How Well Do Test Case Prioritization Techniques Support Statistical Fault Localization*

Bo Jiang, Zhenyu Zhang, T.H. Tse†
The University of Hong Kong
Pokfulam, Hong Kong
{bjiang, zyzhang, thtse}@cs.hku.hk

T. Y. Chen
Swinburne University of Technology
Hawthorn, Victoria 3122, Australia
tychen@swin.edu.au

***Abstract*—In continuous integration, a tight integration of test case prioritization techniques and fault-localization techniques *may* both expose failures faster and locate faults more effectively. Statistical fault-localization techniques use the execution information collected during testing to locate faults. Executing a small fraction of a prioritized test suite reduces the cost of testing, and yet the subsequent fault localization may suffer. This paper presents the first empirical study to examine the impact of test case prioritization on the effectiveness of fault localization. Among many interesting empirical results, we find that coverage-based techniques and random ordering can be more effective than distribution-based techniques in supporting statistical fault localization. Furthermore, the integration of random ordering for test case prioritization and statistical fault localization can be effective in locating faults quickly and economically.**

Keywords*—*Continuous integration; software process integration; test case prioritization; fault localization

I. INTRODUCTION

Continuous Integration (*CI*) [11] refers to a software process, in which developers integrate their software artifacts with a *CI agent* frequently, such as many times a day. CI helps shorten the development cycle and lower the development cost [6]. Each integration is known as a *build*, in which code compilation is followed by regression testing. Multiple developers may submit their artifacts to the CI agent at various chosen times, resulting in one or more integrations within each period. During a busy period, multiple developers may concurrently submit their code and every developer expects the CI agent to run the regression test suite of the same baseline version to verify their individual submissions. The overall integration process is thus heavily loaded. It is, therefore, necessary to optimize this activity.

CI is conducted in stages [10][11]. Figure 1 depicts a typical scenario. After a developer has submitted a set of artifacts to a CI agent, the latter first conducts a *commit build*, which runs a fraction of a regression test suite to verify a target application. In case any failure is revealed, the developer may debug the artifacts based on the bug report generated [21]. CI may include the results of fault-localization techniques in the bug reports to assist developers to locate faults and fix them [2][14][23]. The second stage runs the remaining regression test cases to resume the verification of the application, and hence it often takes longer time and more resources than a commit build. This second stage is not executed as frequently as the commit build. As commit builds are frequently invoked, regression testing has been reported to be a major bottleneck [11].

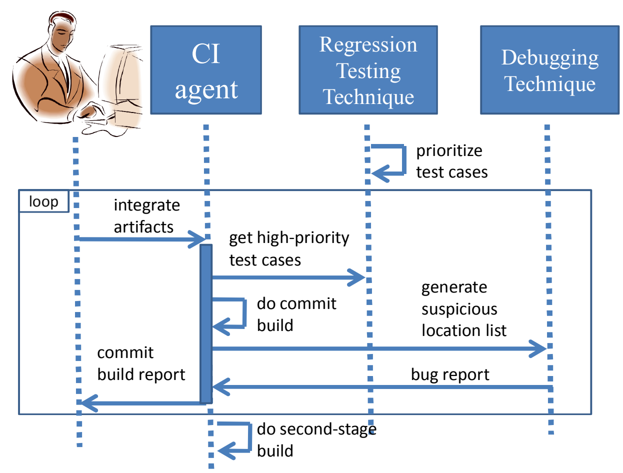

Figure 1: A scenario of continuous integration.

On one hand, the time available for regression testing in a commit build may be limited. Using a smaller number of test cases will reduce the time required to complete the build and let the concerned developers receive the feedback quickly. It would be attractive to select test cases that can, for example, detect failures faster. This is generally known as the test case prioritization problem [8][17][20][24][25].

On the other hand, developers would like as much information as possible to debug the artifacts in CI. For instance, to assist debugging, statistical fault-localization techniques may require the code coverage information of a variety of test cases to assess the suspiciousness of program statements [1][16][21][22].

* This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 716507 and 717308) and a Discovery Grant of the Australian Research Council.

† All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2193. Email: thtse@cs.hku.hk.

Thus, we face a dilemma. Using a smaller high-priority test suite for a commit build helps shorten the response time of a CI agent in each micro-cycle (such as a single loop in Figure 1). However, such a test suite may carry less information to facilitate fault-localization techniques to iron out the root causes of the failures. It may make fault localization less effective, and hence lengthen the next micro-cycle. Figure 2, for instance, shows a scenario of three test cases (t1, t2, and t3) ordered by two test case prioritization techniques, where different fractions of the prioritized test suite are fed to a statistical fault-localization technique to find the fault in statement s3. A smaller *Expense* in the figure indicates less effort to conduct fault localization (marked as debugging in the figure). Using an appropriate test case prioritization technique (such as random ordering), we can use fewer test cases (2 in this illustration) to locate faults, while the value of *Expense* (66) is not much worse than the case of using the entire test suite (50).

	Test Case					
Statement	t1	t2	t3			
s1	•					
s2	•	•		less	testing effort	more
s3 (faulty)		•	•	more	debugging effort	less
s4	•		•	Size of the test suite used		
Test Outcome	✓	✓	×	1	2	3
Total Stmt (TS)	1st	2nd	3rd	100	100	50
Random (R)	3rd	1st	2nd	100	66	50

Test case prioritization technique (see Section II-A)

Expense (see Section III-E) achieved by Tarantula (see Section II-B)

Figure 2: Testing/debugging dilemma.

How well does a small high-priority test suite support (statistical) fault localization? Is there any efficient strategy to generate effective test suites? Knowing the answers to these questions is critical toward a tight integration of development activities.

This paper conducts an empirical study to examine these important questions. The study employs nine representative test case prioritization techniques, four statistical fault-localization techniques, and seven popular subject programs. It broadly studies the extent to which test case prioritization techniques affect the effectiveness of fault-localization techniques in multiple dimensions, including granularity, prioritization strategy, and percentage of prioritized test suites used.

The main contribution of the paper is threefold. (i) We report the first empirical study to evaluate the impact of test case prioritization techniques on statistical fault-localization techniques. (ii) The empirical results interestingly show that the coverage-based test case prioritization strategy is less sensitive than the other studied strategies in supporting such integration. It also shows that the additional-statement (AS) technique is the least sensitive to changes in the percentage of prioritized test suits applied to statistical fault-localization techniques. (iii) Surprisingly, random ordering of test cases, which is less sensitive than the distribution-based techniques in the study, is found to be a good candidate. The low-cost and objective nature of random prioritization (despite its relatively low effectiveness in the speed to detect regression faults) shows that black-box regression testing techniques can be promising to integrate with fault-localization techniques.

This paper will be organized as follows: Section II revisits selected test case prioritization techniques and fault-localization techniques. Section III describes the empirical study, followed by its results in Section IV. Section V reviews related work. We conclude the paper in Section VI.

II. Techniques Revisited

This section describes the test case prioritization techniques and fault-localization techniques to be used in our empirical study.

A. Test Case Prioritization Techniques

We study two dimensions in test case prioritization techniques. The first is *granularity*. We follow [8] to use statement coverage to represent a finer granularity and use functional coverage to represent a coarser granularity. The second dimension is the *prioritization strategy*. We study *coverage-based* techniques and *distribution-based* techniques in this dimension. The coverage-based techniques are greedy algorithms [5][7][8][25], which can be further subdivided into the *total* and the *additional* strategies. For distribution-based techniques, we study those proposed in [3][18][19]. Furthermore, coverage information on each test case is obtained from the test execution on the previous (baseline) version of the program.

1) Coverage-based techniques.

The total statement (TS) test case prioritization technique sorts test cases in descending order of the total number of statements covered by each test case in the previous version. In case of a tie, it randomly orders the test cases involved. The total function (TF) test case prioritization technique is the same as TS, except that it uses function coverage information instead of statement coverage information.

The additional statement (AS) test case prioritization technique is like TS, except that it selects the test case that covers the maximum number of statements not yet covered in each round. When no remaining test case can further improve the statement coverage, the technique will reset all the statements to "not covered" and reapply AS on the remaining test cases. When more than one test case covers the same number of statements not yet covered, it just picks one such test case randomly. The additional function (AF) test case prioritization technique is the same as AS, except that it uses function coverage information instead of statement coverage information.

2) *Distribution-based techniques.*

Leon et al. [18] propose distribution-based techniques for test case filtering, which prioritize test cases based on the distribution of their execution profiles via dissimilarity metrics [3][18]. The dissimilarity metrics define the distances between pairs of test cases. We use two dissimilarity metrics, namely the *count metric* and the *proportional binary metric* [3][18][19]. We strictly follow [18][19] to use the hierarchical agglomerative clustering algorithm with one-per-cluster sampling [12] in our empirical study.

Suppose a program consists of *m* statements. Each test case is represented by an *m*-dimensional vector. Each element in the vector holds the execution count of every statement.

The *count metric* between a pair of test cases is the Euclidean distance between two *m*-dimensional vectors.

The *proportional binary metric* (*pbm*) is a modified Euclidean distance formula. It aims to balance between coverage information and distribution information [19]. Let us define $C_{i,j}$ as the number of times that statement *j* has been exercised by test case *i* (represented by the *j-th* element of the vector for test case *i*). Suppose we have k test cases in total. We further define $\min_k\{C_{i,j}\}$ as the minimum $C_{i,j}$ among the *k* test cases, and $\max_k\{C_{i,j}\}$ as the maximum of $C_{i,j}$ among the test cases. Following [19], we define the distance between two test cases *u* and *v* as

$$D_{u,v} = \sqrt{\sum_k (P_{u,k} - P_{v,k})^2 + |B_{u,k} - B_{v,k}|}$$

where $u, v = 1, 2, \dots, k$, $P_{i,j} = \frac{C_{i,j} - min_k\{C_{k,j}\}}{max_k\{C_{k,j}\} - min_k\{C_{k,j}\}}$, and $B_{i,j} = \begin{cases} 0 & if\ P_{i,j} = 0 \\ 1 & otherwise \end{cases}$, which models whether the execution of statement *i* covers statement *j*.

We use the clustering algorithm with *count* metric at *statement* level (CS) to illustrate how to apply the distribution-based test case prioritization techniques. (i) For each test case in a test suite, we create a vector containing the execution count for every statement in the program. (ii) Using the count metric, we compute a distance matrix containing the distances (dissimilarity values) between pairs of test cases. (iii) We strictly follow [3][18][19] to use 1, 2.5, 5, 10, 15, 25, and 30 percents of a test suite as cluster count parameters. Using the hierarchical agglomerative clustering algorithm, we merge the nearest two test cases in each step until we obtain the required cluster count. (iv) Following one-per-cluster sampling, we randomly select one test case from each cluster every time. We repeat this selection process until all test cases have been selected.

The clustering algorithm with *count* metric at *function* level (CF) is the same as CS, except that it uses the statistical counts of function executions rather than those of statement executions. The clustering algorithm with the *proportional binary* metric at *statement* level (PBS) is the same as CS, except that it uses the proportional binary metric. The clustering algorithm with the *proportional binary* metric at *function* level (PBF) is the same as PBS, except that it uses the statistical counts of function executions rather than those of statement executions.

We also compare the above techniques with the random test case prioritization "technique" [8]. We summarize the properties of all the nine techniques in Table 1.

TABLE 1. PRIORITIZATION TECHNIQUES

Acronym	Strategy Category				Granularity	
	Coverage		Distribution		Statement	Function
	Total	Additional	Count Metric	pbm		
R	random					
TS	✓				✓	
AS		✓			✓	
TF	✓					✓
AF		✓				✓
CS			✓		✓	
PBS				✓	✓	
CF			✓			✓
PBF				✓		✓

B. Fault-Localization Techniques

Researchers have proposed several techniques to help developers locate faults. We revisit four such techniques used in our empirical study.

1) *Tarantula.*

Jones et al. [16] propose the Tarantula technique, which was used initially for the visualization of testing information. To rank program statements, Tarantula computes two metrics, *suspiciousness* and *confidence*, according to the coverage information on passed and failed test cases.

The suspiciousness of a statement *s* is given by

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

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. The suspiciousness is 0 when statement *s* is least suspicious, or 1 when *s* is most suspicious.

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.

2) *Statistical Bug Isolation (SBI)*

Liblit et al. [21] propose Statistical Bug Isolation (SBI) for computing the suspiciousness of a predicate *P* in a program, thus:

$$Failure(P) = \frac{failed(P)}{passed(P) + failed(P)}$$

The function *failed* (*passed*, respectively) tallies the number of test cases for which *P* is evaluated to be false (true).

For ease of comparison with other fault-localization techniques, Yu et al. [29] adapt the equation to calculate 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 evaluated to be false (true).

3) Jaccard.

Abreu et al. [1] propose a Jaccard metric as the suspiciousness formula instead of that in Tarantula. 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 number of failed test cases in the test suite. The technique ranks the statements similarly to Tarantula.

4) Ochiai.

Abreu et al. [1] also propose to use the Ochiai metric as another suspiciousness formula. The equation for Ochiai (from [1]) is given by

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

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

III. Empirical Study

A. Research Questions

The empirical study addresses the following research questions:

RQ1: To what extent will a fault-localization technique be affected if it only uses a fraction of a prioritized test suite as input?

RQ2: When reordering test suites with a view to faster localization of faults, are there any particularly outstanding strategies or granularities for test case prioritization?

RQ3: Can random test case prioritization outperform other prioritization techniques for faster localization of faults?

RQ1 studies whether commit builds may help fault localization effectively. If RQ1 indicates that test case prioritization may help, RQ2 answers whether there are test case prioritization strategies that are particularly attractive or unattractive for continuous integration. RQ3 studies whether random ordering, commonly considered to be ineffective for prioritizing test cases, may be a good technique to help developers locate faults in programs.

TABLE 2. Subject programs

Subject	Faulty Versions	LOC	Test Pool Size
tcas	41	133–137	1608
schedule	9	291–294	2650
schedule2	10	261–263	2710
tot_info	23	272–274	1052
print_tokens	7	341–342	4130
print_tokens2	10	350–354	4115
replace	32	508–515	5542

B. Subject Programs and Test Pools

We used the *Siemens* programs as the subjects for the empirical study. We obtained them from the Software-artifact Infrastructure Repository (SIR) [4] available at http://sir.unl.edu (last accessed in April 2009). Table 2 shows the descriptive statistics of the subject programs. The *Faulty Versions* column lists the number of faulty versions for each subject program. The column *LOC* shows the variations in the numbers of executable lines of code for the faulty versions of each program. The column *Test Pool Size* represents the total number of available test cases in the test pool for each program.

Following previous work [5][8], we excluded those versions whose faults cannot be revealed by any test case. We followed [9] to remove the versions whose faults are too evident (such that more than 25% of the test cases in the pool can detect them). We use the standard coverage tool gcov in conjunction with gcc to collect coverage information of program executions, and hence we also excluded those versions that gcov cannot handle owing to segmentation faults. Finally, we used all the remaining 121 faulty versions in our data analysis.

C. Experimental Setup

This section presents the experiment setup for the empirical study.

For each faulty version, we selected test cases randomly from the test pools provided by SIR to eliminate any bias due to a particular test case generation strategies. More specifically, we repeatedly selected one random test case at a time from a given test pool (without replacement and without considering its test outcome) until we obtained the desired number (n) of test cases in a test suite. We chose n = 50, 100, 200, 300, 400, and 500. We repeated this test suite construction procedure 100 times. In short, we created 600 test suites per faulty version. We then applied each of the nine test case prioritization techniques to prioritize every test suite. For each prioritized test suite, we took the top-most percentage (m%) of the test cases and input them to each of the fault-localization techniques. We chose m = 10, 30, 50, 70, 90, and 100.

D. *Experimental Environment*

We carried out the empirical study on a Dell PowerEdge 2950 server run under Solaris UNIX. The server has 2 Xeon 5430 (2.66 Hz, 4 core) processors with 4 GBytes of physical memory.

E. *Metrics*

To measure the effectiveness of fault localization, we follow [13][15][29] to use the *Expense* metric. For a ranked list produced by a fault-localization technique, *Expense* measures the percentage of statements in a program that must be examined to locate the fault (the lower, the better). We adopt the following definition proposed by [13][15][29]:

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

We further use the notation *Expense*(m) to represent the percentage of statements examined when only the topmost m percent of a given test suite is used. For instance, *Expense*(100) refers to the percentage of statements examined when the entire test suite is used for fault localization.

Since we are interested in how test suites of different sizes may affect the values of *Expense* in a fault-localization technique, we further define a *Relative Expense* metric as follows:

$$Relative\ Expense(m) = \frac{Expense(m) - Expense(100)}{Expense(100)}$$

IV. Data Analysis and Discussions

A. *Empirical Results*

In this section, we first present the raw results, and then analyze them to answer the research questions RQ1, RQ2, and RQ3.

Figure 3 depicts the respective mean *Expense* of the four fault-localization techniques when they use different percentages of prioritized test suites to locate faults. There are nine points in each plot, representing, from left to right, the prioritization techniques CF, PBF, CS, PBS, AF, AS, R, TF, and TS. There are six plots in each row, representing, from left to right, the results when 10, 30, 50, 70, 90, and 100 percents of the prioritized test suite is used. For instance, the leftmost point in the leftmost plot of the Tarantula row represents the mean *Expense* of Tarantula using the first 10% of the prioritized test suite generated by CF to locate a fault.

1) Answering RQ1: To what extent will a fault-localization technique be affected if it only uses a fraction of a prioritized test suite as input?

We first observe that, across the plots in the same column, the corresponding points are quite close to one another in terms of *Expense*. It indicates that different fault-localization techniques may be affected to a similar extent. We have applied ANalysis Of VAriance (ANOVA) hypothesis testing to confirm this observation. The results also show that there is no significant difference.[1] For simplicity of presentation, therefore, we will only discuss the typical empirical results acrosss techniques for research question RQ1, unless a particular technique warrants specific highlights.

Across the plots in different columns with the same technique, the changes in *Expense* at the corresponding points are very noticeable. For instance, the leftmost point of the Tarantula row is 0.705, which is very different from the corresponding points on the other five plots, namely (from left) 0.487, 0.366, 0.323, 0.294, and 0.274.

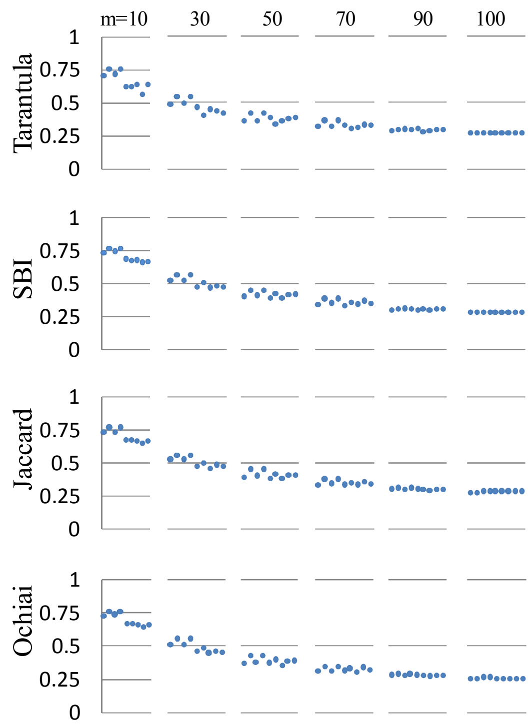

Figure 3. *Mean* effectiveness of fault-localization techniques on the use of fractions of the test suites produced by test case prioritization techniques.

We also observe from these plots that, even when half of a test suite prioritized by AS is used for commit build, the effectiveness of fault localization will not deteriorate much. The empirical results of Tarantula show that, for the AS prioritization techniques, developers only need to examine 7% more code to locate the fault when using the topmost 50% of a test suite (as compared with the use of the whole test suite).

In other scenarios such as the PBF points of the Jaccard plots, however, developers would need to examine 15% more code to locate the fault when using the topmost 50% of a test suite. A further reduction of the test suite may cause the developer to examine significantly more code. For instance, the developer would need to examine 10% more code if another 20% of test suite is not used in a commit

[1] Owing to space limitation, we omit the ANOVA results in this paper.

build. The empirical results also show that the differences in effectiveness according to the use of different fractions of test suites are generally significant.

In summary, with respect to research question RQ1, we find that fault-localization techniques exhibit better effectiveness when they use only a fraction of the prioritized test suite. In other words, such techniques can indeed prioritize test cases to help fault localization.

2) Answering RQ2: When reordering test suites with a view to faster localization of faults, are there any particularly outstanding strategies or granularities for test case prioritization?

To study the overall effect of test case prioritization on fault localization using different percentages of a test suite, we further analyze the entire dataset for RQ1 via the mean *Relative Expense* of the four fault-localization techniques. For instance, for the AS row, we sum up (0.625–0.274) / 0.274 from the Tarantula plots, (0.672–0.288) / 0.288 from the SBI plots, (0.668–0.281) / 0.281 from the Jaccard plots, and (0.662–0.261) / 0.261 from the Ochiai plots. We then divide the sum by four. Other means can be computed in the same manner. Table 3 shows the results.

TABLE 3. MEAN *RELATIVE EXPENSE*

	m	10	30	50	70	90	100
Distribution-based	CF	164%	86%	39%	20%	7%	0%
	PBF	**177%**	**102%**	**59%**	**33%**	**9%**	**0%**
	CS	164%	86%	41%	21%	7%	0%
	PBS	**176%**	**102%**	**59%**	**33%**	**9%**	**0%**
Random	R	140%	70%	39%	19%	7%	0%
Coverage-based	AF	140%	73%	43%	22%	7%	0%
	AS	**139%**	**64%**	**36%**	**17%**	**5%**	**0%**
	TF	128%	70%	45%	26%	8%	0%
	TS	139%	66%	45%	22%	8%	0%

We observe from Table 3 that, irrespective of the granularity level or the prioritization strategy, distribution-based techniques have been affected more adversely than coverage-based techniques. Furthermore, the results of using proportional binary metrics (PBF and PBS) are the worst in terms of their effects on *Expense* (or the percentage of statements examined). It indicates that the fault-localization techniques are most adversely affected by the sizes of the prioritized test suites when the ordering is conducted via PBF or PBS. On the other hand, fault-localization techniques are generally least affected when the test suites are generated by AS.

In summary, with respect to research question RQ2, we find the additional statement (AS) technique to be consistently outstanding in the effectiveness of fault localization and the early detection of faults. We are going to further study AS in Section 3) below.

3) Answering RQ3: Can random test case prioritization outperform other prioritization techniques for faster localization of faults?

To study random test case prioritization, we further compute the ratio of a cell in a column of Table 3 to the cell for *R* in the same column. The resultant value indicates how a prioritization technique makes an impact on the changes in *Relative Expense* for a fault-localization technique (as compared with random ordering). Table 4 shows the results. Informally, a value above 1 (below 1, respectively) in a cell means that, using only a fraction of the prioritized test cases, the technique is more (less) adversely affected than random ordering.

TABLE 4. MEAN EFFECTIVENESS RELATIVE TO RANDOM ORDERING

	m	10	30	50	70	90	100
Distribution-based	CF	1.168	1.222	0.986	1.025	0.913	1.000
	PBF	1.258	1.450	1.502	1.715	1.252	1.000
	CS	1.172	1.232	1.029	1.073	0.947	1.000
	PBS	1.255	1.449	1.501	1.713	1.250	1.000
Random	R	1.000	1.000	1.000	1.000	1.000	1.000
Coverage-based	AF	0.997	1.043	1.084	1.126	0.887	1.000
	AS	0.994	0.911	0.908	0.907	0.624	1.000
	TF	0.913	0.996	1.129	1.326	1.048	1.000
	TS	0.990	0.947	1.133	1.149	1.077	1.000

Interestingly, irrespective of the granularity level or the prioritization strategy, every distribution-based technique is worse than random ordering. This result further strengthens the finding in Section 2) that test suites generated by distribution-based test case prioritization techniques may not integrate well with statistical fault-localization techniques.

In particular, the only examined technique that can consistently outperform random prioritization is AS. Combined with the result in Section 2), we find that AS can be promising in providing effective test case prioritization as well as supporting statistical fault localization.

Furthermore, random prioritization even outperforms AF at times (see the highlighted cells of the AF rows in Table 4). It indicates that its granularity level (even for the additional test case prioritization strategy) is insufficient for generating prioritized test suites that assist developers to debug programs better than random ordering. The total test case prioritization strategy also suffers from a similar problem.

In summary, with respect to research question RQ3, we find that random prioritization is a cost effective technique for fault localization. It can perform as good as (if not better than) all other prioritization techniques except the AS technique.

B. *Threats to Validity*

We use the Siemens programs as subjects in the study. All of them are small-sized programs with seeded faults. Further empirical studies on larger programs with multiple faults may further strengthen the external validity of our findings. In practice, although redundancy in test suites used in practice may exist, not all test suites may fully satisfy certain testing criteria. To address this issue, we use test suites that are randomly constructed. Still, random test suites from the test pool are limited by the contents of the original pool. However, a study of whether the test pool of the Siemens suite represents a realistic setting is beyond the scope of this paper.

We only choose C programs in our empirical study. A further investigation of subject programs written in other programming languages may help generalize our findings.

Another threat to validity is the correctness of our experimentation tools. We have measured the Average Percentage of Faults Detected (APFD) of the prioritization techniques, and find that our APFD values are almost the same as these published in the literature such as [8][19]. We believe the minor difference is due to the choice of different test suites between the empirical study reported in this paper and those in [8][19].

We use *Expense* as a metric in our empirical study. *Expense* may indicate a conservative way of locating faults. In practice, when developers examine a particular statement s1, they may spot problems in another statement s2 close to s1. Statement s2 may have a much lower rank. This indicates that a less amount of code can be examined to locate faults than what *Expense* may indicate.

Coverage-based techniques are a kind of greedy approach that selects test cases based on the coverage information on a previous version of the program. On the other hand, the execution profile of a test suite may sometimes change drastically across two different versions of the same program. Thus, the result that AS is more effective than other techniques may not necessarily be generalized. It would be interesting to find the characteristics of changes that would favor (or disfavor) the application of coverage-based techniques.

V. Related Work

This section reviews related work that has not been discussed in previous sections.

Wong et al. [28] proposed an approach that combines test suite minimization and prioritization to select cases according to the cost per additional coverage. Srivastava et al. [26] developed a binary matching technique to calculate the changes in program at the basic block level and prioritize test cases to optimally cover the affected program changes. Walcott et al. [27] studied a time-aware test suite prioritization technique based on genetic algorithms to order test cases under testing time constraints. Li et al. [20] evaluated various search algorithms for test case prioritization.

Apart from distribution-based techniques, Leon and colleagues [3][18][19] also proposed a family of failure-pursuit sampling techniques. They select one initial sample per cluster and, when a failure is found, *k* nearest neighbors are selected and checked. If additional failures are found, the process will be repeated. We do not evaluate this technique because it requires the outcomes of test cases on the modified version and is, therefore, not suitable for continuous integration, in which fast turnaround is required.

Yu et al. [29] conduct an empirical study of the effect of test-suite reduction on fault-localization techniques. They find that the effectiveness of fault localization varies according to different test-suite reduction strategies. The focus of the study, however, is mainly on test-suite reduction from the viewpoint of test-suite composition.

VI. Conclusion

In continuous integration, the total time allowed for testing and fault localization is limited. Thus, it is desirable to use both test case prioritization and fault localization to help developers detect and locate the faults. In this paper, we conduct an empirical study to explore the impact of test case prioritization on statistical fault localization. We find that test suites prioritized by coverage-based strategies are better than those from other strategies in terms of the effectiveness of fault localization. Although random ordering can be less effective than the additional statement technique, no other technique can outperform random ordering. In particular, random prioritization is even better than distribution-based techniques in terms of *Relative Expense*. Our result provides a strong piece of evidence to clear the misconception on random prioritization — random ordering can indeed be effective in supporting such integration.

In the future, we would like to examine the underlying reasons why random prioritization is better than distribution-based techniques, with a view to further developing better variants of the random strategy. For instance, it will be interesting to study the effectiveness of applying adaptive randomness (in the sense of adaptive random testing) for test case prioritization in CI. We also wish to study how to achieve a tighter integration between regression testing and debugging techniques.

Acknowledgment

We are most grateful to Dr. W. K. Chan of City University of Hong Kong for his excellent inputs to the paper.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference: Practice*

And Research Techniques (*TAICPART-MUTATION 2007*), pages 89–98. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pages 342–351. ACM Press, New York, NY, 2005.

[3] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundation of Software Engineering (ESEC 2001/ FSE-9), pages 246–255. ACM Press, New York, NY, 2001.

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

[5] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: an empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11: 33–70, 2006.

[6] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison Wesley, Upper Saddle River, NJ, 2007.

[7] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, 25 (5): 102–112, 2000.

[8] S. G. 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.

[9] S. G. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12 (3): 185–210, 2004.

[10] D. Farley. The Development Pipeline. http://studios.thoughtworks.com/assets/2007/5/11/The-Deployment-Pipeline-by-Dave-Farley-2007.pdf. Last accessed 2008.

[11] M. Fowler. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html. Last accessed 2008.

[12] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Upper Saddle River, NJ, 1988.

[13] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2008*), pages 167–178. ACM Press, New York, NY, 2008.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2005*), pages 273–282. ACM Press, New York, NY, 2005.

[15] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 16–26. ACM Press, New York, NY, 2007.

[16] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), pages 467–477. ACM Press, New York, NY, 2002.

[17] 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*), pages 119–129. ACM Press, New York, NY, 2002.

[18] 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*), pages 412–421. ACM Press, New York, NY, 2005.

[19] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *Proceedings of the 14th International Symposium on Software Reliability Engineering* (*ISSRE 2003*), pages 442–453. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[20] 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.

[21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2005*), pages 15–26. ACM Press, New York, NY, 2005.

[22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundation of Software Engineering (ESEC 2005/FSE-13), pages 286–295. ACM Press, New York, NY, 2005.

[23] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 30–39. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[24] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings of the 15th IEEE International Conference on Software Maintenance* (*ICSM '99*), pages 179–188. IEEE Computer Society Press, Los Alamitos, CA, 1999.

[25] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27 (10): 929–948, 2001.

[26] 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*), pages 97–106. ACM Press, New York, NY, 2002.

[27] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. TimeAware test suite prioritization. In *Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2006*), pages 1–12. ACM Press, New York, NY, 2006.

[28] 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 '97*), pages 264–274. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[29] 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*), pages 201–210. ACM Press, New York, NY, 2008.