On the Integration of Test Adequacy, Test Case Prioritization, and Statistical Fault Localization*

Bo Jiang
The University of Hong Kong
Pokfulam, Hong Kong
bjiang@cs.hku.hk

W. K. Chan†
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

Abstract—Testing and debugging account for at least 30% of the project effort. Scientific advancements in individual activities or their integration may bring significant impacts to the practice of software development. Fault localization is the foremost debugging sub-activity. Any effective integration between testing and debugging should address how well testing and fault localization can be worked together productively. How likely does a testing technique provide test suites for effective fault localization? To what extent may such a test suite be prioritized so that the test cases having higher priority can be effectively used in a standalone manner to support fault localization? In this paper, we empirically study these two research questions in the context of test data adequacy, test case prioritization and statistical fault localization. Our preliminary postmortem analysis results on 16 test case prioritization techniques and four statistical fault localizations show that branch-adequate test suites on the Siemens suite are unlikely to support effective fault localization. On the other hand, if such a test suite is effective, around 60% of the test cases can be further prioritized to support effective fault localization, which indicates that the potential savings in terms of effort can be significant.

Keywords: debugging, testing, continuous integration.

I. INTRODUCTION

Testing is a software development activity to detect the presence of faults in programs. However, simply knowing the presence of faults is inadequate in practice — the developers should locate the faults and fix them — they want to debug the faulty programs. However, once the program is modified, retesting the modified program is necessary. Testing and debugging should be tightly integrated. Indeed, they account for at least 30% of project effort [18]. The advancement in the integration of the techniques and processes for these two activities is an area to make software development more predictable and productive.

Continuous Integration (CI) [6] is a software development process, in which developers submit their code modules of an application to a CI server and the CI server compiles the application and tests the application to provide quick feedbacks to individual developers. Consequently, such an automated process should construct the executable version of the application and perform a skimmed version of regression testing in each round of application integration.

In a CI process, developers often find regression testing to be the bottleneck [6]. Fowler proposed to perform the integration in stages [8][9]. In a staged CI process, after a developer submits a code module to a CI server, the server first performs a *commit build*, which runs a small amount of test cases to enable the developers to continue to work on the code module if the application passes the test. The second stage is to test the application comprehensively at the CI server, which usually takes much more time and resources than that of the commit build. The testing in this two-stage build process may involve test case prioritization [12] so that a set of higher priority test cases can be identified and is scheduled to be executed earlier (in the commit build) than those test cases to be executed in the subsequent build.

If the CI server detects any failure during a build, it may send a bug report to the developers who submit the code modules [6][8]. The developers then analyze the bug reports, locate the faults in the code modules, fix them, and resubmit the repaired code modules to the CI server. This triggers the server to perform a new round of integration that either accepts the submitted work or rejects it [6][8]. Moreover, while a developer is carrying out the debugging process, the CI server may undergo a later stage build, which uses a larger and more comprehensive test suite to test the application.

After receiving a bug report, if the developers use manual methods to locate faults, the debugging process can be laborious. To address the fault localization problem [14], researchers have proposed semi-automatic fault-localization techniques [3][14][19] to help the developers to locate faults. These fault localization techniques use program coverage information as well as the test outcomes of test cases to help developer to assess the suspiciousness of program entities.

Previous studies have explored the integration problem of test case prioritization and statistical fault localization [12]. In a CI environment, if only a portion of a test suite is selected for execution in a commit build, the coverage statistics achieved by different prioritized test suites may be different. Because statistical fault localization techniques use the execution statistics of test cases to pinpoint suspicious program entities [12][14], understanding the impact of test case prioritization techniques on the effectiveness of

* This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. CityU 123207 and HKU 717308).

† All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. Tel: (+852) 2788 9684. Fax: (+852) 2788 8614. Email: wkchan@cs.cityu.edu.hk.

fault localization techniques is crucial. Jiang et al. [12] studied the effectiveness of using prioritized test suites generated by different test case prioritization techniques to locate faults through statistical fault localization techniques. They found that random ordering and additional-statement can be more effective than the other techniques in terms of relative mean percentage of code examined to locate the faults.

In this paper, we supplement their study. We study the same research topic from a backward and theoretical perspective. We firstly suppose that an effective fault localization can be achieved based on a given test suite. We then ask a couple of research questions. In this paper, we report the results on using the *all-edges* (aka branch coverage) criterion [10] and on the Siemens suite [12]. In the experiment, we first obtain 1000 all-edges adequate test suites and then run four statistical fault localization techniques to compute the fault suspiciousness of the program entities based on these test suites. We examine what if effective fault localization (in terms of a low percentage of code examined to locate the fault successfully) can be achieved, what proportion of such test suites may enable these fault localization techniques to deliver a particular level of effectiveness.

We further prioritize the 1000 all-edge adequate test suites through 16 test case prioritization techniques and input the reordered test suites to four statistical fault localization techniques. To support the CI process, the higher priority test cases can be run in the commit build. We examine the chance of such fragments of prioritized test suites leading to effective fault localization.

We conducted a postmortem analysis results based on the above-mentioned experiment. The results show that branch-adequate test suites on the Siemens suite are unlikely to support effective statistical fault localization. Our results find that only around 10% of branch-adequate test suites can enable the studied statistical fault localization techniques to locate the faults within the top 1% of code examined. Moreover, if such a test suite can be effective, around 60% of them can be further prioritized to support effective statistical fault localization, which indicates that a significant saving could be achieved if a proper identification of such effective adequate test suites can be done.

The main contribution of this paper is fourfold: First, it is the first work to study the effective percentage of adequacy test suites that can help fault localization techniques to locate faults successfully. Second, we report the first analysis on how likely an average test case prioritization technique can effectively support an average fault localization technique. Third, we report an experiment that studies the integration between testing and debugging activities. Our empirical study involves 16 existing test case prioritization techniques and 4 existing fault localization techniques. To the best of our knowledge, it is the largest experiment reported in the literature. Fourth, our findings show that branch-adequate test suites have inadequate information to support existing statistical fault localization techniques to locate faults effectively. On the other hands, executing 60% of an average prioritized test suite is statistically comparable to executing the entire adequate test suite for the scenarios of effective fault localization.

We present the rest of paper as follows: In Section II, we review selected test case prioritization techniques and fault localization techniques. We describe our controlled experimental study in Section III. After that, we present the experiment results and their implications in Section IV. Section V describes the related work followed by a conclusion in Section VI.

II. TECHNIQUES

This section describes the test case prioritization and fault localization techniques involved in our study.

A. Test Case Prioritization Techniques

We follow [12] to organize the test case prioritization techniques into two dimensions. The first dimension is *granularity*: *statement*, *branch* and *function*. The second dimension is *prioritization strategy*. We study *coverage-based* strategy and the *ART-based* strategy. In this paper, the coverage-based strategy can be realized as greedy algorithms reported in [7], which can be further subdivided into the *Total* and *Additional* test case prioritization strategies. The *ART-based* strategy can be implemented as the ART-based prioritization techniques proposed in [13].

a) Coverage-based techniques. When we combine the two coverage-based strategies with the three granularities, we produce six coverage-based techniques: total statement (total-st), total branch (total-br), total function (total-fn), additional statement (addtl-st), additional branch (addtl-br), and additional function (addtl-fn).

The total statement (total-st) test case prioritization technique computes which statements each test case has covered for each program version. It permutes test cases in descending order of the total number of statements achieved by individual test case. When two test cases covered the same number of statements, it just orders them randomly. The total branch (total-br) and the total function (total-fn) test case prioritization techniques are the same as total-st, except that they use branch coverage and function coverage information instead of statement coverage information, respectively.

The additional statement (addtl-st) test case prioritization technique is the same as total-st, except that it selects a test case covering the maximum number of statements not yet covered in each round. When no remaining test case can further improve the statement coverage of the test suite being constructed, the addtl-st will reset all the statements to "not yet covered" and reapply the same procedure on the remaining test cases. When the test cases have covered the same number of additional statements, it just picks one randomly.

The additional branch (addtl-br) and additional function (addtl-fn) test case prioritization techniques are the same as addtl-st, except that they use branch coverage and function coverage information rather than statement coverage information, respectively.

b) ART-based techniques. We summarize the algorithm for the ART-based prioritization techniques [13] as follows. The algorithm accepts a test suite containing a sequence of test cases as its input, and produces a sequence of prioritized test cases. The algorithm prioritizes the test cases by iteratively building a candidate set of test cases and, in turn, picks one test case out of the candidate set until all given test cases have been selected. To generate the candidate set of test cases, the algorithm randomly adds those not yet selected test cases into the candidate set one by one as long as they can increase the code coverage achieved by the candidate set. To decide which candidate test case to be selected from the candidate set, the algorithm uses a function (denoted by f1) to calculate the distance between a pair of test cases and another function (denoted by f2) to select a test case from the candidate set that is farthest away from the set of prioritized test cases.

For function f1, we follow [13] to measure the distance between two test cases using the Jaccard distance based on their code coverage information. Suppose the set of statements (or functions or branches) covered by test case p_j and c_i are $S(p_j)$ and $S(c_i)$, respectively. We have:

$$f_1(p_j, c_i) = 1 - |S(p_j) \cap S(c_i)| \,/\, |S(p_j) \cup S(c_i)|$$

Function f_2 describes the strategy to select the farthest away test case from those already prioritized test cases. We also follow [13] to define the distance between a test case and a set of prioritized test cases as their minimum, average, or maximum test case distance. We then find a candidate test case that is associated with the longest distance with the set of the already selected test cases.

$$f_2(D) = \begin{cases} \min\limits_{0 \le i \le |P|} d_{ij} = \max\limits_{0 \le j \le |C|} \{ \min\limits_{0 \le i \le |P|} d_{ij} \} & (1) \\ \text{avg}\limits_{0 \le i \le |P|} d_{ij} = \max\limits_{0 \le j \le |C|} \{ \text{avg}\limits_{0 \le i \le |P|} d_{ij} \} & (2) \\ \max\limits_{0 \le i \le |P|} d_{ij} = \max\limits_{0 \le j \le |C|} \{ \max\limits_{0 \le i \le |P|} d_{ij} \} & (3) \end{cases}$$

We list all the test case prioritization techniques considered in our study in Table I. A total of 16 test case prioritization techniques are considered including nine ART test case prioritization techniques, six greedy test case prioritization techniques and the random ordering.

B. Fault-Localization Techniques

Researchers have proposed many techniques to help developers locate faults. We revisit four such techniques used in the study. Each technique first computes the suspiciousness of individual statements. One can rank these statements according to their suspiciousness values. Table II summarizes the fault localization techniques.

a) Tarantula [14] computes two metrics, namely, suspiciousness and confidence, according to the coverage information on passed and failed test cases.

TABLE I. TEST CASE PRIORITIZATION TECHNIQUES.

Ref.	Name	Descriptions	
*T*1	Random	Random prioritization	
*T*2	total-st	Total statement	
*T*3	total-fn	Total function	
*T*4	total-br	Total branch	
*T*5	addtl-st	Additional statement	
*T*6	addtl-fn	Additional function	
*T*7	addtl-br	Additional branch	
Ref.	**ART**	**Level of Coverage Information**	**Test Set Distance (f_2)**
*T*8	ART-st-maxmin	Statement	Equation (1)
*T*9	ART-st-maxavg	Statement	Equation (2)
*T*10	ART-st-maxmax	Statement	Equation (3)
*T*11	ART-fn-maxmin	Function	Equation (1)
*T*12	ART-fn-maxavg	Function	Equation (2)
*T*13	ART-fn-maxmax	Function	Equation (3)
*T*14	ART-br-maxmin	Branch	Equation (1)
*T*15	ART-br-maxavg	Branch	Equation (2)
*T*16	ART-br-maxmax	Branch	Equation (3)

The suspiciousness of a statement *s* is given by

$$suspiciousness_T(s) = \frac{\%failed(s)}{\%passed(s) + \%failed(s)}$$

where the function *%failed* tallies the percentage of failed test cases that execute statement *s* (among all the failed test cases in the test suite). The function *%passed* is similarly defined. When two statements have the same suspiciousness value, Tarantula uses a confidence metric, computed as follows, indicates the degree of confidence on a suspiciousness value:

$$confidence(s) = \max(\%\text{failed}(s), \%\text{passed}(s))$$

Tarantula ranks all the statements in a program in descending order of suspiciousness and uses the confidence values to resolve ties.

b) Statistical Bug Isolation. Yu et al. [22] adapted CBI proposed by Liblit et al. [3] to compute the suspiciousness of a statement *s* as follows:

$$suspiciousness_S(s) = \frac{failed(s)}{passed(s) + failed(s)}$$

The function *failed* (*passed*, respectively) tallies the number of test cases for which *s* is executed. We also use Tarantula ranking strategy to rank the statements produced by SBI.

c) Jaccard and Ochiai. Abreu et al. [1] study the use of the Jaccard metric and Ochiai metric as the suspiciousness formulas.

The equation for Jaccard is given by

$$suspiciousness_J(s) = \frac{failed(s)}{totalfailed + failed(s)}$$

The functions *failed* and *passed* have the same meaning as those in SBI. The variable *totalfailed* is the total number of failed test cases in the test suite. The technique ranks the statements similarly to Tarantula.

The equation for Ochiai is given by

$$suspiciousness_O(s) = \frac{failed(s)}{sqrt(totalfailed * (failed(s)+passed(s)))}$$

where *passed*, *failed*, and *totalfailed* carry the same meanings as those in Jaccard. The technique also ranks the statements similarly to Tarantula.

III. EXPERIMENT

A. Research Questions

We study two research questions in this paper.

RQ1: How likely does the all-edge adequacy criterion to provide adequate test suites for a statistical fault localization technique to locate faults effectively?

In previous study on the integration of test case prioritization and fault localization [12], researchers only explore the impact of test case prioritization techniques on the effectiveness of fault localization. However, a test suite used for test case prioritization may be constructed using different approaches such as random testing or test data adequacy criteria. In this study, we use the all-edges test adequacy criteria to select the test suites from a test pool. We choose all-edges because the results of all-edges adequate test suites on test case prioritization have been reported in the literature [7] and it is a relatively practical criterion.

RQ2: To what extent may such a test suite be prioritized so that the test cases having higher priority can be used in a standalone manner for fault localization techniques to locate faults effectively?

In the CI environment, only parts of the test suite may be executed in the first stage build. Answering this question helps test engineers to decide whether the effort on prioritizing and executing more test cases in the main cycle is worthwhile.

TABLE II. STATISTICAL FAULT LOCALIZATION TECHNIQUES

Technique	Basic formula	Tiebreaker
Tarantula [14]	$\frac{\%failed(s)}{\%failed(s) + \%passed(s)}$	*max*(%failed(s), %passed(s))
Statistical Bug Isolation (SBI) [4]	$\frac{failed\ (s)}{passed\ (s)\ +\ failed\ (s)}$	–
Jaccard [1]	$\frac{failed\ (s)}{totalfailed\ +\ failed\ (s)}$	–
Ochiai [1]	$\frac{failed\ (s)}{\sqrt{totalfailed * (\ failed\ (s) + \text{passed(s)})}}$	–

B. Subject Program and Test Suites

We reuse the *Siemens* programs as the subject programs of our controlled experimental study (see Table III).

The Researchers in Siemens originally created *Siemens* programs to support research on data-flow and control-flow test adequacy criteria [10]. We download the *Siemens* subject programs from the Software-artifact Infrastructure Repository (SIR) [7].

The Siemens suite consists of seven programs. Each program comes with several faulty versions, a test pool and a set of branch-adequate test suites. Following the documents of SIR, we exclude the faulty versions whose faults cannot be revealed by any test case. We also follow [7] to remove the faulty versions whose faults are too easy (more than 25% of the test cases in the pool can detect them). Besides, since we use gcov to collect the program profile (including the execution count of each statement) dynamically, we exclude those faulty versions where segmentation fault occurs because gcov cannot handle these cases. Finally, we use all the remaining *122* faulty versions in our experiment.

TABLE III. SUBJECT PROGRAMS

Subject	Versions	LOC	Size of Test Pool	Description
tcas	41	133–137	1608	aviation control
schedule	9	291–294	2650	scheduler
schedule2	10	261–263	2710	scheduler
tot_info	23	272–274	1052	statistics
print_tokens	7	341–342	4130	lexical analyzer
print_tokens2	10	350–354	4115	lexical analyzer
replace	32	508–515	5542	pattern matcher

C. Metrics

To measure the fault localization effectiveness, we follow [11] to use the metric *Expense*. For a ranking list produced by a fault localization technique, *Expense* measures how many percent of statements in a program a developer must examine to find the fault. It is computed by the following equation.

$$Expense = \frac{rank\ of\ faulty\ statement}{number\ of\ executable\ statement}$$

An expense value only indicates the effort of an engineer must use to locate the fault based on the fault localization results. However, in practice, a software engineer will only have the patience to walk through a small portion of the ranking list. As a result, a high expense value may be useless for debugging. Thus, we consider a (portion of) test suite can support fault localization ***effectively*** only if the expense value of using (a portion of) a test suite for fault localization is lower than a threshold value. For a test case prioritization technique, we define a metric *Percentage* to measure the percentage of a prioritized test suites generated by a test case prioritization technique that can help a fault localization technique to achieve an expense value lower than a given threshold.

Given a test case prioritization technique ***T*** and a threshold expense value ***e***, we define *Percentage* as:

$$Percentage(T, e) = \frac{\text{no. of test suites generated by } T \text{ with expense value} < e}{\text{total no. of test suites prioritized by } T}$$

D. Experiment Setup

We applied the 16 test case prioritization techniques (see Table I) and the 4 fault-localization techniques (see Table II) to the 122 versions of our subject programs and their test suites. In this section, we present the set up the experiment.

The experiment involves seven Siemens programs obtained from Software Infrastructure Repository. Following [13], we use the branch-adequate test suites provided by SIR to conduct the test case prioritization. There are 1,000 small test suites and 1,000 large test suites. For a small suite, it contains about 30 test cases. We can simply retest all test cases, as the execution time of the entire test suite is trivial. There is no real need to perform test case prioritization. Thus, we choose to use the large test suite for our empirical study, as it would be more meaningful to perform test case prioritization on them.

All the ART techniques are based on random selection. Therefore, we repeat each of them 20 times to obtain an average performance. To reduce the huge computation cost incurred in the experiment, we randomly select 50 suites from the available 1000 test suites for each subject. Thus, we conduct a total 1000 prioritizations for each ART technique.

For each prioritized test suite generated by each test case prioritization technique, we execute its top 10%, 20%, 30%, 40%, …, 100% test cases in turn. For each such portion of individual prioritized test suites, we collect the expense value from the four fault localization techniques, and compute the metric value based on the *Percentage* Metric. As a result, each prioritized test suite will generate 10 test suites of different sizes for fault localization. In this way, we can evaluate how a test case prioritization technique changes in terms of *Percentage* when different portions of the prioritized test suite are used for fault localization.

E. Experiment Environment

We carried out the experiment on a Dell PowerEdge 2950 server serving a Solaris UNIX. The server has two Xeon 5430 (2.66Hz, 4 core) processors with 4GB physical memory.

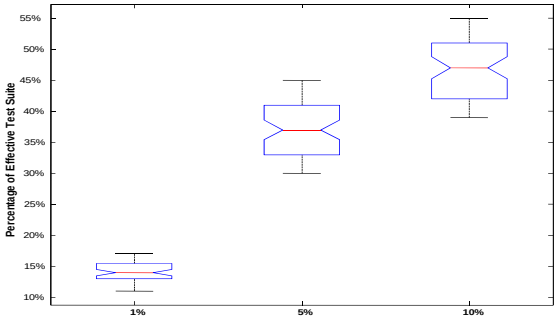


Figure 1. Percentage distribution for all-edge adequate test suite.

IV. Data Analysis

A. Experiment Results

a) Research Question One.

We examine the effect when a fault localization technique uses the entire test suite to locate faults. As a result, we need not to differentiate different test case prioritization techniques, as the test suite generated by them will have the same fault localization results. We calculate the percentage of effective test suites over all combination of such test suites and fault localization techniques for each percentage of code inspection. Then we compute their distribution as shown in Figure 1.

We observe that in general, with an increase in the affordable code inspection range, the percentage of effective test suites increases significantly. More specifically, when examining less than 1% of the code to locate fault, the median, higher quartile and lower quartile value for the percentage of effective test suite are 14%, 15.5% and 13%. These values are low.

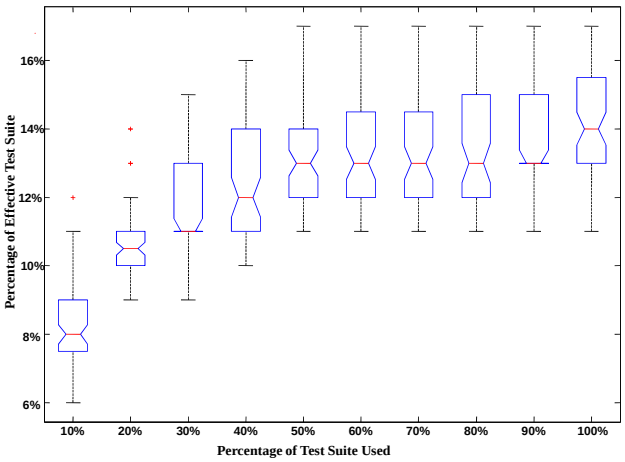
If we relax the code inspection range from 1% to 5% and 10%, the median *Percentage* value will improve to 37% and 47%, respectively, which are still not high. Furthermore, for real-life sized programs with thousands of code, 5% or 10% of code inspection indicates about hundreds of lines of code for inspection, which may be demanding for software engineers.

In summary, we observe from the experiment that on the Siemens suite, using branch-adequate test suites is unlikely to support existing statistical fault localization techniques to locate faults effectively. This observation raise an interesting question for future work: can we define test data adequacy criteria that likely construct test suites for effective fault localization?

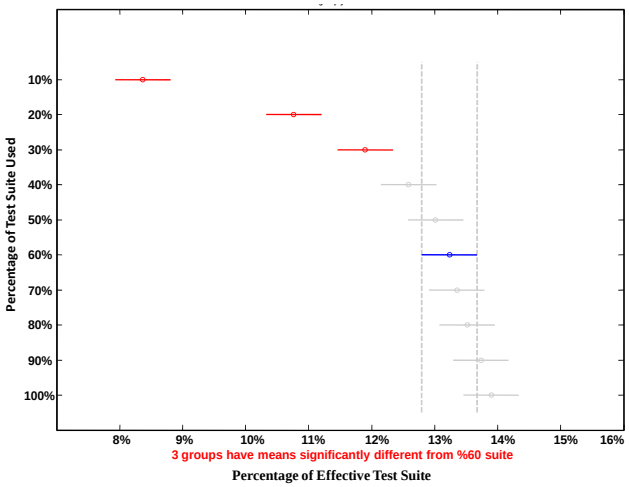
b) Research Question Two. To answer this research question, we perform a postmortem analysis on the integration results. We use three different threshold expense values: 1%, 5%, and 10% as the criteria to deem a test suite to be effective.

Then, given a percentage of test suite used for test case prioritization (e.g. 10% of the suite) and a threshold expense value, we calculate the *percentage* metric value for each test case prioritization technique on each fault localization technique. Instead of studying the *percentage* metric for individual test case prioritization technique case by case, we show the average (mean) distribution of *percentage* values of all the existing test case prioritization techniques and fault localization techniques as shown in Figure 2. In this way, we can get an idea that how likely the *existing* test case prioritization techniques in general produce a test suite for effective fault localization.

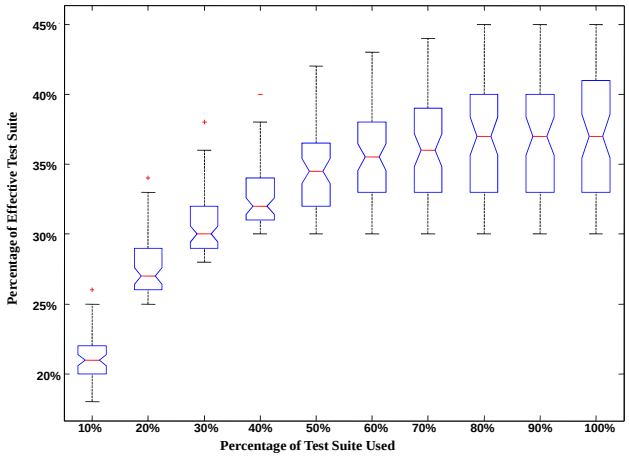
Figures 2 (a), (c), and (e) show the distributions of percentage of effective test suites for the expense threshold values 1%, 5% and 10%, respectively. In Figures 2 (a), (c), and (e), the x-axis shows different percentages of test suite used for fault localization, the y-axis is the *Percentage* value over a test case prioritization technique to locate the faults by examining up to the threshold percentage of code.



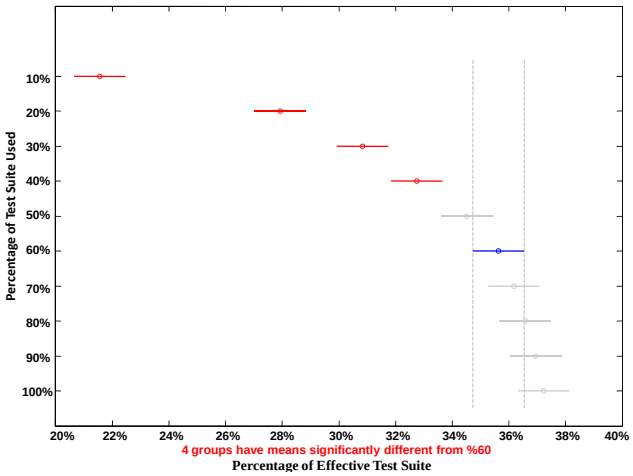
(a) Expense < 1%



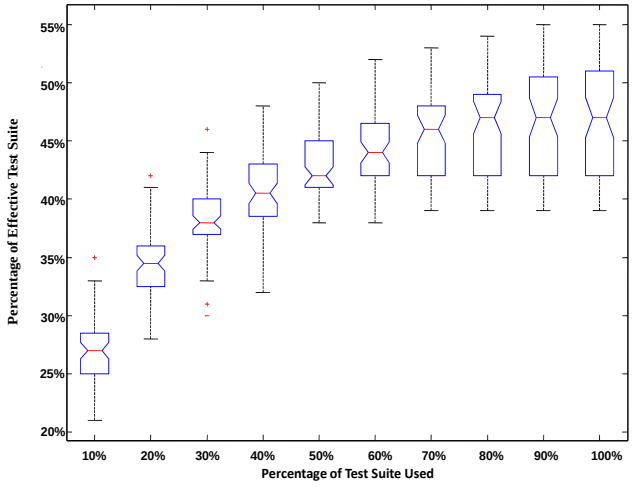
(b) Expense < 1%



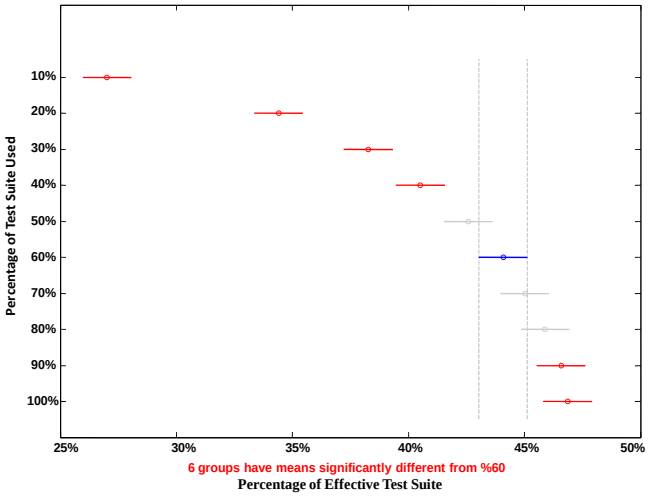
(c) Expense < 5%



(d) Expense < 5%



(e) Expense < 10%



(f) Expense < 10%

Figure 2. The probability of test case prioritization techniques supporting effective fault localization.

From Figure 2 (a), we observe that by inspecting the top 1% of the ranking list of statements, the median of *Percentage* value test suite is 8% if we prioritize and execute the top 10% of test suite for fault localization, which is very low. Even if we increase the percentage of test suite to 100%, the median of the percentage of effective test suites is still less than 14%.

From Figures 2 (a), (c), and (e), the trend is that if a higher percentage of an original test suite is used for fault localization, the percentage of effective test suite increases. However, the increase is gradually less noticeable when the percentage of test suite used is around 60%. In particular, given the code inspection range of 1%, using 60% of the prioritized test cases for the fault localization already achieves a *percentage* value of 13%, whereas using all the remaining 40% of test cases will increase the *percentage* value to 14% only. We observe a similar trend for code inspection ranges 5% and 10% from Figures 2 (c) and (e), respectively.

We further perform ANOVA analysis to compare their mean values. The analysis result consistently rejects the null hypothesis that using different percentages of test suites has the same *Percentage* values.

To see what percentage of test suites different from each other in terms of *Percentage*, we also conduct multiple comparisons to find how different percentages of test suites differ significantly from each other at the 5% significance level [12]. Figures 2 (b), (d), and (f) represent the multiple comparison results by comparing different percentages of test suites with 60% of the test suite. The solid lines not intersected by the two vertical lines represent those percentages of test suites whose means differ significantly from using 60% of the suite for fault localization, while the gray lines represents those percentages of suites comparable to using 60% of suites for fault localization.

From Figure 2 (b), executing 60% of the test suite has no significant difference from executing 40−100% of the test suite. Similarly, from Figure 2 (d), executing 60% of the test suite has no significant difference from executing 50−100% of the test suite, and from Figure 2 (f), 50−80% alike. They indicate that executing 60% of the test suite could be a cost-effective choice to support fault localization.

c) Threats to Validity. In this study, we use the Siemens programs as our subject program. All of them are small-sized programs with single seeded fault.

We choose only C programs in our empirical study as C programming language is still widely used for system application like OS, database, and Web Server application. Also, since the difference between the faulty version and original version of our subject programs are just the seeded faults, this prevent us from measuring the fault localization effectiveness by filtering out those test statements unrelated to faults. But the current expense metric is still meaningful, as the relative ranking of statements will not change even after the filtering process.

In this study, due to time and resource constraint, we only evaluate the random, the coverage-based and the white-box ART-based test case prioritization techniques. Since they are the best general test case prioritization techniques studied in previous work. We use the SIR-provided branch-adequate test suites for our experiment. Using other adequate test suites may lead to different conclusions.

V. Related Work

Apart from the 16 test case prioritization techniques and the 4 fault localization techniques in Section II, There are many studies on integrating different testing and/or debugging techniques. For instance, Wong and colleagues proposed an approach to combining test suite minimization and prioritization to select cases based on the cost per additional coverage [21]. Yu and colleagues examined the effect of test suite reduction on fault localization [22]. Their studies found that test suite reduction does have an impact on the effectiveness of fault localization techniques. However, they neither studied test case prioritizations nor the extent of reductions that may lead to effective fault localizations similar to what we reported in this paper. Jiang et al. [12] examined the integration of test case prioritization and fault localization. They found that test case prioritization has impact on the effectiveness of fault localization techniques and many existing prioritization techniques are no better than random ordering. However, they do not study to what extent test case prioritizations may generate test suites that existing fault localization techniques may use them to locate faults effectively.

There are studies on test case prioritization. Srivastava et al. [20] developed a binary matching technique to calculate the changes in program at the basic block level and prioritize test cases to cover the affected program changes maximally. Li et al. [17] evaluated various search algorithms for test cases prioritization. Leon and colleagues [16] also proposed failure-pursuit sampling techniques. The failure-pursuit sampling uses one-per-cluster sampling to select the initial sample and if a failure is found, its *k* nearest neighbors are selected and checked. If additional failures are found, the process will repeat.

There are also studies on fault localization techniques that are closely related to the four techniques used in our experiment. For instance, Delta Debugging [5] automatically isolates failure-inducing inputs, produces cause-effect chains, and finds the faults. Renieris and Reiss [19] found that the execution trace difference between a failed run and its nearest passed neighbor run is more effective than using other pairs for fault localization. Jeffrey et al. proposed a value-profile based approach to ranking program statements according to their likelihood of being faulty [11]. Zhang et al. [23] proposed to use edge profiles to represent passed executions and failed executions, compare them to model how each basic block contributes to failures and how the infected program states propagate along adjacent basic blocks through control flow edges, and calculate the score for each basic block for ranking.

Baudry et al. [2] used a bacteriologic approach to create test suites that aim at maximizing the number of dynamic basic blocks and use the ranking algorithm in [14].

VI. CONCLUSION

In this paper, we have studied empirically the integration between testing techniques and debugging techniques. We have conducted an experiment on 16 test case prioritization techniques, 4 fault localization techniques over the Siemens suite using branch-adequate test suites. We have analyzed the ratio of such test suites after prioritization can help fault location techniques to locate faults effectively. Our finding shows that many such branch-adequate test suites cannot enable existing fault localization techniques to locate fault effectively. Moreover, using 60% of the prioritized test suites can be a good choice to apply test case prioritization techniques if the entire test case is effective. Future work includes more experimentation and more details analysis.

REFERENCE

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In Proceedings of Mutation *Testing: Academic and Industrial Conference Practice and Research Techniques* (*TAICPART-MUTATION 2007*), pp. 89−98, 2007.

[2] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software engineering* (*ICSE 2006*), pp. 82−91, 2006.

[3] B. Liblit, M. Naik, Z. X. Zheng, A. Aiken, M. I. Jordon. Scalable statistical bug isolation, In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation* (*PLDI 2005*), pp. 15−26, 2005.

[4] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization, In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC/FSE 2005*), pp. 286−295, 2005.

[5] H. Cleve, and A. Zeller. Locating causes of program failures. In P*roceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*), pp. 342−351, 2005.

[6] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration.* pp. 25−45. Addison-Wesley Professional. 2007.

[7] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies, *IEEE Transactions on Software Engineering*, 28 (2): 159−182, 2002.

[8] M. Fowler. Continuous Integration. Available at http://martinfowler.com/articles/continuousIntegration.html. 2006.

[9] D. Farley. The Development Pipeline. Available at http://studios.thoughtworks.com/assets/2007/5/11/The-Deployment-Pipeline-by-Dave-Farley-2007.pdf. 2007.

[10] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (*ICSE 1994*), pp. 191−200, 1994.

[11] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pp. 167− 178, 2008.

[12] B. Jiang, Z. Zhang, T.H. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference* (*COMPSAC 2009*), pp. 99−106, 2009.

[13] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse. Adaptive random test case prioritization. In *Proceedings of the 24rd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2009*), pp. 233−244, 2009.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), pp. 467−477, 2002.

[15] J.-M. Kim, and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), pp. 119-129, 2002.

[16] D. Leon, W. Masri, and A. Podgurski. An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pp 412−421, 2005.

[17] Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization, *IEEE Transactions on Software Engineering*, 33 (4): 225−237, 2007.

[18] NIST. *Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing*, 2002. Available at: www.nist.gov/director/prog-ofc/report02-3.pdf. Last access: 31 Mar 2010.

[19] M. Renieres, and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of 18th International Conference on Automated. Software Engineering* (*ASE 2003*), pp. 30−39, 2003.

[20] A. Srivastava, and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*), pp. 97−106, 2002.

[21] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A Study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering* (*ISSRE 1997*), pp. 230−238, 1997.

[22] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th International Conference on Software Engineering* (ICSE 2008), pp. 201−210, 2008.

[23] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundation of Software Engineering* (*ESEC 2009/FSE-17*), pp. 43−52, 2009.