Diversity Maximization Speedup for Fault Localization

Liang Gong[1]*, David Lo[2], Lingxiao Jiang[2], and Hongyu Zhang[1]
[1]School of Software, Tsinghua University, Beijing 100084, China
Tsinghua National Laboratory for Information Science and Technology (TNList)
[1]gongliang10@mails.tsinghua.edu.cn, hongyu@tsinghua.edu.cn
[2]School of Information Systems, Singapore Management University, Singapore
[2]davidlo@smu.edu.sg, lxjiang@smu.edu.sg

ABSTRACT

Fault localization is useful for reducing debugging effort. However, many fault localization techniques require non-trivial number of test cases *with oracles*, which can determine whether a program behaves correctly for every test input. Test oracle creation is expensive because it can take much manual labeling effort. 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.

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. Our experiments with more than 250 bugs from the Software-artifact Infrastructure Repository show (1) that DMS can help existing fault localization techniques to achieve comparable accuracy with on average 67% fewer labeled test cases than previously best test case prioritization techniques, 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.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation, Reliability

Keywords

Test Case Prioritization, Fault Localization

1. INTRODUCTION

Software testing and debugging activities are often labor-intensive, accounting for 30% to 90% of labor spent for a project [7]. 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) [23]. Many automated testing and debugging techniques have been proposed to reduce the high cost in such activities.

Spectrum-based fault localization (e.g., [19, 1, 5]) is an automated debugging techniques that can narrow down the possible locations of software faults and help save developers' debugging time. Many spectrum-based fault localization techniques take a set of executions and labels as input, 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 for a program can 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 [11, 29], 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 human to decide whether the results are good or not, which could be a laborious and error prone activity. Recently, Artzi *et al.* propose a directed test case generation approach for fault localization [3]. They however only handle two kinds of errors in web applications that automated test oracles can be constructed: program crashes and invalid HTML documents. 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?*

*The work was done while the author was visiting Singapore Management University.

ASE '12, September 3-7, 2012, Essen, Germany
Copyright 12 ACM 978-1-4503-1204-2/12/09 ...$15.00.

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	●	●	●	●	●	●	●	●	●	●	●	●	*3*	*9*	*0*	*0*	*0.500*	*0.500*	*0.250*
`int let, dig, c;`	s2																			
`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

Suspicious Group (the groups are ordered according to their suspiciousness)	Selected Test Case	Program Spectra												Normalized *Ochiai* Score										
		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

Figure 1: Running Example

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 sufficient 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.*

1.1 Running Example

Figure 1(a) and 1(b) illustrate how our concept helps reduce the number of test cases for effective fault localization.

There are 11 statements s_1...s_{11} in the program in Figure 1(a) (adapted from previous papers [13, 16]), 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 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* [1], *Tarantula* [18], and *Jaccard* [1] 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 suspiciousness scores 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 [9]). In total, we analyze 254 faults, and demonstrate that our approach significantly 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 [27, 10], Art [17]), fault-exposing potential based [27], and *diagnostic prioritization* [12, 13, 16], and our approach achieves, on average, test case reduction rates from 10% to 96%.

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), 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 debugging

cost. In comparison with other test case selection techniques, we show, with Wilcoxon signed-rank test [30] 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

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., [22, 18]) show that such techniques can be effective in guiding developers to locate faults. Parnin *et al.* conduct a user study [25] 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* [18], *Ochiai* [1], and *Jaccard* [1]. Given a program element s, $N_{ef}(s)$ is the number of ***failed*** executions that ***execute*** s; $N_{np}(s)$ numerates ***passed*** executions that do ***not*** hit s; by the same token, $N_{nf}(s)$ counts ***failed*** executions that do ***not*** hit s and $N_{ep}(s)$ counts ***passed*** executions that ***execute*** s.

Table 1: Spectrum-based fault localization

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

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

In [27], Rothermel *et al.* define the problem of test case prioritization as follows:

Definition 2.1 (**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 [27] is a test case prioritization strategy that assigns higher priorities to a test case that executes more statements in a program. **Stmt-Addtl** [27] 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**) [17] 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 [17]. However, recent study [2] 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 [27] 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* [15]: 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.* [16] 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.* 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** [13], sort test cases so that the information entropy of the diagnostic distribution can be minimized. Soon after, González-Sanchez *et al.* propose another strategy called *Ambiguity Group Reduction* to sort test cases. In their tool named **Raptor** [12], 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.

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.

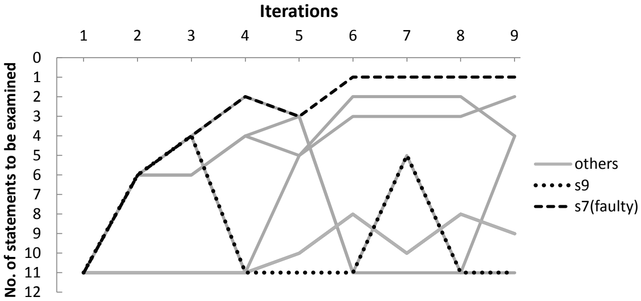

Figure 2: Motivating Example

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* [1] 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 [12] 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 $T_{\mathcal{U}}$. The vertical axis is the rank[1] of a statement sorted based on suspiciousness. 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)?

3.2 Formal Definition of DMS

DEFINITION 3.1 (**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 3.1, 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 the previous section which 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

[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.

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

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	...

Linear Model Construction. Then we use *linear regression analysis* [14] 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. Here we define the change-potential of a program element with trend $\mathcal{T}$ as:

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

$\hat{\beta}_1$ is estimated by *least squares* and $\hat{\sigma}_{\beta_1}$ is the error of estimating β_1 [14]. 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. Using this metric, our method isolates trends that evolve in a monotonous and stable way. Table 3 shows a few example trends and their change-potentials according to Equation 2.

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

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 on the change-potential($\mathcal{W}_{\mathcal{T}}$). First, program elements with the same suspiciousness scores are grouped together, they are termed as *suspicious group* in this paper. 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 $\mathcal{T}_d$ is the change-potential of the program element d based on the labeled execution trace profiles.

The overall change-potential score of all suspicious groups(G) are 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 ($G \Leftarrow t$), and chooses the test case that can maximize the difference 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 $\frac{1}{2} \times (0.4 + 0.6) = 0.5$.

Note that DMS does not intentionally increase suspiciousness scores of promising statements which could lead to *confirmation bias.* More specifically, 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 *pass*, or it is not covered in the next selected trace and the trace is labeled *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 Figure 3) takes in a set of unlabeled traces $T_{\mathcal{U}}$ and the labeling 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) is also used as an input because a developer usually starts debugging only when at least one test fails, 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.

In the pseudocode, `manual_label`(t) asks a user to check the correctness of the outcome from the test case t. Proce-

Table 4: Evolution of Suspiciousness Scores for the Running Example in Table 1(a) using RAPTOR [12].

Ambiguity Group (the groups are ordered according to their suspiciousness)	Selected Test Case	Program Spectra												Normalized *Ochiai* Score										
		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	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

```
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 prioritization technique $\mathcal{P}$
 3: while $|T_{tmp}| \le k$ and $|T_{tmp}| \le w$ 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 $|T_{\mathcal{S}}| \le k$ 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 $T_{\mathcal{S}}$
```

Figure 3: Diversity Maximization Speedup

dure div(T) counts the number of unique suspicious scores (diversity) generated by f_T, which is defined as follows:

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

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 pseudocode 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 also helps to confirm the previously selected test cases and make the final result more robust.

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 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 a suspicious group g. All non-zero suspicious groups compose a group set $G = \{g\}$. RAPTOR would then choose t_8 and ask developer to label (*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 $\{s_5,s_6,s_7,s_8,s_9,s_{10}\}$, they are called g_1 and g_2 respectively. At this time, the 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: $\{t_i | 1 \le i \le 12 \wedge i \notin \{2, 8\}\}$ 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 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 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 Table 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 ($t_8 \rightarrow t_6 \rightarrow t_4 \rightarrow t_7 \rightarrow t_9$).

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. Section 5.1 gives details about experimental setup. In Section 5.2, we introduce the subject programs in our study. Section 5.3 shows the results followed by Section 5.4 discussing the threats to validity.

5.1 Experimental Setup

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

We compare the effectiveness of different prioritization methods based on the diagnostic cost when the same number of test cases are selected. The diagnostic cost is defined as follows:

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

where $\mathcal{D}$ consists of all program elements appearing in the program. We calculate the average cost as the percentage of elements that developers have to examine until locating the root cause(d_*) of failure. 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, we use Raptor as the bootstrapping technique ($\mathcal{P}$ in Figure 3). During the bootstrapping process, w is set to 10 to facilitate trend analysis.

Following [16], 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.

5.2 Subject Programs

We use five real *C* programs and seven Siemens test programs from the *Software-artifact Infrastructure Repository* (SIR) [9]. 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 size. Following [19, 1], we exclude faults not directly observable by the profiling tool[2] (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.

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

5.3 Experimental Results

In this subsection, we conduct several controlled experiments to show the effectiveness of Dms.

5.3.1 *Effectiveness on Reducing The Number of Test Cases Needed for a Target Cost*

We compare Dms with previous test case prioritization techniques in terms of labeling effort when given an expected fault localization 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. Then we define x% base line effectiveness (c_x) as follows:

$$c_x = \frac{x}{100} \times c \tag{8}$$

Table 6 shows how many labels are needed on average to achieve 101% base line effectiveness (i.e., within 1% accuracy lost) 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).

Table 6: Labeling Effort on Subject Programs

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

5.3.2 *Effectiveness on Reducing Cost for a Given Number of Labeled Test Cases*

We select 30 test cases, 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 are added further (See Figure 4). 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

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

on the selected program spectra. A prioritization technique that obtains a lower cost is better.

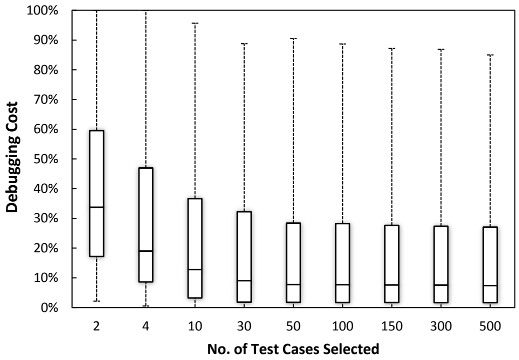

Figure 4: Average Cost of Dms when Selecting Different Numbers of Test Cases.

Following [4, 5] 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 equal 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). 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 P_A over P_B is 10%, which means that developers would examine 10% fewer statements if P_A is deployed.

Summary Table 7 and 8 summarize the comparison between our method and the existing prioritizing techniques, the results show that our method outperforms all of them.

Table 7: Comparison of Prioritization methods.

Test Pri. Tech.	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%

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 33.46% faulty versions with positive improvement values have improvements between 0.03% and 7.71%, and the other half have improvements between 7.71% and 77.42%. The average positive improvement of Dms over Raptor is 7.71%.

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 Dms is statistically significantly better than the existing best approach on Unix programs at 95% confidence interval.

Detailed Comparison Table 6 shows that Raptor, Fep-Addtl and Art-Min achieve 101% base line effectiveness with less than 500 test cases on subject programs. Due to limited space, we only show the comparison between Dms and these methods in detail.

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%

Figure 5, 6, and 7 show the comparison between different prioritization techniques based on fault localization Cost. 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 [27, 10] 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) [13] [3] which is also used in our study.

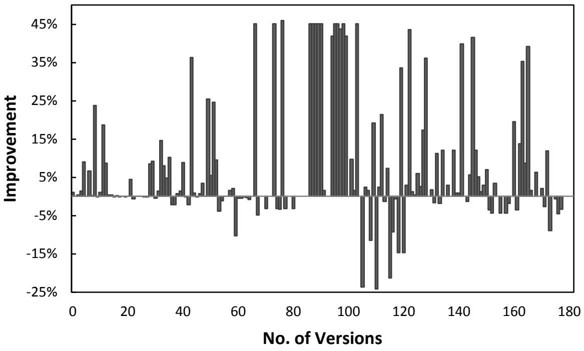

Figure 5: Improvement of Dms over Fep-Addtl.

Figure 5 presents the comparison between Dms and Fep-Addtl over all faulty versions. 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 due to the limited space. 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 is better than Fep-Addtl. Out of 153 versions that show differences in Costs, our prioritization method performs better than Fep-Addtl on 102 versions but performs worse than the Fep-Addtl on

[3] 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.

51 versions. The positive improvement ranges from 0.03% to 45.90%, with an average of 6.35%.

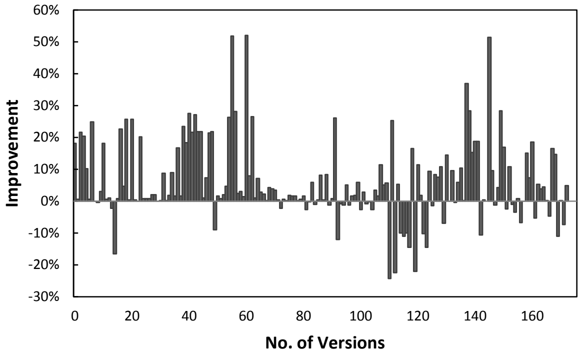

Figure 6: Improvement of Dms over Art-Min.

Dms vs Art-Min In this study we compare the effectiveness of Dms to *Adaptive Random Test Prioritization*(Art) [17]. There are various strategies for Art, in this experiment we only compare with the best one: Art-Min [17, 13, 12]. Figure 6 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 Costs, our prioritization method performs better than Art-Min on 80 versions but performs worse than the Art-Min on 49 versions.

Dms vs Raptor Figure 7 shows the comparison between Dms and Raptor on Unix programs. Here we use Raptor as the reference metric. The comparison shows that Dms is better than Raptor. On Unix programs Dms outperforms Raptor on 20 versions by at least 1% cost, and only 5 versions worse than Raptor over 1% cost.

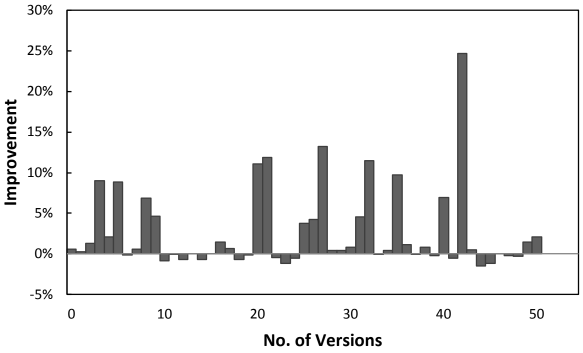

Figure 7: Improvement of Dms over Raptor on Unix programs.

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 comparison on Unix programs. This is probably due to the small software size. On Siemens programs the existing best approach can reach 101% of the base line effectiveness by only selecting less than 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.

5.4 Threats to Validity

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 works [18, 1, 28].

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 improvements of those versions are relatively small or even trivial comparing to the positive improvements. We also conduct statistical test to further confirm the superiority of Dms.

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 are various, including state-of-arts (e.g. *Tarantula* [19, 18] and *Ochiai* [1]). Renieris and Reiss propose a nearest neighbor fault localization tool called Whither [26] that compares the failed execution to the correct execution and reports the most suspicious locations in the program. Zeller applies *Delta Debugging* to search for the minimum state differences between a failed execution and a successful execution that may cause the failure [32]. Liblit *et al.* consider predicates whose true evaluation correlates with failures [21] are more likely to be the root cause.

Test Case Prioritization Test case prioritization techniques are initially proposed for early fault detection in regression testing. Rothermel *et al.* [27] show the coverage-based and Fault-exposing-potential based approaches can improve the rate of fault detection of test suites. Elbaum *et al.* [10] further investigate "version-specific prioritization" on different profile granularities. In [20], Li *et al.* show that *Additional Greedy Algorithm* is among the best approaches for regression test case prioritization. Baudry *et al.* propose *Dynamic Basic Block* (Dbb) [6] for test suite reduction. Their method focuses on the number of Dbbs. González-Sanchez *et al.* [12] further consider group size.

Oracle Construction Although in recent years, many studies [24, 31, 8] aim to automatically generate test oracles, they are often heavy weight, based on certain assumption and thus applicable to specific scenarios. *Eclat* [24] can generate assertions based on a learning model, but they assume correct executions. Xie proposes a method called *Orstra* [31] for oracle checking. Bowring *et al.* propose Argo [8] 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 technique aiming to minimize the amount of effort in manual oracle construction, while still permitting effective fault localization. In comparison with existing prioritization techniques on 12 *C* programs, we have shown that: our method only requires on average a small number of test cases to accomplish the target average cost within 1% accuracy lost, and outperform existing methods in terms of reducing debugging cost for the subject programs. We have also shown that the differences on real-life programs 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 grant 61073006 and Tsinghua University project 2010THZ0. 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.

9. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 2009.

[2] A. Arcuri and L. C. Briand. Adaptive random testing: an illusion of effectiveness? In *ISSTA*, pages 265–275, 2011.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA*, pages 49–60, 2010.

[4] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *ISSTA*, pages 73–84, 2010.

[5] G. K. Baah, A. Podgurski, and M. J. Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *SIGSOFT FSE*, pages 146–156, 2011.

[6] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *ICSE*, pages 82–91, 2006.

[7] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, Boston, 2nd edition, 1990.

[8] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA*, pages 195–205, 2004.

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

[10] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. In *IEEE TSE*, volume 28, pages 159–182, 2002.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[12] A. González-Sanchez, R. Abreu, H.-G. Groß, and A. J. C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *ASE*, pages 83–92, 2011.

[13] A. González-Sanchez, É. Piel, R. Abreu, H.-G. Groß, and A. J. C. van Gemund. Prioritizing tests for software fault diagnosis. *Softw., Pract. Exper.*, 41(10):1105–1129, 2011.

[14] F. A. Graybill and H. K. Iyer. *Regression Analysis: Concepts and Applications*. Duxbury Press, 1994.

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

[16] B. Jiang, W. K. Chan, and T. H. Tse. On practical adequate test suites for integrated test case prioritization and fault localization. In *QSIC*, pages 21–30, 2011.

[17] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244, 2009.

[18] J. Jones and M. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[19] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault detection. In *ICSE*, 2002.

[20] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE TSE*, 3:225–237, 2007.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, pages 141–154, 2003.

[22] P. A. Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound boolean predicates. In *ISSTA*, pages 5–15, 2007.

[23] National Institute of Standards and Technology (NIST). *Software Errors Cost U.S. Economy* $59.5 *Billion Annually*, June 28, 2002.

[24] C. Pacheco and M. D. Ernst. Automatic generation and classification of test inputs. In *ECOOP*, pages 504–527, 2005.

[25] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, pages 199–209, 2011.

[26] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, pages 141–154, 2003.

[27] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. In *IEEE TSE*, pages 929–948, 2001.

[28] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, pages 56–66, 2009.

[29] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[30] F. Wilcoxon. Individual comparisons by ranking methods. In *Biometrics*, pages 80–3, 1943.

[31] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, 2006.

[32] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, pages 1–10, 2002.