411 faults. 254 of them are in single-fault versions from these 12 programs, while the other 157 faults are in 173 versions of 8 of these programs. The results demonstrate that our approach outperforms existing test case selection methods for fault localization.

1. Given a target fault localization accuracy, our approach can significantly reduce the number of test cases needed to achieve it. In particular, we compare with several state-of-the-art test case prioritization strategies, including coverage-based (e.g., Stmt-Total (Rothermel et al, 2001; Elbaum et al, 2002), Art (Jiang et al, 2009)), fault-exposing potential based (Rothermel et al, 2001), and *diagnostic prioritization* (González-Sanchez et al, 2011a,b; Jiang et al, 2011), and our approach achieves, on average, test case reduction rates from 10% to 96% for single-fault programs, and 6% to 67% for multi-fault programs.
2. Given a maximum number of test cases that a programmer can manually label (i.e., given a fixed number of test cases to be used for fault localization or a testing budget), Dms can improve the accuracy of fault localization and thus helps reduce the amount of code programmers need to investigate to locate faults and reduce testing and debugging cost. In comparison with other test case selection techniques, we show, with Wilcoxon signed-rank test (Wilcoxon, 1943) at 95% confidence level, that the cost saving achieved by Dms is statistically significant on real-life programs.

1.3 Paper Outline

The rest of this paper is organized as follows: Section 2 describes fault localization and test case prioritization techniques that we use in our study. Section 3 formally introduces the problem we address. Section 4 presents our approach in detail. Section 5 presents our empirical evaluation. Section 6 describes more related works. Finally, Section 7 concludes with future work.

2 Preliminaries

In this section, we summarize relevant materials on software fault localization and test case prioritization that we use in our empirical evaluation.

2.1 Fault Localization

Spectrum-based fault localization aims to locate faults by analyzing program spectra of passed and failed executions. A program spectra often consists of information about whether a program element (e.g., a function, a statement, or

a predicate) is hit in an execution. Program spectra between passed and failed executions are used to compute the suspiciousness score for every element. All elements are then sorted in descending order according to their suspiciousness for developers to investigate. Empirical studies (e.g. Nainar et al, 2007; Jones and Harrold, 2005) show that such techniques can be effective in guiding developers to locate faults. Parnin and Orso (2011) conduct a user study and show that by using a fault localization tool, developers can complete a task significantly faster than without the tool on simpler code. However, fault localization may be much less useful for inexperienced developers.

The key for a spectrum-based fault localization technique is the formula used to calculate suspiciousness. Table 1 lists the formulae of three well-known techniques: *Tarantula* (Jones and Harrold, 2005), *Ochiai* (Abreu et al, 2009), and *Jaccard* (Abreu et al, 2009). Given a program element s, $N_{ef}(s)$ is the number of ***f**ailed* executions that ***e**xecute* s; $N_{np}(s)$ numerates ***p**assed* executions that do ***n**ot* hit s; by the same token, $N_{nf}(s)$ counts ***f**ailed* executions that do ***n**ot* hit s and $N_{ep}(s)$ counts ***p**assed* executions that ***e**xecute* s.

Table 1 Spectrum-based fault localization

Name	Formula
Tarantula	$\frac{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)}}{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)}+\frac{N_{ep}(s)}{N_{ep}(s)+N_{np}(s)}}$
Ochiai	$\frac{N_{ef}(s)}{\sqrt{(N_{ef}(s)+N_{nf}(s))\cdot(N_{ef}(s)+N_{ep}(s))}}$
Jaccard	$\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)+N_{ep}(s)}$

Example. Each column for t_i in Figure 1(a) is a spectrum. The columns N_{ef}, N_{ep}, N_{nf}, and N_{np} can thus be calculated from the spectra. The suspiciousness scores of *Tarantula*, *Ochiai*, and *Jaccard* for each statement are then calculated based on the formulae in Table 1.

2.2 Test Case Prioritization

Rothermel et al (2001) define the problem of test case prioritization as follows:

Definition 1 (Test Case Prioritization) Given (1) T, a set of test cases, (2) PT, the set of permutations of T, and (3) f, a function mapping PT to real numbers, the problem is to find a permutation $p \in PT$ such that: $\forall p' \in PT.f(p) \geq f(p')$.

In this definition, PT represents the set of all possible orderings of T; f is an award function indicating the value for each ordering. The higher the value,

the better it is. For easier implementation, award functions in the literature are often defined as a priority function mapping test cases to real numbers, and then the optimal permutation is simply to sort the test cases in descending order according to their values. The key for a test case prioritization technique to be effective is to design a priority function that assigns appropriate priority to the test cases under given situations. The following subsections highlight some test case prioritization techniques that we compare with our approach.

2.2.1 Coverage Based Prioritization

Stmt-Total (Rothermel et al, 2001) is a test case prioritization strategy that assigns higher priorities to a test case that executes more statements in a program. **Stmt-Addtl** (Rothermel et al, 2001) extends Stmt-Total by selecting next test case that covers more statements that have not been covered by previously selected test cases. *Adaptive Random Test Prioritization* (**Art**) (Jiang et al, 2009) starts by randomly selecting a set of test cases that achieves maximal coverage, and then sort the unlabeled test cases based on their *Jaccard distances* to previous selected test cases. Among its several variants, **Art-Min** was shown to be the best test case prioritization strategy (Jiang et al, 2009). However, recent study (Arcuri and Briand, 2011) shows that Art may not be effective when the failure rate is low and the high distance calculations cost might overshadow the reduction on test execution times.

2.2.2 Fault-Exposing Potential Based Prioritization

Fep-Addtl (Rothermel et al, 2001) aims to sort test cases so that the *rate of failure detection* of the prioritized test cases can be maximized. To reduce the need for test oracles, the rate of failure detection is approximated by the *fault-exposing potential* (Fep) of a test case, which is in turn estimated based on program *mutation analysis* (Hamlet, 1977): each program element s_j is mutated many times and the test case t_i is executed on each mutant; the Fep of t_i for s_j (Fep_{ij}) is calculated as the ratio of mutants of s_j detected by t_i over the total number of mutants of s_j; then, the Fep of t_i (Fep_i) is the sum of the FEP of t_i for all elements ($\sum_j \text{Fep}_{ij}$).

2.2.3 Diagnostic Prioritization

Jiang et al (2011) investigate the effects of previous test case prioritization techniques on fault localization and find that coverage-based techniques may be insufficient since the prioritized test cases may not be useful in supporting effective fault localization. González-Sanchez et al (2011b) use the concept of *diagnostic distribution* that represents the probability of a program element to be faulty, which is then estimated by *Bayesian inference* based on previous selected test cases, and in their tool named **Sequoia**, sort test cases so that the information entropy of the diagnostic distribution can be minimized. Soon after, González-Sanchez et al (2011a) propose another strategy called

Ambiguity Group Reduction to sort test cases. In their tool named **Raptor**, program elements having the same spectrum record are considered to be in the same ambiguity group (Ag), and Raptor aims to select next test case that would maximize the number of ambiguity groups while trying to minimize the deviation on the sizes of the ambiguity groups.

2.2.4 Practical Usage

To use the above mentioned test case prioritization techniques in practice, a program needs to be instrumented first and executed with test cases to collect the program spectra (execution traces) of the test cases. Then, developers can apply one of the test case prioritization techniques to select top-n ranked test cases, and manually judge whether each of the test cases passes or fails. Based on these selected test cases and their corresponding labels (passed or failed), a fault localization technique (e.g. Nainar et al, 2007; Jones and Harrold, 2005) can then be applied to locate faults. Our test case prioritization technique in this paper can also be applied in the same fashion as the above mentioned test case prioritization techniques.

3 Problem Definition

In this section we show a motivating example and formally introduce our approach: ***D**iversity **M**aximization **S**peedup* (Dms). Dms employs trend analysis to give priorities to test cases that can quickly increase the diversity of suspiciousness scores generated by fault localization techniques for various program elements. In the subsections, we illustrate its intuition and formally define it as a variant of test case prioritization.

3.1 Running Example Revisited

We use the running example (Figure 1(a)) to explain the intuitions for Dms. With sufficient test cases, an effective fault localization technique is more likely to assign high suspiciousness scores to faulty program elements while assigning low scores to non-faulty elements, and each element should be assigned a unique rank according to their suspiciousness scores to facilitate further investigation (such as the scores shown in the last three columns in Figure 1(a)).

With fewer test cases, a fault localization technique may not be able to achieve an effective ranking. Figure 2 shows the evolution trend of the ranks of the running example's program statements based on their *Ochiai* (Abreu et al, 2009) scores as test cases are added one by one. The test cases are added by Raptor which is the existing best approach in the literature (González-Sanchez et al, 2011a) for selecting test cases for fault localization. In this figure, the horizontal axis represents the number of iterations to select test cases. In each iteration, one test case is picked from the unlabeled test case pool $T_{\mathcal{U}}$.

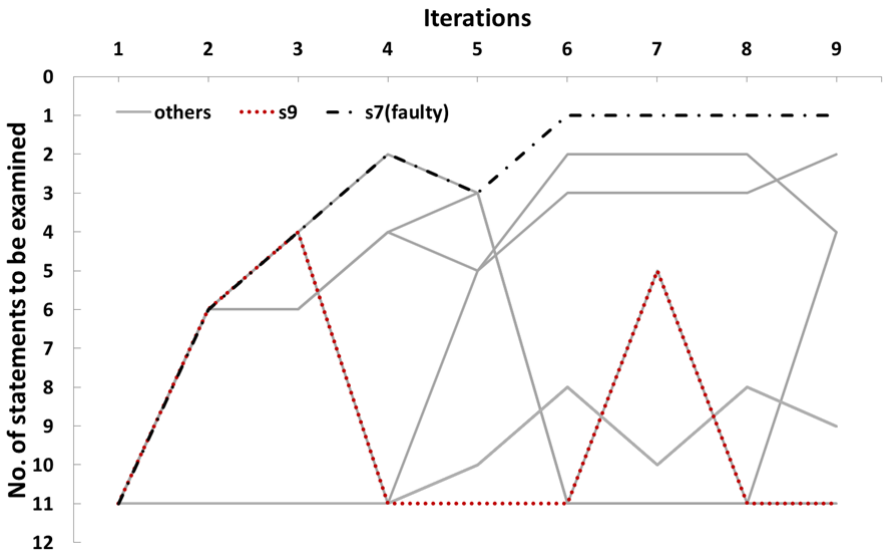

Fig. 2 Motivating Example

The vertical axis is the rank of a statement sorted based on suspiciousness.[1] Each line in the figure depicts the evolution of the suspiciousness rank for one specific statement. For example, s_7 (the faulty statement) is ranked 11^{th} in the first iteration, and 6^{th} in the second iteration.

This figure shows that the ranks of different statements may evolve in different ways as more test cases are added. Specifically, some statements keep rising in general (e.g., s_7); some others oscillate back and forth (e.g., s_9). Ideally, we should only use test cases that could enable a fault localization technique to assign elements the scores close to the final score when all test cases are used. Comparing to the changes of s_7, the oscillation of s_9 is less important as its final rank is the same as its initial rank. Thus, when we add test cases, we should look for test cases that could offer more *changing opportunities* to "promising" elements like s_7 (with clear trend) instead of s_9 so that the ranks (for both s_7 and s_9) may quickly approach their final position.

The following questions prompted us to define Dms:

1. Can we analyze the change trend of every program element and identify "promising" elements with *high change potentials* (i.e., elements whose ranks are likely to change much in a stable way)?
2. For program elements having high change potentials, can we select appropriate test cases to speed up their rank-changing process so that these elements can reach their final ranks faster (i.e., with fewer test cases)?

[1] Program elements with the same suspiciousness score are assigned the same *low* rank since developers are expected to investigate all of the elements having the same score if they are ever going to investigate one. For example, if statements s_1, s_2, s_3 have the highest suspiciousness score, then the ranks of the 3 statements are all 3.

3.2 Formal Definition of DMS

Definition 2 (Diversity Maximization Speedup) Given (1) T, a set of test cases, (2) PT, the set of permutations of T, and (3) k, a positive integer, we use p^k to represent a permutation $p \in PT$ truncated at length k, and PT^k to represent all such truncated permutations (i.e., $PT^k = \{p^k | p \in PT\}$).

Then, with f, a function mapping PT^k to real numbers, the problem of DMS is to find a permutation $p \in PT$ such that: $\forall p_i^k \in PT^k.\ f(p^k) \geq f(p_i^k)$, for the given k.

In Definition 2, f is an award function indicating the value of an ordering in PT^k, which in our case, would be the effectiveness of a fault localization technique based on k labeled test cases. The number k can be used as a labeling budget, indicating the number of test cases developers are willing to label for fault localization. Thus, the goal for Dms is to quickly maximize the effectiveness of fault localization techniques with at most k labeled test cases.

4 Approach Details

In this section we answer the two questions raised in Section 3.1 to conceptualize Dms.

4.1 Identify High Change-potential Elements

In order to evaluate the change potential of program elements, we first represent program element's rank changes as time series data points. We then fit the points to a linear model using regression analysis. The regression coefficient of the model and the error (i.e., discrepancy between the model and the real points) are used as proxy to identify program elements with high change potentials. More details are described as follows.

Representative Time Series Construction. We capture changes in the ranks of a program element as a series of *trend units*:

1. When the rank of the program element decreases, its current trend unit is `[+]`.
2. When the rank of the program element increases, its current trend unit is `[-]`.
3. If the element's rank stays the same, its current trend unit is `[0]`.

For example, the ranks of statement s_8 in different iterations and its corresponding trend units are listed in Table 2. This series of trend units is further converted to a series of points $< x_i, y_i >$, where x_i represents the iteration number, and y_i represents cumulated changes in program ranks at iteration i. We set y_0 as 0. When the trend in iteration i is `[+]`, $y_i = y_{i-1} + 1$. If the i-th trend is `[-]`, $y_i = y_{i-1} - 1$, otherwise, if the trend does not change(`[0]`)

Table 2 Evolution Trend of s_8.

Iteration (x_i)	1	2	3	4	5	6	7	...
Rank	11	6	4	2	3	11	5	...
Trend ($\mathcal{T}$)		[+]	[+]	[+]	[-]	[-]	[+]	...
y_i	0	1	2	3	2	1	2	...

then $y_i = y_{i-1}$. We refer to this series of points as the *evolution trend* of the corresponding program element.

Linear Model Construction. Then we use *linear regression analysis* (Graybill and Iyer, 1994) to model the trend of each program element. Each trend is modeled as a linear equation:

$$y_i = \beta_1 \cdot x_i + \beta_0 + \epsilon_i \tag{1}$$

Change Potential Computation. In order to speed up the overall evolution process, our approach needs to select next test case that keeps elements with monotonic trends (high change-potential trends) evolving their rankings. In other words, we do not care about changing elements' ranking with unstable trends. In order to identify those high change-potential elements, we need a metric to evaluate and compare trends of different elements. Here we define the change potential of a program element d with the trend $\mathcal{T}_d$ as follows:

$$\mathcal{W}_{\mathcal{T}_d} = \frac{|\hat{\beta}_1|}{\hat{\sigma}_{\beta_1} + 1} \tag{2}$$

$\hat{\beta}_1$ is estimated by *least squares* and $\hat{\sigma}_{\beta_1}$ is the error of estimating β_1 (Graybill and Iyer, 1994). In this metric, the numerator is the absolute value of the trend slope and the denominator considers the fitness of the regression model which represents the deviation of the actual value from the regression model. When the context is clear, we also use $\mathcal{W}_{\mathcal{T}}$ or $\mathcal{W}_d$ to represent the change potential of a trend or a program element.

Rationale of Equation 2: We want to evolve the ranks of statements in a fast, monotonic way. In linear models, *a fast changing monotonic trend should have a larger slope β_1 as well as a smaller deviation ϵ_i from the linear model.* Using this metric in Equation 2 that uses the estimated slope in the numerator and the estimated deviation in the denominator, we may isolate trends that evolve in faster and more stable (less oscillation) ways. Table 3 shows a few sample trends and their change potentials according to Equation 2.

4.2 Speed up the Rank Change Process

After evaluating the program elements according to their change potentials, DMS will try to speed up the evolution trend of the program elements based

Table 3 Trend examples and their potentials

$\mathcal{T}$		$\hat{\beta}_1$	$\hat{\sigma}_{\beta_1}$	$\mathcal{W}_{\mathcal{T}}$
[+]	[+]	1	0	1
[+]	[-]	0	0.577	0
[+]	[0]	0.5	0.289	0.388
[0]	[0]	0	0	0

on the change potential ($\mathcal{W}_{\mathcal{T}}$). First, program elements with the same suspiciousness scores are grouped together, they are termed as *suspicious groups* in this paper.[2] These suspicious groups are then assigned change-potential scores based on the change potentials of their constituent program elements. When new test cases are added, based on the actual program elements that get executed, some groups can be broken into two. When this happens, the diversity of the suspiciousness scores increases in most cases. The goal of DMS is to select a new test case that breaks a group into two sub-groups where the overall change potentials are maximized.

We calculate the potential of a group g by summing up the potential of all program elements d that belongs to g.

$$\mathcal{W}_g = \sum_{d \in g} \mathcal{W}_{\mathcal{T}_d} \tag{3}$$

where $\mathcal{W}_{\mathcal{T}_d}$ is the change potential of the program element d based on the labelled execution trace profiles.

Rationale of Equation 3: *A group with high change-potential elements should be given a higher priority to break.* We want to diversify the rankings of elements in the suspicious group that has a high change-potential score. To identify those high change-potential groups, we measure the sum of change-potential scores of its member elements as in Equation 3.

The overall change-potential score of all suspicious groups (G) is calculated as follows:

$$\mathcal{H}_G = \sum_{g_i \in G} \mathcal{W}_{g_i}^2 \tag{4}$$

To evaluate an unlabeled trace t, DMS calculates the difference between the overall change-potential score of the current groups G ($\mathcal{H}_G$) and the overall change-potential score of all groups when t is added to the pool of labeled test cases ($G \Leftarrow t$). Since we want to stabilize the ranks of all program elements as quickly as possible with as fewer test cases as possible, we heuristically

[2] We call such groups as *suspicious groups* since we simply want to state the fact that every group may contain potentially suspicious elements. Some other studies (e.g. González-Sanchez et al, 2011a) call them *ambiguity groups* as that term may emphasize more on the fact that the elements in the groups have the same but ambiguous suspiciousness scores.

choose the test case that can maximize the difference and thus the reduction of change potentials as the next one for labeling.

$$\arg\max_{t \in T_{\mathcal{U}}} \left\{ \mathcal{H}_G - \mathcal{H}_{(G \Leftarrow t)} \right\} \tag{5}$$

The new groups $(G \Leftarrow t)$ and their change potential $\mathcal{H}_{(G \Leftarrow t)}$ can be estimated based on t's spectrum (i.e., the set of program elements hit by t) even when the pass/fail label for t is unknown. Given an existing suspicious group, if a newly added test case t only covers a subset of the group elements, this group may be broken into two: one contains the elements hit by t, and the other contains the elements uncovered by t. Then, each subgroup inherits a portion of the original group's change potential proportional to its size. For example, suppose a group g in $\mathcal{H}_G$ contains 2 elements, whose potentials are 0.4 and 0.6 respectively, and a new test case t breaks g into g_1 and g_2, each of which contains 1 element; then, the change potentials $\mathcal{W}_{g_1}$ and $\mathcal{W}_{g_2}$ are both $\frac{1}{2} \times (0.4 + 0.6) = 0.5$.

Rationale of Equation 4 and 5: *A test case that breaks more groups with higher change potentials should be given a higher priority.* Equation 4 measures the overall change potential score of all suspicious groups and its square form manifests the diversity of element ranks. As an example, suppose there are two groups g_1 and g_2. Group g_1 has two high change-potential elements with change-potential score 0.3 and 0.4. Group g_2 has two low change-potential elements with change-potential score 0.1 and 0.2. According to Equation 4, $\mathcal{H}_G = (0.3+0.4)^2 + (0.1+0.2)^2 = 0.58$. After choosing a test case that breaks g_1 only and does not change the change-potential score of any element, then according to Equation 5, the new change potential would be $0.3^2+0.4^2+(0.1+0.2)^2 = 0.34$. However, if we choose another test case that breaks g_2 only and does not change the change-potential score of any element, the new change potential would be $(0.3+0.4)^2 + 0.1^2 + 0.2^2 = 0.54$. As a result, the test case that breaks the high change-potential group (*i.e.,* g_1) leads to a larger decrease of the overall change potential and thus will be given a higher priority to be selected.

Note that Dms does not intentionally increase suspiciousness scores of promising statements that could lead to *confirmation bias*. Dms might make an initially promising statement become less suspicious if the statement is covered in the next selected trace and the trace is labeled as *pass*, or it is not covered in the next selected trace and the trace is labeled as *fail*.

4.3 Overall Approach

Before prioritization, all test cases will be executed on instrumented program versions and the corresponding traces would be collected. Our approach (pseudocode in Algorithm 1) takes in a set of unlabeled traces $T_{\mathcal{U}}$ and the labelling budget k (i.e., the maximum number of traces to be manually labeled), and outputs k selected traces for manual analysis. One failed trace (t_0 in Line 1)

Algorithm 1 Diversity Maximization Speedup

Procedure DiversityMaximizationSpeedup
Input:
k - Maximum number of traces to be selected
w - Switching threshold
$T_{\mathcal{U}}$ - Unlabeled trace set, where $|T_{\mathcal{U}}| > k$
t_0 - Initial failed trace
Output:
k selected test cases prioritized
Method:

1: $T_{tmp} \leftarrow \{< t_0, fail >\}$
2: *//Bootstraping with another prioritization technique* $\mathcal{P}$
3: **while** $|T_{tmp}| \leq k$ **and** $|T_{tmp}| \leq w$ **do**
4: Select t by $\mathcal{P}$
5: $c \leftarrow$`manual_label`(t)
6: $T_{tmp} \leftarrow T_{tmp} \cup \{ < t, c > \}$; $T_{\mathcal{U}} \leftarrow T_{\mathcal{U}} \setminus \{t\}$
7: $\forall d \in \mathcal{D}$, calculate suspicious score $f_{T_{tmp}}(d)$
8: $\forall d \in \mathcal{D}$, update trend $\mathcal{T}_d$ based on $f_{T_{tmp}}(d)$
9: **end while**
10: $T_{\mathcal{S}} \leftarrow T_{tmp}$
11: *//Speedup*
12: **while** $|T_{\mathcal{S}}| \leq k$ **do**
13: $\forall d \in \mathcal{D}$, calculate $\mathcal{W}_{\mathcal{T}_d}$ by Equation 2
14: Select t by Equation 5
15: $c \leftarrow$`manual_label`(t)
16: $T_{tmp} \leftarrow T_{tmp} \cup \{ < t, c > \}$; $T_{\mathcal{U}} \leftarrow T_{\mathcal{U}} \setminus \{t\}$
17: $\forall d \in \mathcal{D}$, calculate suspicious score $f_{T_{tmp}}(d)$
18: $\forall d \in \mathcal{D}$, update $\mathcal{T}_d$ based on $f_{T_{tmp}}(d)$
19: $T_{\mathcal{S}} \leftarrow T_{\mathcal{S}} \cup T_{tmp}$
20: **if** `div`(T_{tmp}) cease growing **then**
21: $T_{tmp} \leftarrow \{< t_0, fail >\}$
22: $\forall d \in \mathcal{D}$, clear $\mathcal{T}_d$
23: **end if**
24: **end while**
25: return $T_{\mathcal{S}}$

is also used as an input because a developer usually starts debugging only when at least one test fails,[3] and fault localization techniques rarely produce meaningful results if all spectra consists of only passed executions.

To collect indicative trends for analyzing and speedup, at Lines 3-9 we first collect w traces by one generic prioritization technique $\mathcal{P}$ and record evolution trend $\mathcal{T}_d$ for each program element d. This step is desirable since it helps bootstrap the trend analysis in our solution. At Lines 12-24, we perform the second stage which speeds up the change process based on existing trends. Note that after selecting each test case t in this stage, we will update the trend for all elements. f_T represents a fault localization technique (e.g., *Ochiai*), built based on the set of test cases T. $f_T(d)$ returns the suspicious score for the program element d.

[3] If there is more than one test that fails, DMS randomly selects one of them to begin with.

In the pseudocode, `manual_label`(t) asks a user to check the correctness of the outcome from the test case t. Procedure `div`(T) counts the number of unique suspicious scores (diversity) generated by f_T, which is defined as follows:

$$\texttt{div}(T) = \left| \bigcup_{d \in \mathcal{D}} \{f_T(d)\} \right| \tag{6}$$

The diversity of small programs may reach the maximum after selecting a small number of test cases. To avoid random selection after that happens, the pseudo-code at Lines 20-23 resets the set T_{tmp} based on which the suspiciousness scores of all program elements are calculated. With this step, DMS can continually choose test cases from $T_{\mathcal{U}}$ that maximally diversify suspicious scores calculated based on T_{tmp}. Repeating the diversity selection process may help to confirm the previously selected test cases and make the final result more robust.

4.4 Example

We describe step by step how DMS minimizes the number of test cases needed by *Ochiai* to locate the fault in the running example in Figure 1(a) and Figure 1(b).

Since the example code snippet is quite small, there is no need to use a large number of initial test cases to bootstrap our trend analysis. We set $w = 1$ and thus only use one test case (in addition to t_0) for bootstrapping. In this example and our evaluation in Section 5, we use RAPTOR, one of the previously best techniques, in the bootstrapping process for better comparison.

Initially, users execute the program and expose a failure (t_2 in this example) in which all statements are covered. Thus all statements get equal non-zero suspiciousness and constitute one suspicious group g (cf. either Figure 1(b) and Table 4). All non-zero suspicious groups compose a group set $G = \{g\}$. RAPTOR would then choose t_8 since t_8 has the maximum ambiguity reduction values, and present it to developers for labeling as either *pass* or *fail*).

After the bootstrapping stage, *Ochiai* updates the suspiciousness score for each statement based on the selected traces and the existing suspicious group set are broken into $\{s_1,s_2,s_3,s_4,s_{11}\}$ and $\{s_5,s_6,s_7,s_8,s_9,s_{10}\}$ (cf. either Figure 1(b) and Table 4), they are called g_1 and g_2 respectively. At this time, the

Table 4 Evolution of Suspiciousness Scores for the Running Example in Figure 1(a) using RAPTOR (González-Sanchez et al, 2011a).

Suspicious Group(s) (the groups are ordered according to their suspiciousness)	Selected Test Case	Program Spectra												Normalized *Ochiai* Scores									
		s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11	p/f	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10	s11
{s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11}	t2	1	1	1	1	1	1	1	1	1	1	1	F	0.0909	0.0909	0.0909	0.0909	0.0909	0.0909	**0.0909**	0.0909	0.0909	0.0909	0.0909
{s5,s6,s7,s8,s9,s10},{s1,s2,s3,s4,s11}	t8	1	1	1	1	0	0	0	0	0	0	1	P	0.0742	0.0742	0.0742	0.0742	0.1049	0.1049	**0.1049**	0.1049	0.1049	0.1049	0.0742
{s7,s8,s9,s10},{s6,s5},{s1,s2,s3,s4,s11}	t6	1	1	1	1	1	1	0	0	0	0	1	P	0.0696	0.0696	0.0696	0.0696	0.0852	0.0852	**0.1205**	0.1205	0.1205	0.1205	0.0696
{s7,s8},{s5,s6}{s1,s2,s3,s4,s11},{s9,s10}	t4	1	1	1	1	1	1	1	1	1	0	0	1	F	0.0824	0.0824	0.0824	0.0824	0.0951	0.0951	**0.1165**	0.1165	0.0824	0.0824	0.0824
{s7,s8},{s6},{s5},{s10},{s1,s2,s3,s4,s11},{s9}	t7	1	1	1	1	1	0	1	1	1	0	1	P	0.0840	0.0840	0.0840	0.0840	0.0940	0.1085	**0.1085**	0.1085	0.0664	0.0940	0.0840
{s7},{s10},{s5},{s1,s2,s3,s4,s11},{s6},{s8},{s9}	t9	1	1	1	1	1	0	1	0	1	1	1	F	0.0885	0.0885	0.0885	0.0885	0.0969	0.0834	**0.1084**	0.0834	0.0834	0.1022	0.0885

trend for the statements in g_1 is [+], because the ranks of these statements change from 11 to 6, while the trend for the statements in g_2 is [0], because their ranks are still 11. The corresponding time series of the statements in g_2 are: $y_0 = 0$ and $y_1 = 1$. Applying Equation 2, we obtain the change potential of the trend of the program elements in g_2 as 1.

We now calculate $\mathcal{H}_G$ for the current suspicious group set $G = \{g_1, g_2\}$ according to Equation 3:

$$\mathcal{H}_G = \mathcal{W}_{g_1}^2 + \mathcal{W}_{g_2}^2 = (\sum_{d \in g_1} 0)^2 + (\sum_{d \in g_2} 1)^2 = 36$$

.

Now there are 10 candidate traces: $\{t_i | 1 \leq \ i \ \leq \ 12 \wedge i \notin \{2, 8\}\}$ to be evaluated. We will use each candidate trace t_i to break ties in G ($G \Leftarrow t_i$). Then we calculate the score that evaluates the breaking effect: $\mathcal{H}_{(G \Leftarrow t_i)}$.

For example, when evaluating t_6, t_6 covers s_1,s_2,s_3,s_4,s_5,s_6 and s_{11}, thus breaks suspicious g_2 into $\{s_5,s_6\}$ and $\{s_7,s_8,s_9,\ s_{10}\}$, let us call them g_{21} and g_{22} respectively. Now, the score $\mathcal{W}_{g_{21}} = \frac{2}{6} \times \mathcal{W}_g = 2$, $\mathcal{W}_{g_{22}} = \frac{4}{6} \times 6 = 4$. So if choosing t_6, the score for ($G \Leftarrow t_6$) is

$$\mathcal{H}_{(G \Leftarrow t_6)} = \mathcal{W}_{g_{21}}^2 + \mathcal{W}_{g_{22}}^2 = 20$$

And the reduction is

$$\mathcal{H}_G - \mathcal{H}_{(G \Leftarrow t_6)} = 36 - 20 = 16$$

.

In the same way, we evaluate all candidate traces and find that the reduction of t_6 is maximal, so we select t_6 as the next trace and ask a developer to manually label t_6. The developer then labels it as *"pass"*. After adding newly labeled trace t_6 into the selected trace set $T_{\mathcal{S}}$, we recalculate the suspicious score of all program elements according to the current selected trace set. After calculation, the normalized suspicious score of the elements in $\{s_5,s_6\}$ reduced from 0.1049 to 0.0852 and their ranks remains the same. The suspicious scores of the elements in $\{s_7,s_8,s_9,s_{10}\}$ increase from 0.1049 to 0.1205 and thus their ranks rises from 6 to 4. After that, the trends of program elements are updated. For example, the trend of elements in $\{s_1,s_2,s_3,s_4,s_{13}\}$ becomes ([0][0]), the trend of the statements in $\{s_5,s_6\}$ becomes ([+][0]) and those in $\{s_7,s_8,s_9,s_{10}\}$ corresponds to ([+][+]).

Note that right now $\{s_7,s_8,s_9,s_{10}\}$ gets the highest change-potential score and thus can get more chances to be broken up. As shown in Figure 1(b), after three iterations, DMS selects ($t_8 \rightarrow t_6 \rightarrow t_4$). In the next iteration, DMS chooses t_9 and breaks $\{s_7,s_8\}$ and $\{s_5,s_6\}$ which have greater change potentials and consequently ranks s_7 the highest. Overall, DMS only requires user to manually label four additional traces ($t_8 \rightarrow t_6 \rightarrow t_4 \rightarrow t_9$).

As a comparison, RAPTOR always chooses the test case that maximally reduces the overall sizes of groups of statements that have the spectrum records (i.e., Ambiguity Group Reduction, c.f. Section 2.2.3). As shown in Table 4,

Raptor effectively selects the same test cases as Dms in the first four iterations; however, it chooses t_7 in the next iteration to break $\{s_1,s_2,s_3,s_4,s_9,s_{10},s_{11}\}$ and $\{s_5,s_6\}$, and it takes one more iteration to rank s_7 the highest. It thus requires users to label five additional test cases besides t_2 ($t_8 \rightarrow t_6 \rightarrow t_4 \rightarrow t_7 \rightarrow t_9$).

4.5 Approach Complexity and Robustness

The time and space complexity of our approach depend on many various factors. It takes linear time with respect to the number of iterations or test cases we want to select, takes quadratic time with respect to the total number of available test cases, takes cubic time with respect to the number of program elements, takes cubic time with respect to the number of suspicious groups in each iteration. Our approach would also store all of the test spectra in the memory for convenience. In comparison with the other existing approach Raptor, the trend analysis step in our approach may be more computationally expensive. However, the step would still take short absolute amount of time since the number of selected test cases can be limited to tens or hundreds to achieve effective fault localization.

The results of our approach may not be deterministic either since there are random factors (e.g., the first failed test case to choose) and several user-defined parameters (e.g., the maximal number of test cases to select) employed in the approach. Thus, we repeat running our approach for multiple times in our evaluation to take average performance. The maximal number of test cases to select (i.e., k in Algorithm 1) and the switching threshold w in the bootstrap phase would affect how much improvements our approach can achieve over other existing techniques, but they do not matter for the main focus of our evaluation, which is to evaluate whether Diversity Maximization Speedup (Dms) is effective for reducing manual labelling effort and diagnostic cost. Although we expect different optimal k and w for different kinds of programs and faults, we use a consistent setting $k <= 500$ and $w = 10$ in our evaluation and would suggest potential users to starts with a small setting where they can tolerate the manual labelling cost.

5 Empirical Evaluation

In this section we present empirical evaluation that analyzes the impact of Dms on manual effort needed for test case labeling, and compares our approach with multiple previous test case prioritization methods.

In particular, our seek for answers to the following two research questions:

RQ1 What is the effectiveness of Dms on single-fault programs?
RQ2 What is the effectiveness of Dms on multi-fault programs?

Section 5.1 shows our experimental results that answer the first research question. Section 5.2 presents answers to the second research question. Section 5.3 describe the discussion and some threats to validity.

5.1 RQ1: Single-Fault Programs

Section 5.1.1 gives details about our experimental setup for single-fault programs. In Section 5.1.2, we introduce the subject programs used in our study. Sections 5.1.3 & 5.1.4 show the results.

5.1.1 Experimental Setups and Measures

In our experiment, every test case prioritization technique starts from an arbitrary labeled failed trace because developers start debugging only when test cases fail.

In this paper, we use RAPTOR as the bootstrapping technique ($\mathcal{P}$ in Figure 1). During the bootstrapping process, w is set to 10 to facilitate trend analysis.

Following Jiang et al (2011), for each faulty version, we repeat each prioritization technique 20 times to obtain its average cost. For each time, a randomly chosen failed trace is used as the starting point to alleviate the sensitivity of the technique to the choice of starting traces. On the other hand, to fairly compare our approach with other prioritization methods, the *same randomly* chosen failed traces are used as the starting traces for all methods.

The effectiveness of test case prioritization methods would be manifested as the effectiveness of the subsequent fault localization results. So one way to compare the effectiveness of different prioritization methods based on the different diagnostic costs of the subsequently applied fault localization technique when the same number of test cases are selected by the different prioritization methods. In the literature, many fault localization studies use the percentage of program elements that need to be inspected according to the ranked list of fault localization results to locate all faults as one kind of diagnostic costs, which is defined as follows:

$$cost = \frac{\mid \{j \mid f_{T_S}(d_j) \geq f_{T_S}(d_*)\} \mid}{|\mathcal{D}|} \tag{7}$$

where $\mathcal{D}$ consists of all program elements appearing in the program and d_* represents the fault(s) (i.e., the root cause(s) of failures) of a program. We calculate the average cost as the percentage of elements that developers have to examine until locating the root causes (d_*) of failures. The lower the cost is, the better a fault localization technique is. Since multiple program elements can be assigned with the same suspicious score, the numerator is considered as the number of program elements d_j that have bigger or the same suspicious score to d_* in this paper.[4]

[4] There are a number of alternative ways to define diagnostic costs and accuracies, such as using the number of faults located when up to a certain percentage of program elements are inspected (e.g. Wong et al, 2014; Debroy and Wong, 2013; Cleve and Zeller, 2005; Baah et al, 2010; Jones et al, 2002; Lucia et al, 2014), or assuming a random ordering for elements with the same score and incorporating their expected rank $\frac{\left|\{j \mid f_{T_S}(d_j)=f_{T_S}(d_*)\}\right|+1}{2}$ in calculating cost (e.g. Ali et al, 2009; Steimann et al, 2013; Steimann and Frenkel, 2012; Xu et al, 2011). We use the one in Equation 7 and 8 as they are commonly used and

We can also define the *accuracy* of a fault localization technique as the reverse of the cost, which is the higher the better:

$$accuracy = 1 - cost \tag{8}$$

In the following parts of the paper, we thus use *cost* and *accuracy* interchangeably when the context is clear.

Another way to measure the effectiveness of test case prioritization methods is to see how many test cases can be reduced by each method. A major goal of our paper is to minimize the number of test cases that need manual labelling but can maintain fault localization accuracies. So, in the following evaluation results, we also show the numbers of reduced test cases *with respect to a targeted fault localization cost (or accuracy).*

If labeling all test cases and performing fault localization on all program spectra results in an average diagnostic cost c, we call it the *base line cost*. If a test prioritization technique or fault localization technique leads to a diagnostic cost c', then we say the technique achieves *x% of base line effectiveness*, where x is defined as follows:

$$x = \frac{c'}{c} \times 100 \tag{9}$$

To be fair, the number of reduced test cases by each prioritization technique should be measured when the technique achieves 100% of base line effectiveness. However, in reality, it is hardly possible to directly control the cost to be exactly 100% of base line. So, we allow 1% deviation; i.e., in the following evaluation results, we measure the numbers of reduced test cases *when at most 101% of base line effectiveness is achieved.*

5.1.2 Subject Programs

We use five real *C* programs and seven Siemens test programs from the *Software-artifact Infrastructure Repository* (SIR) (Do et al, 2005). We refer to the five real programs (`sed`, `flex`, `grep`, `gzip`, and `space`) as UNIX programs. Table 5 shows the descriptive statistics of each subject, including the number of faults, available test cases and code sizes. Following many previous studies (e.g. Jones et al, 2002; Abreu et al, 2009), we exclude faults not directly observable by the `gcov` profiling tool[5] (e.g., some faults lead to a crash before `gcov` dumps profiling information and some faults do not cause any test case to fail), and in total we study 254 faults.

easy to understand, and our focus is on evaluating whether different test case prioritization techniques *change* the diagnostic costs, instead of measuring the absolute costs. We believe that if our technique shows significant improvements over one kind of diagnostic cost, it should also show improvements over other kinds of diagnostic cost.

[5] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Table 5 Subject Programs

Program	Description	LOC	Tests	Faults
tcas	Aircraft Control	173	1609	41
schedule2	Priority Scheduler	374	2710	8
schedule	Priority Scheduler	412	2651	8
replace	Pattern Matcher	564	5543	31
tot_info	Info Measure	565	1052	22
print_tokens2	Lexical Analyzer	570	4055	10
print_tokens	Lexical Analyzer	726	4070	7
space	ADL Compiler	9564	1343	30
flex	Lexical Parser	10124	567	43
sed	Text Processor	9289	371	22
grep	Text Processor	9089	809	17
gzip	Data Compressor	5159	217	15

Table 6 Labeling Effort on Subject Programs

Subject Programs	Dms	Raptor	Sequoia	Stmt-Addtl	Stmt-Total	Fep-Addtl	Art-Min
Siemens	**18**	20	500+	500+	500+	97	150
Unix	**16**	48	176	150	500+	98	56

5.1.3 Experimental Results: Reducing Number of Test Cases

Here, we investigate the effectiveness of Dms in reducing the number of test cases needed for a targeted diagnostic cost. We compare Dms with previous test case prioritization techniques in terms of labeling effort when to achieve 101% of base line effectiveness as stated in Section 5.1.1.

Since Dms would output a ranked list of suspicious program elements, we compute the diagnostic cost c_n for Dms when we just inspect top n ($n \in \{1, 2, \cdots, |D|\}$) suspicious elements. We record the *maximum* n that c_n is still within 101% of base line cost as the amount of labeling effort. Also, we limit the maximum number of test cases allowed to select (i.e., k in Algorithm 1) to 500 in this specific evaluation.

Table 6 shows how many labels are needed on average to achieve 101% of base line effectiveness for each approach. E.g., Raptor requires 48 labels on average for each faulty version from the 5 Unix programs while Dms only needs 16. Overall, Dms requires the minimal amount of labeling effort by achieving 67.7% labeling reduction on Unix programs and 10% reduction on Siemens programs in comparison with the existing best approach (Raptor).

5.1.4 Experimental Results: Reducing Cost

Here, we investigate the effectiveness of DMS in reducing cost given a targeted number of labeled test cases. We select 30 test cases (i.e., set $k = 30$ in Algorithm 1), which we believe are not too many to manually label. We also find that in our experiments the average debugging cost of using DMS will not reduce noticeably even if more labeled test cases beyond 50 are added further (See Figure 3), which is in line with studies in the literature (e.g. Abreu et al, 2009; Liblit et al, 2005) that tens of passed and failed spectra may suffice for fault localization. During the bootstrapping process, the first 10 test cases are picked by RAPTOR. We use different prioritization techniques and apply *Ochiai* to evaluate program elements on the selected program spectra. A prioritization technique that obtains a lower cost is better.

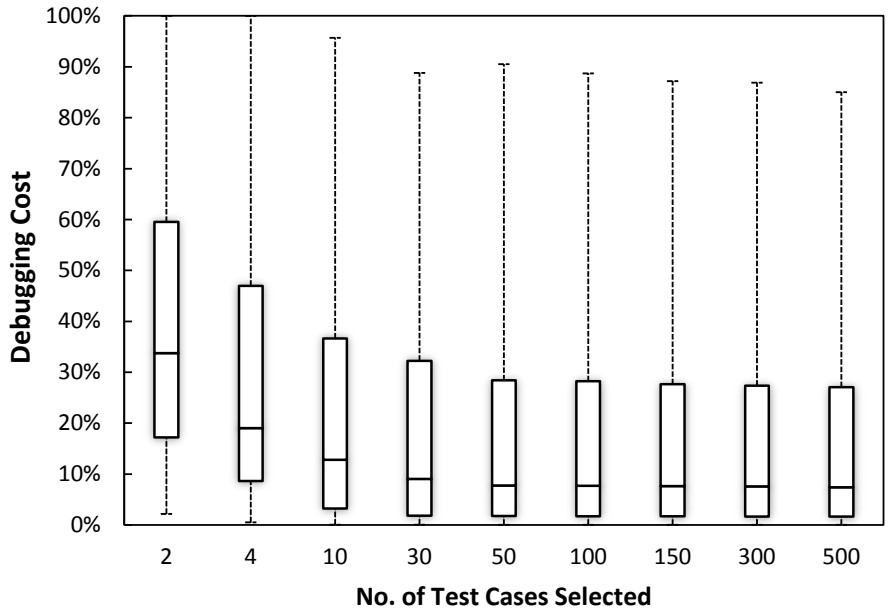

Fig. 3 Average Cost of DMS when Selecting Different Numbers of Test Cases.

Following Baah et al (2010, 2011) and the *cost* metric (Equation 7), we compare the effectiveness of two prioritization methods P_A and P_B by using one of the methods (for example, P_B) as reference measure. When selecting the same number of traces k, the cost difference: $cost(P_B) - cost(P_A)$ is considered as the improvement of P_A over P_B. A positive value means that P_A performs better than P_B (since lower cost is better) and a negative value means that the performance deteriorates if we use P_A to replace P_B. The difference corresponds to the magnitude of improvement. For example, if the cost of test cases from P_A is 30% and the cost of P_B is 40%, then the improvement of

Table 7 Comparison of Prioritization methods.

Test Prioritization Method	Positive	Negative	Neutral
Dms vs Raptor	**25.20%**	19.29%	55.51%
Dms vs Sequoia	**33.46%**	19.69%	46.85%
Dms vs Stmt-Addtl	**42.13%**	19.29%	38.58%
Dms vs Stmt-Total	**62.99%**	7.87%	29.13%
Dms vs Fep-Addtl	**40.16%**	20.08%	39.76%
Dms vs Art-Min	**31.50%**	19.29%	49.21%

Table 8 Distribution of positive improvements.

Test Pri. Tech.	Max	Mean	Median	Min
Dms vs Raptor	**77.42%**	7.71%	3.93%	0.03%
Dms vs Sequoia	**66.67%**	14.38%	8.06%	0.23%
Dms vs Stmt-Addtl	**72.87%**	14.68%	5.17%	0.03%
Dms vs Stmt-Total	**94.97%**	27.68%	22.29%	0.03%
Dms vs Fep-Addtl	**45.90%**	13.83%	6.35%	0.03%
Dms vs Art-Min	**53.81%**	7.70%	3.23%	0.03%

P_A over P_B is 10%, which means that developers would examine 10% fewer statements if P_A is deployed.

Result Summary. Table 7, 8, and 9 compare our method with the existing prioritizing techniques. The results show that our method outperforms no worse than other methods for the majority of faulty program versions.

As illustrated in Table 7, Dms performs better than Raptor on 25.20% of the faulty versions, worse on 19.29% of the faulty versions, and shows no improvement on 55.51% of the faulty versions. The first row of Table 8 characterizes the degree of positive improvement of Dms over Raptor. As the table indicates, half of the 25.20% faulty versions with positive improvement values have improvements between 0.03% and 3.93%, and the other half have improvements between 3.93% and 77.42%. The average positive improvement of Dms over Raptor is 7.71%.

Table 9 characterizes the degree of negative deterioration of Dms over other techniques. As the first row in the table indicates, for half of the 19.29% faulty versions, Dms deteriorates between 0.03% and 0.60% from Raptor, and for the other half, Dms deteriorates between 0.60% and 1.15%. The average percentage of negative deterioration of Dms over Raptor is 0.54%.

We conduct paired Wilcoxon signed-rank test to confirm the difference in performance between Dms and six existing prioritization techniques. The statistical test result rejects the null hypothesis and suggests that the improve-

Table 9 Distribution of negative deterioration.

Test Pri. Tech.	Max	Mean	Median	Min
Dms vs Raptor	**1.15%**	0.54%	0.60%	0.03%
Dms vs Sequoia	**31.71%**	4.01%	1.33%	0.03%
Dms vs Stmt-Addtl	**30.73%**	4.14%	1.52%	0.03%
Dms vs Stmt-Total	**27.88%**	4.61%	2.64%	0.17%
Dms vs Fep-Addtl	**24.70%**	5.06%	2.15%	0.03%
Dms vs Art-Min	**22.41%**	4.11%	1.72%	0.03%

ments of Dms over other existing techniques are statistically significant at 95% confidence interval.

Detailed Comparison. Table 6 shows that Raptor, Fep-Addtl and Art-Min achieve 101% of base line effectiveness with less than 500 test cases on subject programs. Figure 4, 5, and 6 show the comparison of fault localization costs between Dms and the three different prioritization techniques. The horizontal axes represent the number of versions that show differences in the Cost of fault localization. The vertical axes represent the percentage difference in Costs. If Dms is better than the reference method, the area above zero-level line will be larger.

Dms vs Fep-Addtl. Previous studies (Rothermel et al, 2001; Elbaum et al, 2002) show that Fep-Addtl is the most promising prioritizing method for fault detection. Without test oracles, Fep can be estimated by 1−*False Negative Rate* (Fnr) (González-Sanchez et al, 2011b) [6] which is also used in our study. Figure 4 presents the comparison between Dms and Fep-Addtl over all faulty versions that show performance differences. Fep-Addtl is used as the reference prioritization technique. The baseline represents the fault localization cost on program spectra prioritized by Fep-Addtl. Each program version is a bar in this graph and we remove versions from the graph that have no cost differences. In the figure, the vertical axis represents the magnitude of improvement of Dms over Fep-Addtl. If the bar of a faulty version is above the horizontal axis, that means on this version Dms performs better than Fep-Addtl (positive improvement) and the bars below the horizontal axis represent faulty versions for which Dms performs worse than Fep-Addtl.

The comparison shows that Dms performs better than Fep-Addtl on 102 versions, out of 153 versions that show differences in cost, but performs worse than Fep-Addtl on 51 versions. The positive improvement ranges from 0.03% to 45.90%, with an average of 6.35%.

[6] Fnr is the program passing rate when program element is the real fault and executed in test case. Usually when Fnr is high, the fault is difficult to be detected by Spectrum-based fault localization techniques.

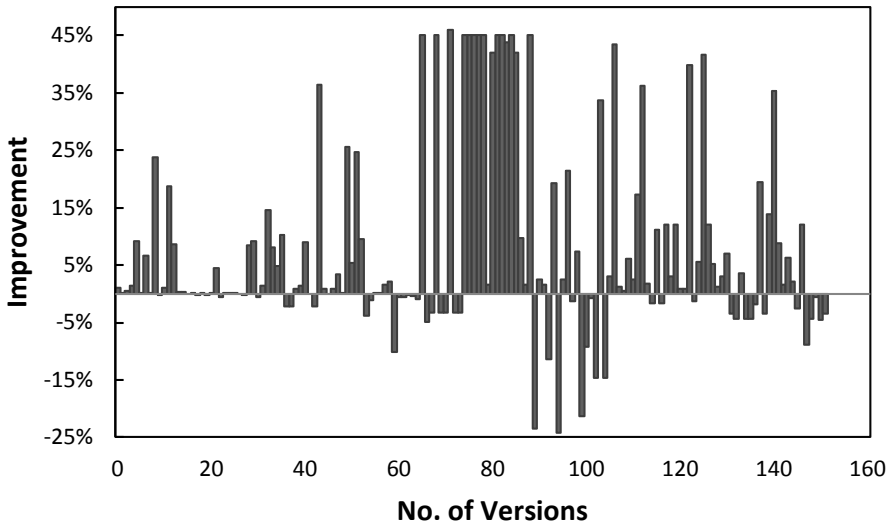

Fig. 4 Improvement of DMS over FEP-ADDTL.

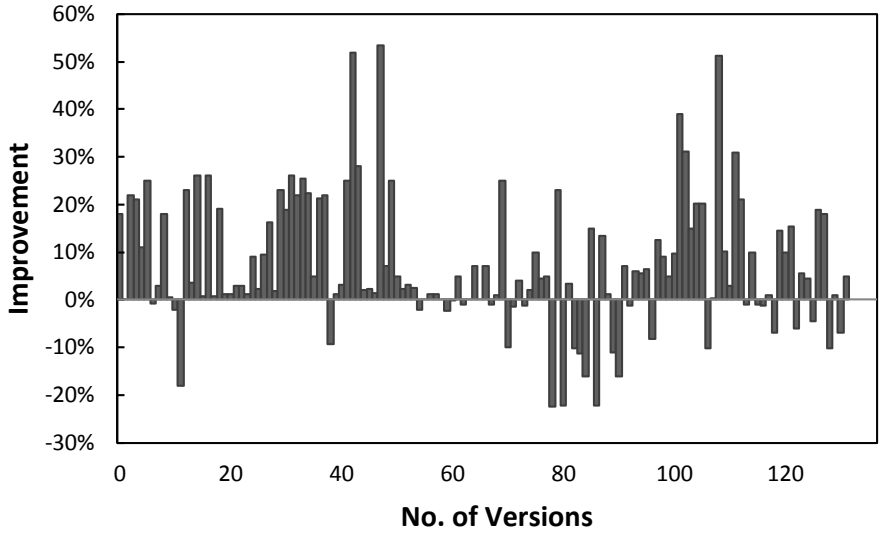

Fig. 5 Improvement of DMS over Art-MIN.

DMS vs Art-MIN. In this study we compare the effectiveness of DMS to *Adaptive Random Test Prioritization*(ART) (Jiang et al, 2009). There are various strategies for ART, in this experiment we only compare with the best one: ART-MIN (Jiang et al, 2009; González-Sanchez et al, 2011b,a). Figure 5 shows the results of the study in which ART-MIN is used as the baseline method. The comparison shows that DMS is better than ART-MIN. Out of 129 versions that show differences in cost, our prioritization method performs better

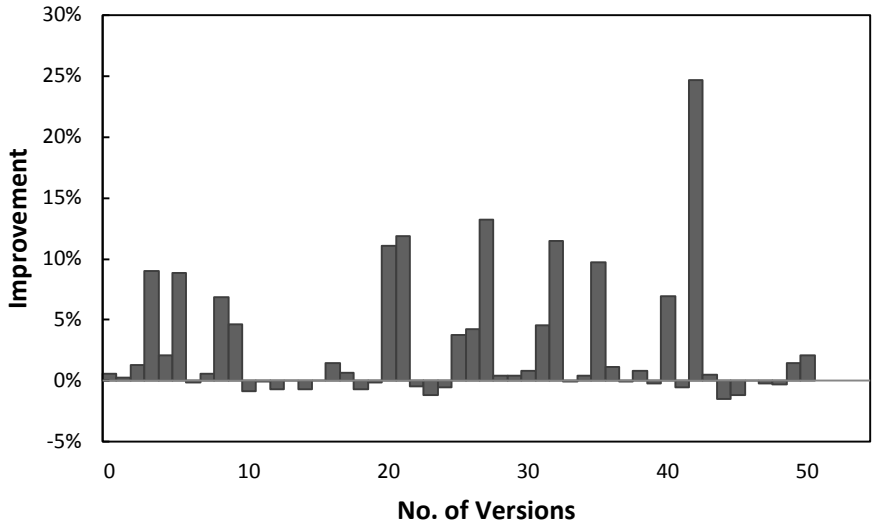

Fig. 6 Improvement of DMS over RAPTOR on UNIX programs.

than ART-MIN on 80 versions but performs worse than the ART-MIN on 49 versions.

DMS vs RAPTOR. Figure 6 shows the comparison between DMS and RAPTOR on UNIX programs. Here we use RAPTOR as the reference metric. The comparison shows that DMS outperforms RAPTOR on 20 versions by at least 1% cost, and on only 5 versions, it is worse than RAPTOR by over 1% cost.

There is also improvement on Siemens programs: 32.2% versions show differences and the average debugging cost improvement is 1.3%, which is not so significant as compared with UNIX programs. This is probably due to the small software size. On Siemens programs, RAPTOR can reach 101% of base line effectiveness by only selecting 20 test cases on average (see Table 6). By selecting such few test cases, RAPTOR already obtains the maximal ambiguity group reduction due to very limited different coverage profiles. For example, all test cases of `tcas` only have less than 15 ambiguity groups in all faulty versions. In this case, the speedup by our method is trivial. In real scenario, programs to be diagnosed would be more similar to UNIX programs.

5.2 RQ2: multi-fault Programs

Section 5.2.1 gives details about our experimental setup for multi-fault programs. Section 5.2.2 introduces the subject programs used in our study. Sections 5.2.3 & 5.2.4 show the results.

5.2.1 Experimental Setups and Measures

The overall experimental setups and measures used for comparison for the multi-fault setting is similar to the single-fault setting.

There is only a minor difference in the definition of the diagnostic cost as now there are multiple faults. The diagnostic cost is defined as follows:

$$cost = \frac{|\{j \mid f_{T_S}(d_j) \geq \min_{d_* \in D*} f_{T_S}(d_*)\}|}{|\mathcal{D}|} \quad (10)$$

where $\mathcal{D}$ consists of all program elements appearing in the program and D_* is a set of faults in a program. We calculate the average cost as the percentage of elements that developers have to examine until locating all root causes (D_*). Since multiple program elements can be assigned with the same suspiciousness score, the numerator is considered as the number of program elements d_j that have bigger or the same suspiciousness score to a root cause d_* in D_* with the lowest suspiciousness score. In this setting, we consider the worst-case scenario where developers need to find all root causes by inspecting all elements that have a score no lower than the score of *any* root cause.

5.2.2 Subject Programs

Each multi-fault program version used in our study contains more than one fault where each fault involves only one line (or one simple statement if the statement is broken into more than one line) in the program and different faults affect different lines. This consideration is aligned with previous studies (e.g. Zhang et al, 2013; Abreu et al, 2009). We use a dataset containing 173 multi-fault versions of 8 C programs as shown in Table 10. Different versions may contain the same fault, and there are 157 distinct faults in total. The dataset was previously used by Lucia et al (2014) to evaluate 40 different association measures.

Table 10 multi-fault Subject Programs

Program	# Bugs Per Version	# Tests	# Versions
tcas	5	1,608	41
schedule2	2	2,710	10
schedule	5	2,650	9
replace	5	5,542	32
tot_info	5	1,052	23
print_tokens2	5	4,115	10
print_tokens	2	4,130	10
space	5	1,343	38

5.2.3 Experimental Results: Reducing Number of Test Cases

We investigate the effectiveness of DMS in reducing the number of test cases needed for a targeted diagnostic cost for our multi-fault subject programs. Table 11 shows how many labels are needed on average to achieve 101% of base line effectiveness (cf. Section 5.1.1) for each approach. For example, Raptor requires 98 labels on average for each faulty version from all of the eight program datasets while Dms needs 79. In total, Dms requires the least amount of labeling effort; in comparison with the existing best approach (FepAddtl), Dms achieves 5.95% labeling reduction on all of the datasets.

Table 11 Labeling Effort on Subject Programs

Subject Programs	Dms	Raptor	Sequoia	Stmt-Addtl	Stmt-Total	Fep-Addtl	Art-Min
All	**79**	98	111	102	240	84	164

5.2.4 Experimental Results: Reducing Cost

This subsection investigates the effectiveness of Dms in reducing cost given a targeted number of labeled test cases. Similar to the single-fault setting, we select 30 test cases and utilize the same method to compare between techniques. We also find that in our evaluation the average debugging cost of using Dms will not reduce significantly even if more labeled test cases than 50 are added further (see Figure 7).

Summary. Table 12, 13, and 14 summarize the comparison between our method and the existing prioritizing techniques. Table 12 illustrates the distributions of Dms's performance against other techniques. For example, the first row shows that Dms performs better than Raptor on 34.68% of the faulty versions, worse on 31.79% of the faulty versions, and shows no improvement on 33.53% of the faulty versions. The first row of Table 13 characterizes the degree of positive improvement of Dms over Raptor. As the table indicates, half of the 34.68% faulty versions with positive improvement values have improvements between 0.03% and 1.05%, and the other half have improvements between 1.05% and 46.75%. The average positive improvement of Dms over Raptor is 5.95%. Table 14 illustrates the degree of negative deterioration of Dms over other techniques. The first row shows that, half of the 31.79% faulty versions for which Dms performs worse than Raptor have deterioration between 0.23% and 2.94%, and the other half have deterioration between 2.94% and 53.30%. The average deterioration of Dms from Raptor is 8.54%.

We conduct paired Wilcoxon signed-rank test to confirm the difference in performance between Dms and six existing prioritization techniques. The statistical test result rejects the null hypothesis and suggests that the performance

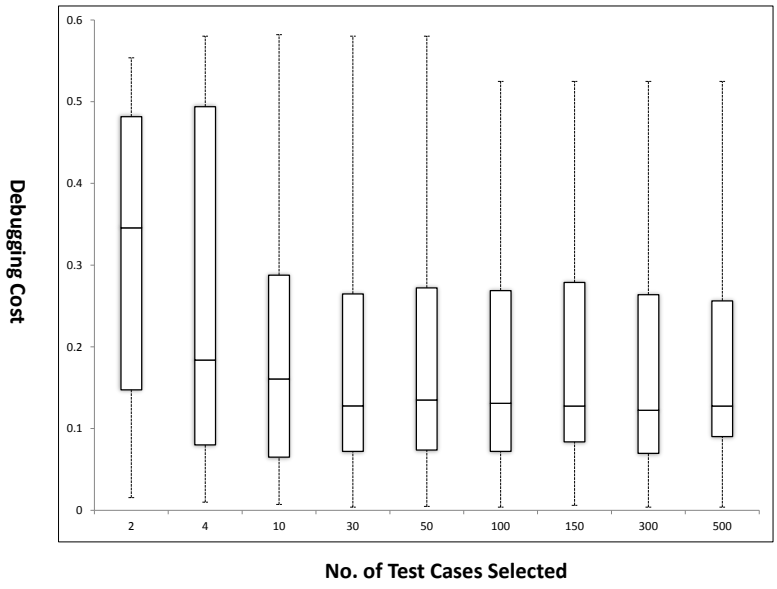

Fig. 7 Average Cost of DMS when Selecting Different Numbers of Test Cases.

Table 12 Comparison of Prioritization methods.

Test Prioritization Method	Positive	Negative	Neutral
DMS vs RAPTOR	**34.68%**	31.79%	33.53%
DMS vs SEQUOIA	**46.24%**	39.31%	14.45%
DMS vs STMT-ADDTL	**50.29%**	28.23%	21.39%
DMS vs STMT-TOTAL	**71.10%**	24.86%	4.05%
DMS vs FEP-ADDTL	**51.45%**	29.48%	19.08%
DMS vs ART-MIN	**71.68%**	23.70%	4.62%

Table 13 Distribution of positive improvements.

Test Prioritization Method	Max	Mean	Median	Min
DMS vs RAPTOR	46.75%	5.95%	1.05%	0.03%
DMS vs SEQUOIA	51.75%	18.31%	14.31%	0.56%
DMS vs STMT-ADDTL	54.24%	10.67%	4.50%	0.04%
DMS vs STMT-TOTAL	56.31%	19.25%	25.42%	0.19%
DMS vs FEP-ADDTL	99.05%	17.94%	9.04%	0.02%
DMS vs ART-MIN	99.13%	42.96%	36.83%	0.14%

Table 14 Distribution of negative deterioration.

Test Prioritization Method	Max	Mean	Median	Min
Dms vs Raptor	53.30%	8.54%	2.94%	0.23%
Dms vs Sequoia	52.00%	8.49%	4.37%	0.19%
Dms vs Stmt-Addtl	53.86%	10.88%	4.87%	0.14%
Dms vs Stmt-Total	51.38%	10.56%	7.10%	0.13%
Dms vs Fep-Addtl	47.13%	10.72%	5.89%	0.04%
Dms vs Art-Min	46.21%	3.33%	2.01%	0.16%

differences between Dms and other techniques are statistically significant at 95% confidence interval.

Detailed Comparison. Similar to the single-fault setting, we show the comparison between Dms and three methods, Raptor, Fep-Addtl and Art-Min, in terms of fault localization costs in Figure 8, 9, and 10.

Dms vs Fep-Addtl. Figure 8 presents the comparison between Dms and Fep-Addtl over all faulty versions that show cost differences. The comparison shows that Dms is better than Fep-Addtl on 89 versions, out of 140 versions that show differences in cost, but performs worse than the Fep-Addtl on 51 versions. The positive improvement ranges from 0.02% to 99.05%, with an average of 17.94%.

Dms vs Art-Min. We compare the effectiveness of Dms to the best variant of *Adaptive Random Test Prioritization*(Art), namely Art-Min (Jiang et al, 2009; González-Sanchez et al, 2011b,a). Figure 9 shows the results of the study in which Art-Min is used as the baseline method. The comparison shows that Dms is better than Art-Min on 124 versions, out of 165 versions that show differences in cost, but performs worse than the Art-Min on 41 versions.

Dms vs Raptor. Figure 10 shows the comparison between Dms and Raptor. The comparison shows that Dms is better than Raptor on 60 versions, out of 115 versions that show differences in cost, but performs worse than the Raptor on 55 versions. The average deterioration (8.54%) of Dms (Table 14 is higher than its average improvement (5.95%) in comparison with Raptor (Table 13), even though Dms reduces the total labelling effort from 98 test cases to 79 (Table 11). We are yet unclear about the reason causing the trade-off in the multi-fault programs. It is very intriguing future work to find ways to balance between labelling effort and diagnostic cost better.

5.3 Discussion and Threats to Validity

As we have noticed, the improvement of Dms in single-fault and multi-fault programs are different. In the 12 single-fault programs, Dms requires the min-

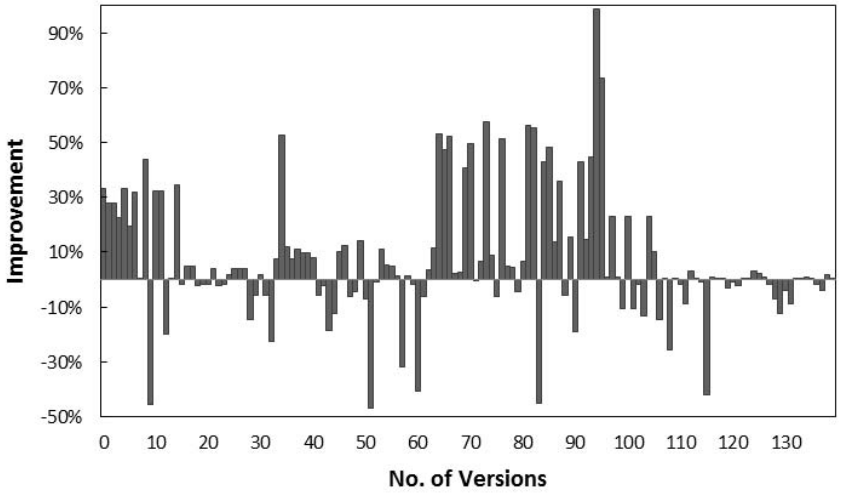

Fig. 8 Improvement of DMS over FEP-ADDTL.

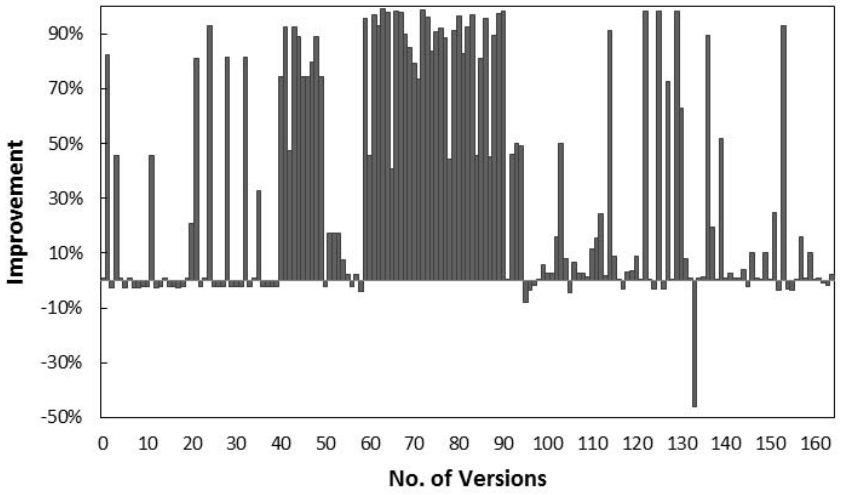

Fig. 9 Improvement of DMS over Art-MIN.

imal amount of labeling effort by achieving 67.7% labeling reduction on Unix programs and 10% reduction on Siemens programs in comparison with the existing best approach—RAPTOR. While in the 8 multi-fault programs, DMS achieves 5.95% labeling reduction in comparison with the existing best approach—FEP-ADDTL. The phenomenon happens since we consider the worst cases of DMS in multi-fault programs, i.e., we consider the root case d_* with the

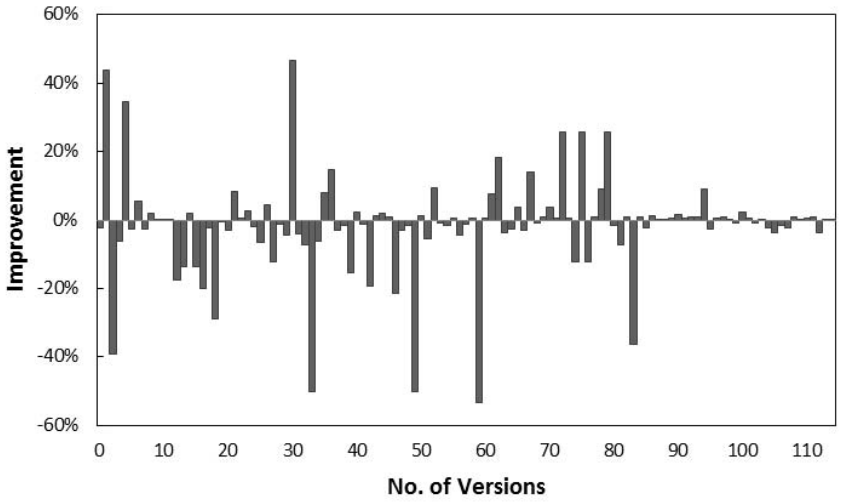

Fig. 10 Improvement of DMS over RAPTOR.

lowest suspiciousness score. In some versions of the multi-fault programs, DMS needs more test cases to achieve 101% of base line effectiveness than that of FEP-ADDTL. For example, in the version 2 of the program print_token2, DMS need to label 500 test cases to achieve 101% of base line effectiveness, while FEP-ADDTL only requires 59 test cases. Thus, the average reduced number of test cases for DMS in multi-fault program is not as high as that in single-fault programs.

However, the improvements of DMS in reducing cost are statistically significant for both single-fault and multi-fault programs at 95% confidence interval via paired Wilcoxon signed-rank tests. Moreover, although we notice that DMS reduce the number of test case of Fep-Addtl by 5.95% in multi-fault programs, but there are 51.45% fault versions that DMS show positive improvement over Fep-Addtl, and 29.48% fault versions that DMS show negative deterioration over Fep-Addtl, while the number for single-fault programs are 40.16% and 20.08%.

The threats to our studies include the issue of how representative the subjects of our studies are. Since the Siemens programs are small, and larger programs may be subject to different testing and debugging traits. To strengthen the external validity, we include UNIX programs which are real-life programs. These subjects have been adopted for evaluation in many studies (e.g. Jones and Harrold, 2005; Abreu et al, 2009; Santelices et al, 2009).

Another possible threat is that although our method outperforms existing method in 25.2% to 62.99% program versions and gets equivalent cost in around 30% versions, there are still a certain percent of versions that our method does not perform very well. But as we can see in the studies, most of the negative deterioration of those versions are relatively small comparing to

the positive improvements. We also conduct statistical tests to further confirm the advantage of DMS.

There are also many other kinds of threats to validity affecting fault localization techniques in general as listed in a recent study by Steimann et al (2013), such as heterogeneity of test cases, biases in injected faults, unrealistic assumptions about locating and understanding faults, etc. Although we focus on evaluating test case prioritization techniques, instead of fault localization techniques, our work inevitably inherits the threats to validity for fault localization techniques since our evaluation of prioritization techniques is done through the evaluation of fault localization. We hope in future work the threats to validity for both fault localization and test case prioritization techniques can be addressed together.

6 Related Work

In this section, we describe related work on fault localization, defect prediction, test case prioritization, diagnostic prioritization, and automated oracle construction. The survey here is by no means a complete list.

Fault Localization. Over the past decade, many automatic fault localization and debugging methods have been proposed. The ways of calculating suspiciousness for program elements based on program spectra are various, such as *Tarantula* (Jones et al, 2002; Jones and Harrold, 2005), *Ochiai* (Abreu et al, 2009)), SOBER (Liu et al, 2005), *DStar* (Wong et al, 2014), and many others (e.g. Xie et al, 2013; Naish et al, 2011; Lucia et al, 2014; Jiang and Su, 2007). Renieris and Reiss (2003) propose a nearest neighbor fault localization tool called WHITHER that compares the failed execution to the correct execution and reports the most ambiguity locations in the program. Zeller (2002) applies *Delta Debugging* to search for the minimum state differences between a failed execution and a successful execution that may cause the failure. Liblit et al (2003) consider predicates whose true evaluation correlates with failures are more likely to be the root cause. Campos et al (2013) propose ENTBUG which applies entropy theory to guide test case generation.

Test Case Prioritization. Test case prioritization techniques are initially proposed for early fault detection in regression testing. Rothermel et al (2001) show the coverage-based and Fault-exposing-potential based approaches can improve the rate of fault detection of test suites. Elbaum et al (2002) further investigate "version-specific prioritization" on different profile granularities. (Li et al, 2007) show that *Additional Greedy Algorithm* is among the best approaches for regression test case prioritization. Baudry et al (2006) propose *Dynamic Basic Block* (DBB) for test suite reduction. Their method focuses on the number of DBBs. González-Sanchez et al (2011a) further consider group sizes for test suite reduction.

Oracle Construction. Although in recent years, many studies (e.g. Pacheco and Ernst, 2005; Xie, 2006; Bowring et al, 2004) aim to automatically generate

test oracles, they are often heavy weight, based on certain assumption and thus applicable to specific scenarios. *Eclat* (Pacheco and Ernst, 2005) can generate assertions based on a learning model, but they assume correct executions. Xie (2006) proposes a method called *Orstra* for oracle checking. Bowring et al (2004) propose ARGO which selects test cases inducing unknown behaviors to actively construct test oracles for improving test quality. The approach is more suitable for regression testing. Our approach complements these studies by reducing the effort needed for the purpose of fault localization.

7 Conclusion and Future Work

This paper proposes a new concept and technique named *Diversity Maximization Speedup* (DMS) aiming to minimize the amount of effort in manual oracle construction, while still permitting effective fault localization. In comparison with six other existing prioritization techniques on 12 *C* programs, we have shown that our technique requires on average a smaller number of labelled test cases to achieve the targeted diagnostic cost of subsequent fault localization techniques, and that if the same number of labelled test cases are allowed, it can choose test cases that may be more effective in reducing debugging cost. We have shown that the improvements made by our technique on real-life programs over other existing techniques are statistically significant.

In future, we will evaluate the proposed approach on more subject programs. We will also explore the possibility of adopting more sophisticated trend analysis methods.

8 Acknowledgement

This work is partially supported by NSFC Program (No.61073006 and 61103032), Tsinghua University project 2010THZ0, and National Key Technology R&D Program of the Ministry of Science and Technology of China (No2013BAH01B03). We thank researchers at University of Nebraska–Lincoln, Georgia Tech, and Siemens Corporate Research for the Software-artifact Infrastructure Repository. We would also like to thank the anonymous reviewers for providing us with constructive comments and suggestions.

References

Abreu R, Zoeteweij P, Golsteijn R, van Gemund A (2009) A practical evaluation of spectrum-based fault localization. Journal of Systems and Software

Ali S, Andrews J, Dhandapani T, Wang W (2009) Evaluating the accuracy of fault localization techniques. In: IEEE/ACM International Conference onAutomated Software Engineering (ASE), pp 76–87

Arcuri A, Briand LC (2011) Adaptive random testing: an illusion of effectiveness? In: ISSTA, pp 265–275

Artzi S, Dolby J, Tip F, Pistoia M (2010) Directed test generation for effective fault localization. In: ISSTA, pp 49–60

Baah GK, Podgurski A, Harrold MJ (2010) Causal inference for statistical fault localization. In: ISSTA, pp 73–84

Baah GK, Podgurski A, Harrold MJ (2011) Mitigating the confounding effects of program dependences for effective fault localization. In: SIGSOFT FSE, pp 146–156

Baudry B, Fleurey F, Traon YL (2006) Improving test suites for efficient fault localization. In: ICSE, pp 82–91

Beizer B (1990) Software Testing Techniques, 2nd edn. International Thomson Computer Press, Boston

Bowring JF, Rehg JM, Harrold MJ (2004) Active learning for automatic classification of software behavior. In: ISSTA, pp 195–205

Campos J, Abreu R, Fraser G, d'Amorim M (2013) Entropy-based test generation for improved fault localization. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, pp 257–267

Cleve H, Zeller A (2005) Locating causes of program failures. In: ICSE

Debroy V, Wong WE (2013) A consensus-based strategy to improve the quality of fault localization. Softw, Pract Exper 43(8):989–1011

Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal 10(4):405–435

Elbaum S, Malishevsky AG, Rothermel G (2002) Test case prioritization: A family of empirical studies. In: IEEE TSE, vol 28, pp 159–182

Godefroid P, Klarlund N, Sen K (2005) Dart: Directed automated random testing. In: PLDI, pp 213–223

González-Sanchez A, Abreu R, Groß HG, van Gemund AJC (2011a) Prioritizing tests for fault localization through ambiguity group reduction. In: ASE, pp 83–92

González-Sanchez A, Piel É, Abreu R, Groß HG, van Gemund AJC (2011b) Prioritizing tests for software fault diagnosis. Softw, Pract Exper 41(10):1105–1129

Graybill FA, Iyer HK (1994) Regression Analysis: Concepts and Applications. Duxbury Press

Hamlet R (1977) Testing programs with the aid of a compiler. IEEE TSE 3(4):279–290

Jiang B, Zhang Z, Chan WK, Tse TH (2009) Adaptive random test case prioritization. In: ASE, pp 233–244

Jiang B, Chan WK, Tse TH (2011) On practical adequate test suites for integrated test case prioritization and fault localization. In: QSIC, pp 21–30

Jiang L, Su Z (2007) Context-aware statistical debugging: from bug predictors to faulty control flow paths. In: IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 184–193

Jones J, Harrold M (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: ASE

Jones J, Harrold M, Stasko J (2002) Visualization of test information to assist fault detection. In: ICSE, Orlando, Florida, pp 467–477
Li Z, Harman M, Hierons R (2007) Search algorithms for regression test case prioritization. IEEE TSE 3:225–237
Liblit B, Aiken A, Zheng AX, Jordan MI (2003) Bug isolation via remote program sampling. In: PLDI, pp 141–154
Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI (2005) Scalable statistical bug isolation. In: ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)
Liu C, Yan X, Fei L, Han J, Midkiff SP (2005) SOBER: Statistical model-based bug localization. In: ESEC/FSE
Lucia, Lo D, Jiang L, Thung F, Budi A (2014) Extended comprehensive study of association measures for fault localization. Journal of Software: Evolution and Process 26(2):172–219
Nainar PA, Chen T, Rosin J, Liblit B (2007) Statistical debugging using compound boolean predicates. In: ISSTA, pp 5–15
Naish L, Lee HJ, Ramamohanarao K (2011) A model for spectra-based software diagnosis. ACM TOSEM 20(3):11:1–11:32
National Institute of Standards and Technology (NIST) (2002) Software Errors Cost U.S. Economy $59.5 Billion Annually
Pacheco C, Ernst MD (2005) Automatic generation and classification of test inputs. In: ECOOP, pp 504–527
Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: ISSTA, pp 199–209
Renieris M, Reiss S (2003) Fault localization with nearest neighbor queries. In: ASE, pp 141–154
Rothermel G, Untch RH, Chu C, Harrold MJ (2001) Prioritizing test cases for regression testing. In: IEEE TSE, pp 929–948
Santelices RA, Jones JA, Yu Y, Harrold MJ (2009) Lightweight fault-localization using multiple coverage types. In: ICSE, pp 56–66
Sen K, Marinov D, Agha G (2005) Cute: a concolic unit testing engine for c. In: ESEC/SIGSOFT FSE, pp 263–272
Steimann F, Frenkel M (2012) Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In: IEEE International Symposium on Software Reliability Engineering (ISSRE), pp 121–130
Steimann F, Frenkel M, Abreu R (2013) Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: International Symposium on Software Testing and Analysis (ISSTA), pp 314–324
Wilcoxon F (1943) Individual comparisons by ranking methods. In: Biometrics, pp 80–3
Wong WE, Debroy V, Gao R, Li Y (2014) The dstar method for effective software fault localization. IEEE Transactions on Reliability 63(1):290–308
Xie T (2006) Augmenting automatically generated unit-test suites with regression oracle checking. In: ECOOP, pp 380–403

Xie X, Chen TY, Kuo FC, Xu B (2013) A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM TOSEM 22(4):31:1–31:40

Xu X, Debroy V, Wong WE, Guo D (2011) Ties within fault localization rankings: Exposing and addressing the problem. International Journal of Software Engineering and Knowledge Engineering 21(6):803–827

Zeller A (2002) Isolating cause-effect chains from computer programs. In: FSE, pp 1–10, DOI http://doi.acm.org/10.1145/587051.587053

Zhang L, Hao D, Zhang L, Rothermel G, Mei H (2013) Bridging the gap between the total and additional test-case prioritization strategies. In: ICSE, IEEE Press, pp 192–201