Postprint of article in *Information and Software Technology* **54** (7): 739-758 (2012)

How well does test case prioritization integrate with statistical fault localization?*

Bo Jiang[a], Zhenyu Zhang[b], W.K. Chan[c,†], T.H. Tse[d], Tsong Yueh Chen[e]

[a] School of Computer Science and Engineering, Beihang University, Beijing, China
[b] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[c] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong
[d] Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
[e] Centre for Software Analysis and Testing, Swinburne University of Technology, Melbourne, Australia

Context: Effective test case prioritization shortens the time to detect failures, and yet the use of fewer test cases may compromise the effectiveness of subsequent fault localization.

Objective: The paper aims at finding whether several previously identified effectiveness factors of test case prioritization techniques, namely strategy, coverage granularity, and time cost, have observable consequences on the effectiveness of statistical fault localization techniques.

Method: This paper uses a controlled experiment to examine these factors. The experiment includes 16 test case prioritization techniques and four statistical fault localization techniques using the Siemens suite of programs as well as grep, gzip, zed, and flex as subjects. The experiment studies the effects of the percentage of code examined to locate faults from these benchmark subjects after a given number of failures have been observed.

Results: We find that if testers have a budgetary concern on the number of test cases for regression testing, the use of test case prioritization can save up to 40% of test case executions for commit builds without significantly affecting the effectiveness of fault localization. A statistical fault localization technique using a smaller fraction of a prioritized test suite is found to compromise its effectiveness seriously. Despite the presence of some variations, the inclusion of more failed test cases will generally improve the fault localization effectiveness during the integration process. Interestingly, during the variation periods, adding more failed test cases actually deteriorates the fault localization effectiveness. In terms of strategies, Random is found to be the most effective, followed by the ART and Additional strategies, while the Total strategy is the least effective. We do not observe sufficient empirical evidence to conclude that using different coverage granularity levels have different overall effects.

Conclusion: The paper empirically identifies that strategy and time-cost of test case prioritization techniques are key factors affecting the effectiveness of statistical fault localization, while coverage granularity is not a significant factor. It also identifies a mid-range deterioration in fault localization effectiveness when adding more test cases to facilitate debugging.

Keywords: *Software process integration; continuous integration; test case prioritization; statistical fault localization; adaptive random testing; coverage*

* © 2012 Elsevier Inc. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/ republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from Elsevier Inc.

† Corresponding author. Tel.: +852 3442 9684. E-mail address: wkchan@cityu.edu.hk (W.K. Chan).

1. Introduction

Continuous Integration (CI) [15][18][19] is a software integration strategy, in which a team of developers integrates a set of software artifacts frequently and regularly, such as every two hours on a business day. Typically, each developer makes several changes to the codebase to add new features or enhance existing ones, fix reported or latent bugs, or improve non-functional properties of an application. Owing to the short time between two consecutive rounds of software integration involved, each change to the codebase is typically small in relation to the entire codebase. Effective and efficient regression testing techniques that aim at reassuring previously working features of the software application is particularly attractive.

In general, a round of CI integration consists of code compilation, linking, testing, and deployment [18][19]. A developer checks out a module of a baseline version of an application and the baseline version itself from a project repository in modifiable and read-only modes, respectively. The developer then modifies the *local* copy of the module, builds and tests the local copy, updates the local baseline version with the modified module, and rebuilds and tests the updated baseline version. If the quality assurance on the updated local baseline version is passed, the developer commits the change to the module to the CI server. The CI server then compiles and links the committed working copy of the module with the baseline version kept by the CI server into an executable version, performs a first-stage and fast build to verify the key functionality of the application, avoids regression bugs, and returns a build report to the developer. We refer to such a build as a *commit build* [19]. If a commit build is successful, the corresponding committed modifications are merged with the baseline code in the CI server [18]. Moreover, for each round of CI integration, multiple developers may commit their code changes to the CI server.

Every developer expects the CI server to run a regression test suite of a baseline version to verify that his or her own code commit has not unintentionally broken the code in other parts of the executable version, and would like the CI server to provide the developer with a quick feedback. Based on the feedback, the developer may, for instance, either fix any reported problem or proceed to make other modifications to the code as required by the software project.

Executing the whole regression test suite may slow down the feedback cycle to individuals. Our first-hand experience [24] on testing Microsoft protocols shows that executing the entire regression test suite for one protocol testing project can take more than a whole day. The continuous integration process stresses on fast feedback to the developers upon their code commit. However, if the regression testing activity during the continuous integration process takes too much time, the effectiveness of the whole CI process will degrade seriously. Thus, it is critical to improve the efficiency of each activity during the CI process.

As described above, CI can be conducted in stages [18][19]. Fig. 1 depicts such a scenario. In the figure, after a developer has submitted a code module to a CI server, the server first conducts a commit build, which runs a fraction of a regression test suite to verify a target modified baseline version. In case any failure is revealed, the developer may debug the module based on the bug report generated from the limited number of test cases [28].

We observe that the CI server may include the results of fault localization techniques in the generated bug reports to assist the developers to locate faults [13][27][32][36]. After passing the data to the developers, the follow-up stage of the CI may start, provided that the modified baseline versions have successfully passed the test in the commit-build stage. For instance, the second-stage build typically involves executing more time-consuming tests such as those interacting with databases or networks [19].

In other words, in the above CI scenario, a commit build serves not only as a quick check of the integration but also as a guard that decides whether to invoke the second stage. Our work is motivated by several implications from this requirement:

The time available for regression testing in a commit build is always limited [18][19]. As a result, the goal of regression testing during the commit build is to make full use of the allocated time to perform testing until the deadline is reached. Simply using test suite reduction to reduce the number of test cases to be executed is an inflexible approach as it is almost impossible to make the execution time of a reduced test suite to be exactly the same as the time budget. Test case prioritization [16][37][46] is a preferential solution to this problem. Since test case prioritiza-tion reorders test cases so as to execute those with the higher priority first, the test resources will be spent on the execution of the most important test cases before the deadline irrespective of the time budget allocated. Hence, it is

crucial in CI to reorder a test suite with test case prioritization techniques to assign higher priority to those test cases estimated to have higher chances in detecting failures.

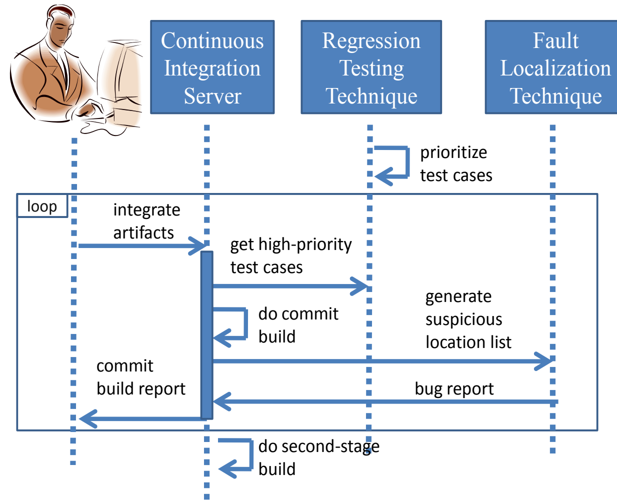

Fig. 1. A scenario of continuous integration.

Given a program *P* and a test suite *T*, a general test case prioritization [16][37] reorders test cases in *T* with the goal of finding a test case ordering that will be useful over a sequence of subsequent modified versions of *P*. General test case prioritization has the advantage of being not on the critical path of software development process. This is because a CI system can conduct general test case prioritization when the developers are modifying the baseline version. In this way, the entire period of regression testing of each integration interval (e.g., two hours in the running scenario) can be used for the actual execution of test cases. The second part of the requirement further demands to cut a reordered test suite into two consecutive fragments so that the higher priority fragment has the ability indistinguishable from the entire test suite to assist developers for their subsequent activities.

One of such activities is program debugging, in which developers would like to obtain as much relevant data as possible to debug the artifacts committed to the CI server. For instance, statistical fault localization techniques in debugging may acquire the code coverage spectrum information of a variety of test cases to assess the suspiciousness of program elements [27][32][44]. Such techniques help developers locate faults by producing a list of ranked statements in descending order of their suspiciousness of being faulty or related to faults [22][27]. Developers may then walk through the ranked list of statements to find the faults.

Using a smaller high-priority test suite for a commit build helps shorten the response time of a CI server in each round of system integration (such as a single loop in Fig. 1). However, such a test suite may carry less information for fault localization techniques to iron out the root causes of detected failures. It may adversely affect fault localization and lengthen the time to develop or maintain a module. Fig. 2, for instance, shows a scenario of three test cases (*t*1, *t*2, and *t*3) 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 *s*3. A smaller expense[1] 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 may use fewer test cases (2 in this illustration) to locate faults, while the expense (0.66) is not much worse than that when the entire test suite is used (0.5).

Many previous studies on test case prioritization and fault localization focused on individual problems separately. More recently, researchers proposed debugging-guided test case prioritization techniques [21]. An inadequate amount of research work (including our preliminary version [26]) studied to what extent existing test case

[1] *Expense* [22][27][44] is a commonly used metric to measure the (in)effectiveness of fault localization in software engineering experiments. It is computed by dividing the number of statements needed to be examined to find a specific fault by the total number of executable statements in the program. Intuitively, a technique having a smaller expense for locating a particular fault means better fault localization effectiveness for it. More details of the metric will be described in Section 3.5.1.

prioritization techniques and fault localization techniques can be integrated, and examined the factors that affect the effective integration of testing and debugging activities.

Statement	Test Case 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	1	1	0.5
Random (R)	3rd	1st	2nd	1	0.66	0.5

Test case prioritization technique (see Section 2.1)

Expense (see Section 3.5.1) by Tarantula (see Section 2.2.1)

Fig. 2. The testing/debugging dilemma.

A developer may want to stop regression testing for a commit build earlier with a view to shortening the CI process. For instance, a CI server may be instructed to stop a regression test after encountering a certain number of failed test cases (say, 10 or more) and return a bug report to the developer for debugging. In this way, the developer can debug the code module while their memory on the modification is still fresh. Alternatively, the CI server may have a limited time budget in regression testing (say, in terms of the number of test cases to be executed). In either case, the commit-build process merely executes some high-priority test cases in a prioritized test suite, and yet its bug reports are supposed to support fault localization conducted by developers later. How well do the high-priority test cases of a test suite support statistical fault localization? To what extent do the test suites prioritized by existing test case prioritization techniques support effective statistical fault localization? What are the factors in the testing techniques that effectively affect such integration? Knowing the answers to these questions is critical toward a tighter integration among software development activities.

This paper extends its preliminary version [26] in the following aspects: (i) In addition to the study of the integration of random ordering and six coverage-based test case prioritization techniques with fault localization effectiveness, this paper reports the empirical study results on the integration of adaptive random test case prioritization techniques with fault localization. (ii) It further investigates the problem of whether the studied test case prioritization techniques can support effective fault localization if testers stop regression testing of the commit build after encountering different numbers of failed test cases. (iii) It strengthens the empirical study by using four additional real-life UNIX utility programs with both single and multi-fault versions as subjects.

We find the following results from our empirical study: If testers want to stop regression testing of the commit build after a certain number of failed test cases have been observed, the test suites produced by random ordering can be a cost-effective option to integrate with (existing) statistical/spectrum-based fault localization techniques. We also find that different levels of coverage granularity do not result in significant differences in supporting effective fault localization by the studied techniques. Interestingly, we find that adding more failed test cases through test case prioritization to statistical fault localization techniques as inputs may deteriorate the fault localization effectiveness in the mid-range. If testers have a budgetary concern on the number of test cases for regression testing, the use of test case prioritization can save up to 40% of test case executions for commit builds without significantly affecting the effectiveness of statistical fault localization. The savings are more noticeable on a medium-sized program than on a small-scale program and, on a multi-fault program than on a single-fault program. Last but not least, we find that the

inclusion of more failed test cases will improve the fault localization effectiveness of integrated techniques, despite the presence of some variations.

The main contribution of the paper with its preliminary version is twofold. (i) To the best of our knowledge, we report the first study on the integration between fault localization and test case prioritization techniques. (ii) The paper reports a multi-faceted result on the integration effectiveness between regression testing and fault localization techniques. Our result shows that the effectiveness of fault localization techniques can be seriously compromised if testers merely use a small fraction of a test suite for a commit build. Fortunately, we also find from the experiment that executing the top 60% of a prioritized test suite can be a cost-effective choice because it can, in general, provide fault localization effectiveness comparable to that of the whole test suite for all the subject programs studied. The study shows that the Random ordering strategy can be a cost-effective technique, followed by the ART and the Additional strategies, while the Total strategy is the least effective in supporting such integration. The result further shows that different coverage granularity levels do not result in significant differences in terms of their support to effective fault localization.

The rest of this paper is organized as follows: Section 2 revisits selected test case prioritization techniques and fault localization techniques to be used in the experiment. Section 3 describes the empirical study, followed by its results in Section 4. Section 5 reviews related work. We conclude the paper in Section 6.

2. Techniques revisited

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

2.1 Test case prioritization techniques

Test case prioritization permutes the test cases in a test suite to increase a specific testing goal. In previous work, such testing goal can be the rate of fault detection, code coverage, or some other units of measurement. Unlike test case reduction or test case selection techniques that discard test cases, test case prioritization retains all the test cases, and hence the fault detection capability of the test suite will not be compromised. Following on with our previous work [25], we study two groups of general test case prioritization techniques, namely, the greedy techniques and ART-based techniques. The greedy techniques are coverage-based greedy algorithms [37], which can be further subdivided into the Additional and the Total strategies. For the ART-based techniques, we study nine techniques proposed in our previous work [23][25]. Both groups of techniques rely on code-coverage information obtained from the test execution of the previous (baseline) version of the program. Therefore, we follow the procedure described in Rothermel et al. [37] to study the effect of code coverage granularity on the effectiveness of test case prioritization techniques. We also follow Rothermel et al. to use statement and branch coverage data to represent a finer granularity and use function coverage data to represent a coarser granularity.

2.1.1 Greedy techniques

When we combine the two greedy strategies with the three coverage (granularity) levels, we produce six greedy 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). We briefly describe these techniques in this subsection. Interested readers may refer to Rothermel et al. [37] for a detailed description.

The total statement (total-st) test case prioritization technique computes the statements that have been covered in the execution of each program version over each test case. It permutes test cases in descending order of the total number of statements covered by the respective test case. When two test cases cover the same number of statements, it 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, respectively, instead of statement coverage information.

The additional statement (addtl-st) test case prioritization technique is the same as total-st, except that it selects a 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 of the test suite being constructed, addtl-st resets all the statements to "not yet covered" and reapplies the same procedure on the remaining test cases. When two test cases cover the same number of additional statements in a round, it randomly picks one.

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 data, respectively, rather than statement coverage data.

2.1.2 ART-based techniques

We summarize the techniques for ART-based test case prioritization [25] as follows. The basic algorithm accepts a test suite containing a sequence of test cases as its input, and produces a sequence of prioritized test cases. It 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 test cases that have not yet been selected into the candidate set one by one as long as they can increase the code coverage achieved by the candidate set. To decide on the 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 our previous work [25] to measure the distance between two test cases using the *Jaccard* distance (a special case of the Tanimoto distance, which satisfies the triangle inequality property of a distance function [33]) based on their code coverage data. Suppose the set of statements (or functions or branches) covered by test cases p_j and c_i are $S(p_j)$ and $S(c_i)$, respectively. We have

$$f_1(p_j, c_i) = 1 - \frac{|S(p_j) \cap S(c_j)|}{|S(p_j) \cup S(c_j)|}.$$

Function f_2 describes the strategy to select the farthest test case from those test cases that have already been prioritized. We also follow our previous work [25] in defining 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 test cases already selected. Given a matrix D of distances between each pair of test cases in the test suite, $f_2(D)$ is given by

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

In total, 16 test case prioritization techniques are considered in this paper, including nine ART test case prioritization techniques, six greedy test case prioritization techniques, and random ordering [16]. We summarize the properties of all the test case prioritization techniques in Table 1.

2.2 Fault localization techniques

Researchers have proposed numerous techniques to help developers locate faults. The statistical fault localization approach [2][27][28][32][34][35][47][48][49][50] conducts statistical analysis on program execution traces and pass/fail information of executed test cases to generate a ranked list of suspicious program entities (such as statements) for the developers to inspect in turn in the code. A popular hypothesis is that if a statement is frequently executed in failed test cases but rarely executed in passed test cases, then it is more suspicious to be faulty or related to faults. We revisit four statistical fault localization techniques in this section, which will be used in our empirical study. They are also the four techniques selected by Yu et al. [44] in their study of the effects of test suite reduction on statistical fault localization. There are also many other fault localization techniques proposed by various researchers [13][22][36] that are not included in our experiment.

We note also that we use the statement-level program spectrum as the basis for fault localization. By so doing, our study not only provides all fault localization techniques with a common basis for comparison, but also represents a fine-level program spectrum that likely leads to higher fault localization effectiveness than the use of a coarse-level program spectrum. This serves the goal of the target integration.

2.2.1 Tarantula

Jones and colleagues [27][28][44] 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

[44], according to the coverage information on passed and failed test cases. The *suspiciousness* of a statement *s* is given by the formula

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

The function $\%failed(s)$ tallies the percentage of failed test cases that execute statement *s* (among all the failed test cases in the test suite). The function $\%passed(s)$ is similarly defined.

The *confidence* metric, computed as follows, indicates the degree of confidence on a suspiciousness value:

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

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

Table 1. Test case prioritization techniques studied.

Ref.	Name	Descriptions	
$T1$	random	Random ordering	
Ref.	**Greedy**	**Level of coverage information**	**Coverage-based strategy**
$T2$	total-st	Statement	Total
$T3$	total-fn	Function	
$T4$	total-br	Branch	
$T5$	addtl-st	Statement	Additional
$T6$	addtl-fn	Function	
$T7$	addtl-br	Branch	
Ref.	**ART**	**Level of coverage information**	**Test set distance (f_2) in ART strategy**
$T8$	ART-st-maxmin	Statement	Equation (1)
$T9$	ART-st-maxavg		Equation (2)
$T10$	ART-st-maxmax		Equation (3)
$T11$	ART-fn-maxmin	Function	Equation (1)
$T12$	ART-fn-maxavg		Equation (2)
$T13$	ART-fn-maxmax		Equation (3)
$T14$	ART-br-maxmin	Branch	Equation (1)
$T15$	ART-br-maxavg		Equation (2)
$T16$	ART-br-maxmax		Equation (3)

2.2.2 Cooperative Bug Isolation (CBI)

Liblit et al. [32] deployed the notion of Cooperative Bug Isolation (CBI) to develop a technique for identifying faults. The technique is adapted by Yu et al. [44] to calculate the suspiciousness (called SBI) of a statement *s* as follows:

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

The functions $failed(s)$ and $passed(s)$ tally the number of failed and passed test cases, respectively, that execute *s*.

2.2.3 Jaccard similarity coefficient

Abreu et al. [1] use the Jaccard similarity coefficient for binary data as a suspiciousness formula. The equation for the Jaccard coefficient is given by

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

The functions $failed(s)$ and $passed(s)$ have the same meaning as in CBI. The variable *totalfailed* is the number of failed test cases in the test suite. The technique ranks the statements similarly to Tarantula.

2.2.4 Ochiai similarity coefficient

Abreu et al. [1] also propose to use the Ochiai similarity coefficient as another suspiciousness formula. The equation for Ochiai coefficient is given by

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{totalfailed \times (failed(s) + passed(s))}}$$

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

3. Empirical study

In this section, we present our study on the relationship between regression testing and debugging.

3.1 Research questions

The empirical study addresses two research questions:

RQ1: During a commit build, if the developers stop regression testing after the process has encountered *i* failed test cases, which test case prioritization strategy (Random, Greedy, or ART) and which level of coverage granularities (function, statement, or branch) are more helpful to developers in conducting effective fault localization?

RQ2: If we are resource conscious about the number of test cases for regression testing, to what extent will a fault localization technique be affected if it only uses the execution statistics of the high priority test cases as input?

Some previous studies [16][31] have empirically shown that both strategy and coverage granularity can be key factors that significantly affect the rates of fault detection or the rates of code coverage of test case prioritization techniques. At the same time, another previous study [1] has shown empirically that a statistical fault localization technique that uses a small number of failed test cases (which is 5 as reported in that study) already suffices to provide the same level of fault localization effectiveness as if the entire test suite were used. If this is the case, a statistical fault localization technique may use a subset of a test suite that contains a few failed test cases. If such a test suite is generated via test case prioritization, it is unclear whether factors like strategy and coverage granularity that affect test case prioritization may significantly affect the effectiveness of statistical fault localization technique. Moreover, if such a factor exhibits any effect, it is also unclear what direction (i.e., improvement or otherwise) that such a factor may act on such integration in the CI process. The answers to these research questions will disclose the relationships between such factors and the effectiveness of statistical fault localization.

RQ2 explores whether quick commit builds may preserve the effectiveness of the fault localization techniques as if the entire test suites were used. In a typical CI process, the time budget for regression testing of a commit build is limited. The use of fewer test cases is an obvious choice to meet such a time budgetary constraint. Nonetheless, the positive or adverse effects of such a decision on the CI process are unclear. The answer to RQ2 unveils the tradeoff between time budgets and fault localization effectiveness, which is crucial if both test case prioritization and fault localization techniques are used in a CI process.

3.2 Subject programs and test pools

We used two sets of subject programs in our experiment. The first was the Siemens suite of seven programs and the second was a suite of four UNIX programs. The Siemens programs were originally created to support research on data-flow and control-flow test adequacy criteria. Because they were small in scale, we also used four real-life UNIX utility programs with real or seeded faults. Since the Siemens suite was first produced, it has gone through many modifications and many papers have reported quite different descriptive statistics [40]. There are also numerous versions of the UNIX programs. For ease of reference by readers, we downloaded all our subject programs from the Software-artifact Infrastructure Repository (SIR) [14], available at http://sir.unl.edu (last accessed in June 2010).

Table 2 shows the descriptive statistics of our subject programs. The column headed by No. of faulty versions lists the number of faulty versions for each subject program. The column headed by No. of faults per version shows the number of faults for each faulty version. The column SLOC shows the numbers of executable source 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. For example, the last row of the table shows the statistics for the sed program: The natural program versions range from version 1.18 to version 3.02. There are 17 single fault faulty versions and six multi-fault faulty versions, each with two to three faults. Finally, there are 4756 to 9289 lines of executable source code and 370 test cases for this program in the whole test pool.

Table 2. Subject programs.

Subject	No. of faulty versions	No. of faults per version	SLOC	Test pool size
tcas	41	1	133–137	1608
schedule	9	1	291–294	2650
schedule2	10	1	261–263	2710
tot_info	23	1	272–274	1052
print_tokens	7	1	341–342	4130
print_tokens2	10	1	350–354	4115
replace	32	1	508–515	5542
flex (2.4.7–2.5.4)	21	1	8571–10124	567
	4	2–3		
grep (2.2–2.4.2)	17	1	8053–9089	809
	6	2–3		
gzip (1.1.2–1.3)	55	1	4081–5159	217
	4	2–3		
sed (1.18–3.02)	17	1	4756–9289	370
	6	2–3		

We excluded the versions whose faults cannot be revealed by any test case in our test infrastructure and environment. We also excluded the versions whose faults were too obvious (where more than 25% of all the test cases in the pool can detect them [16]) and the versions that do not work with the standard coverage tool gcov (which was used to collect the coverage information of program executions in our test infrastructure). Similar strategies, magic numbers, and tools were also used in previous experiments [16][17][23][25]. Finally, we used the remaining 212 faulty versions for data analysis. Faults in the faulty versions were either real or seeded. According to SIR [14], the seeded faults mimicked real world faults made by developers, including logical operator errors, missing statements, wrong definitions, missing definitions, and so on. We used the original program versions as the “golden” versions and compared the execution results of a test case between an original version with that of a corresponding faulty version to determine whether it passes or fails. A fault is exposed if there is a discrepancy between the two test results.

3.3 Experimental setup

This section presents the experiment setup for the empirical study.

We applied the 16 test case prioritization techniques (see Table 1) and the four fault localization techniques to the 212 versions of our subject programs and their test suites. In this section, we present the setup of the experiment.

For the seven Siemens programs, we followed our previous work [23][24][25][26] to use the branch-adequate test suites provided by SIR to conduct test case prioritization. There are 1000 small test suites and 1000 large test suites, both of which are branch adequate. Each small test suite contains about 30 test cases while each large test suite contains about 100 to 200 test cases. For the small test suites, testers can simply retest all test cases because the execution time of the entire test suite is trivial. There is no real need to conduct test case prioritization on them. Consequently, we chose to use the large test suite for our empirical study, because test case prioritization is only needed when the time cost of executing the test suite is non-trivial. For the four UNIX programs, since the test pool size was not large with respect to the program size, we followed [17][23][25] to use the whole test pool as a suite for prioritization.

All the ART-based techniques and the random ordering technique are based on random selection. We repeated each of them 20 times to obtain an average performance. To reduce the huge computation cost incurred in the experiment, we randomly selected 50 test suites from the available 1000 test suites for each subject for the Siemens

programs. Thus, we conduct a total of 1000 prioritizations for each ART-based technique and 4000 rounds of fault localizations for each Siemens program. Since we used the whole test pool of UNIX programs as a test suite, we conducted a total of 20 prioritizations for each ART-based technique and 80 rounds of fault localizations for each UNIX program. The whole experiment took about one month for all the executions to complete.

Suppose the positions of the 1st, 2nd, ..., *m*th failed test cases in the prioritized test suite are $f_1, f_2, ..., f_m$, respectively, where m is the total number of failed test cases for the test suite. To answer research question RQ1, we stop the regression testing in the commit build if f_i test cases (for i = 1, 2, ..., m) has been executed. We conducted the experiment on each faulty version of every subject program and averaged out the results over all the four fault localization techniques considered.

For the experimental setup of research question RQ1, each test case prioritization technique generated m test suites of different sizes for fault localization, where m is again the total number of failed test cases. In this way, we could evaluate how a test case prioritization technique changes in terms of expense when different prefixes of a prioritized test suite were used for fault localization.

To answer research question RQ2, for every prioritized test suite generated by each test case prioritization technique, we used its top 10%, 20%, ..., 100% test cases for the commit build. If a percentage of a test suite is not an integer value, it is rounded down to the nearest integer. We then used the corresponding resultant suites for fault localization. We conducted the experiment on each faulty version of every subject program and averaged out the results over all the four fault localization techniques. As a result, for the experimental setup for research question RQ2, each prioritized test suite generated 10 test suites of different sizes for the commit build.

3.4 Experimental environment

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

3.5 Main metrics

3.5.1 Expense

We use *expense* [22][27][44] as the metric to measure fault localization effectiveness. Given a ranked list produced by a fault localization technique, *expense* measures the minimum percentage of statements in a program that must be examined in descending order of the ranks so as to include the fault in the set of examined statements. It is defined by the formula

$$expense = \frac{\text{rank of the faulty statement}}{\text{number of executable statements}}$$

where a smaller expense indicates a better result.

When conducting fault localization on multi-fault programs, we first measure the expense for locating the first fault. After fixing the first fault, we conduct further fault localization and measure the expense for locating the second fault in the modified program. After fixing the second fault, we continue with the fault localization and measure the expense for locating the third fault, and so on. In this way, we iteratively measure the expense for each fault. Finally, we report the *mean* and *variance* of the expenses for locating all the faults in a multi-fault program. We measure the expenses for multi-fault programs in this way because previous papers (such as [48]) have reported this kind of analysis, which appears to be close to how developers debug multi-fault programs in practice. For instance, developers may firstly fix one bug, submit the changed code to the CI server for regression testing to confirm that the bug has been correctly fixed, and then use the updated bug report to assist them to fix another bug.

3.5.2 Area Under Curve (AUC)

We use *Area Under Curve* (*AUC*) to measure the cumulative fault localization expense for a given strategy or dimension after the tester has encountered a number of failed test cases and stopped regression testing. Thus, given a strategy or dimension and the total number of failed test cases n,

$$\textit{Area Under Curve}\ (AUC) = \sum_{i=1}^{n} expense(i)$$

where *expense*(i) is the mean *expense* achieved when i failed test cases in a test suite has been observed in the commit build. Like the *expense* metric, the lower value of *AUC*, the better will be the result.

For example, the curve in Fig. 3 shows the expenses for a given strategy or dimension after encountering different numbers of failed test cases in a commit build, until all the n failed test cases have been observed. The x-axis represents the number of failed test cases encountered in the commit build whereas the y-axis represents the expenses. If we are interested in the cumulative fault localization effectiveness for that strategy or dimension, we may use the Area Under Curve (and above the x-axis) to represent it.

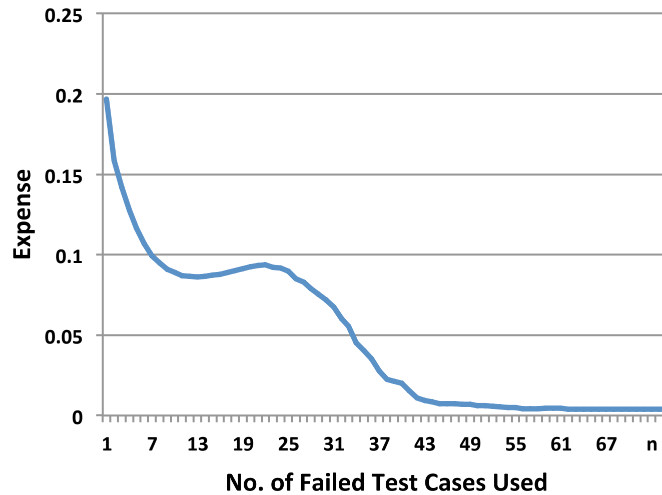

Fig. 3. Example illustrating AUC metric.

4. Data analysis and discussions

4.1 Effectiveness

In this section, we first present the results for each research question on the Siemens programs, and then analyze them to answer research questions RQ1 and RQ2, respectively.

*4.1.1 Answering RQ*1

4.1.1.1 Comparison of test case prioritization strategies

In this section, we report for each test case prioritization strategy the mean result over all the techniques. We use ART to stand for the mean expense for the ART-st-maxmin, ART-st-maxavg, ART-st-maxmax, ART-fn-maxmin, ART-fn-maxavg, ART-fn-maxmax, ART-br-maxmin, ART-br-maxavg, and ART-br-maxmax techniques. The curve for Additional represents the mean expense for the addtl-st, addtl-fn, and addtl-br techniques. The curve for Total stands for the mean expense for total-st, total-fn, and total-br.

We show in Fig. 4 the trend of fault localization effectiveness after encountering the first i failed test cases in a test suite for a commit build on the Siemens programs. The x-axis represents the number of failed test cases observed before we stop the execution of the prioritized test cases during a commit build, and the y-axis represents the mean expense achieved by using the corresponding portion of the prioritized test suites for fault localization. The four curves correspond to the mean expenses achieved by Random ordering, ART, Additional, and Total.

For ease of discussion, we divide the graph in Fig. 4 into three regions using two vertical lines.

(a) We observe that in the leftmost region, the fault localization effectiveness for different test case prioritization strategies improves rapidly if more failed test cases are available. It clearly indicates that regression testing should continue beyond the first failed test case until a few more failed cases have been reported.

(b) In the middle region, on the other hand, the fault localization effectiveness represented by each curve is not monotonically increasing. It shows that in terms of expense, adding more failed test cases may even adversely

affect the fault localization effectiveness. Moreover, the experiment has used quite a number of test case prioritization techniques over 122 faulty versions. Even for the Random ordering strategy that gets rid of the subjective heuristics used in different test case prioritization strategies and coverage granularity levels, we still observe such variations.

We note that every analyzed program version contains a single fault. We can, therefore, rule out the reason that such variations in the fault localization effectiveness of the test suites may be due to the interference among different faults in the same program version.

It appears to us that the use of adequate test suite and the number of failed test cases have complicated interactions. This finding is very interesting and, to the best of our knowledge, no previous empirical study has reported this kind of result.

An empirical study involves the analysis of observations and the validation of hypothesis in specific research questions. It differs from an experimental study, which verifies predicted behaviors based on models and theories. In other words, empirical studies are more of an explorative nature and apply to an early stage of the scientific research on a particular issue before models and theories are postulated. Hence, it is beyond the scope of the present empirical study to explain the theoretical cause of the small variations in fault localization effectiveness.

On the other hand, to facilitate future research, we put forward two potential reasons for consideration. First, some of the test cases in a test suite may show coincidental correctness, which causes the contrasting step in statistical fault localization to be imprecise. Second, the formulas for statistical fault localization techniques are non-linear with respect to the number of failed test cases.

(c) Finally, the effectiveness of different strategies gradually increases and converges to almost the same value at the final range. Like the first range, the finding in this range again agrees with the general intuition that to make a given test suite more effective for fault localization, it is better to make use of as many failed test cases as possible. This finding is regardless of the test case prioritization strategy employed.

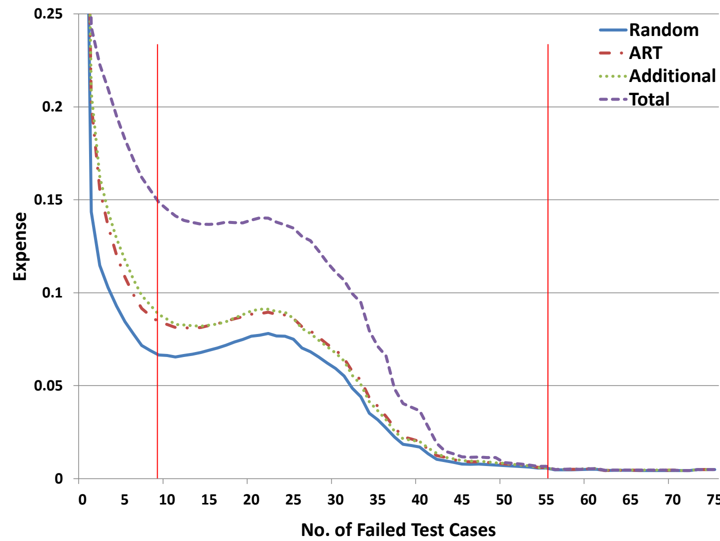

Fig. 4. Comparison of test case prioritization strategies on fault localization effectiveness for Siemens programs.

Our finding is consistent with, but more detailed than, the result reported in Abreu et al. [1], who find that simply using up to five test cases helps improve the fault localization effectiveness of an enhanced test suite to match that of using the entire test pool. Our results show that after encountering the first few failed test cases, the inclusion of more failed test cases may lead to a variation period where the fault localization effectiveness fluctuates. Moreover, only when going beyond this variation period will the fault localization be improved again significantly by the inclusion of more failed test cases. The results in both [1] and this paper show that the inclusion of more failed test cases initially helps improve the fault localization effectiveness. Our results in particular imply that we should continue to include as many failed test cases as possible to achieve better fault localization effectiveness despite the small variations period.

We also observe another interesting result. We recall that previous studies [16][25][37] consistently reported that the ART, Additional, and Total strategies are more effective in exposing faults earlier than the Random ordering strategy on the Siemens programs. Indeed, almost all previous empirical studies on test case prioritization research consider Random ordering as the lower bound technique (on average) to expose faults as early as possible [16][37]. Our results, however, show that although Random can be slower in exposing faults, if we base on the failures observed from a randomly ordered test suite to decide when to stop regression testing, we can actually provide *better* fault localization effectiveness than the ART, Additional, and Total strategies in the initial and middle ranges. We also observe that the curve for Random forms a lower frontier (i.e., the best result) among all the strategies on the Siemens programs for the first two regions. Random ordering is simple and objective, and has the advantage of low cost in terms of the prioritization process. It can be better than other test case prioritization strategies for this scenario (on the Siemens programs) when the target is to locate faults.

Moreover, in a previous empirical study [23][25], the Total strategy is found to be comparable to the ART and Additional strategies in terms of rate of fault detection in the Siemens programs. However, the results in Fig. 4 indicate that Total is relatively the least effective strategy in making statistical fault localization effective.

Nonetheless, the study reported here can only show the correlation, but has not revealed the underlying reason why the Total strategy is ineffective for statistical fault localization. At this stage, it appears to us that Total may select consecutive passed executions clustering across a set of statements, branches, or functions that is quite different from the failed executions. It may effectively provide less contrast to distinguish statistically the statements, branches, or functions related to the failed executions. In this connection, the Random strategy provides a higher probability to allow a statistical fault localization technique to distinguish different program entities.

To compare the cumulative fault localization expenses, we calculate the AUC for each strategy. We find from Table 3 that the Random strategy is 21.6%, 24.0%, and 93.2% better than the ART, Additional, and Total strategies, respectively. In other words, Random (2.83) is the most effective fault localization strategy on the Siemens programs. The ART strategy (3.44) is slightly better than the Additional strategy (3.51). The Total strategy (5.47) is the least effective strategy on the Siemens programs.

Table 3. Areas under curve for different strategies on Siemens programs.

Strategy	Random	ART	Additional	Total
AUC	2.83	3.44	3.51	5.47

4.1.1.2 Comparison of coverage granularity levels

In this section, we report the mean expenses for different levels of coverage granularity across all the test case prioritization techniques studied. We use “Branch”, “Function”, and “Statement” to represent the mean expenses for all the branch-level techniques, function-level techniques, and statement-level techniques studied, respectively, on the Siemens programs. For example, the curve for Branch is the mean of the expenses of using ART-br-maxmin, ART-br-maxavg, ART-br-maxmax, addtl-br, and total-br on all the analyzed test suites. The other two levels of coverage granularity are computed similarly. We do not include Random in this analysis because Random does not depend on such information for ordering.

We observe from Fig. 5 that there is no noticeable difference in fault localization effectiveness between coarse granularity (function-level) techniques and fine granularity (statement- and branch-level) techniques. We compute the area under curve for each level of granularity as shown in Table 4. The result also confirms the observation that branch-level granularity (3.74) is slightly better than function-level granularity (3.91) and statement-level granularity (3.94), and yet the corresponding differences constitute only 4.5% and 5.3%, respectively. Apparently, the effectiveness at different levels of coverage granularity is comparable. We further conduct the ANalysis Of VAriance (ANOVA) test, and the resulting large p-value of 0.112 confirms that there is no significant statistical difference between coarse granularity techniques and fine granularity techniques in terms of the expense over all test suites at a significance level of 5%. Intuitively, given a specific test case prioritization strategy, the use of coarse (function-level) coverage granularity information is a better choice as it incurs less cost in terms of both coverage profiling and prioritization time.

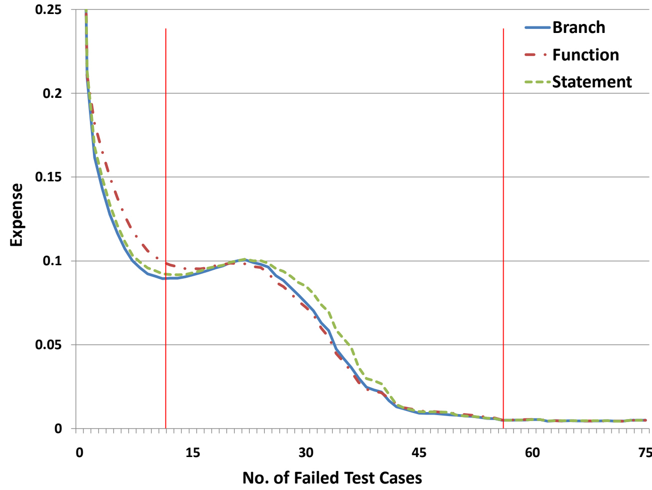

Fig. 5. Comparison of coverage granularity levels on fault localization effectiveness for Siemens programs.

Table 4. Areas under curve for different coverage granularity levels on Siemens programs.

Coverage granularity	Branch	Function	Statement
AUC	3.74	3.91	3.94

4.1.1.3 Comparison of distributions of different test case prioritization techniques

In this section, we report variances of the expenses of using the test suites generated by different test case prioritization techniques for statistical fault localization. Table 5 summarizes the results. Each column represents a test case prioritization technique. Each row represents the variance of expenses for the same number of failed test cases. In each row, we have highlighted in gray the cell with the largest variance (unless there are multiple cells having the same largest variance).

We observe that the 13 Random, ART and Additional techniques are comparable to one another in terms of variances, whereas the variance of the expenses for the three Total techniques are comparably larger than the former techniques when the number of failed test cases is no more than 40. Thus, when compared with each of the three Total techniques, Random ordering achieves a smaller variance when the number of failed test cases is small. When the number of failed test cases exceeds 40, the variances of different test case prioritization techniques are comparable to one another.

Table 5. Variances of expenses on Siemens programs.

Number of failed test cases	Random	ART-br-maxmax	ART-br-maxavg	ART-br-maxmin	ART-fn-maxmax	ART-fn-maxavg	ART-fn-maxmin	ART-st-maxmax	ART-st-maxavg	ART-st-maxmin	addtl-br	addtl-fn	addtl-st	total-br	total-fn	total-st
10	0.016	0.016	0.017	0.015	0.014	0.020	0.019	0.016	0.017	0.015	0.014	0.024	0.014	0.030	0.025	**0.033**
20	0.015	0.017	0.018	0.017	0.017	0.020	0.018	0.017	0.018	0.017	0.016	0.023	0.016	0.030	0.022	**0.031**
30	0.012	0.014	0.014	0.015	0.013	0.014	0.013	0.014	0.015	0.015	0.013	0.016	0.012	0.021	0.016	**0.032**
40	0.004	0.004	0.004	0.005	0.004	0.004	0.004	0.004	0.004	0.005	0.004	0.005	0.003	0.006	0.005	**0.011**
50	0.001	0.001	0.002	0.002	0.002	0.002	0.001	0.002	0.002	0.002	0.001	0.002	0.001	0.002	0.002	0.002
60	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
70	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Max	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001

We further group different test case prioritization techniques in the dimension of strategy as shown in Table 6. We have similar observations that the variances of the expenses for the Random, ART, and Additional strategies are comparable to one another. Also, Random, ART, and Additional have smaller variances than the Total strategy if the number of failed test cases is small (no more than 40), which is likely the case in practice. If we view different test case prioritization techniques in the dimension of coverage granularities, the result is shown in Table 7, which excludes Random as it does not depend on such information. We find that the variances of the expenses of using Statement, Branch, and Function as the coverage granularities are comparable to one another.

Table 6. Variances of expenses for prioritization strategy on Siemens programs.

Number of failed test cases	Random	ART	Additional	Total
10	0.016	0.017	0.017	**0.029**
20	0.015	0.018	0.018	**0.028**
30	0.012	0.014	0.014	**0.023**
40	0.004	0.004	0.004	**0.007**
50	0.001	0.002	0.001	0.002
60	0.001	0.001	0.001	0.001
70	0.001	0.001	0.001	0.001
Max	0.001	0.001	0.001	0.001

Table 7. Variances of expenses for coverage granularity on Siemens programs.

Number of failed test cases	Branch	Function	Statement
10	0.018	**0.020**	0.019
20	0.020	0.020	0.020
30	0.015	0.014	**0.018**
40	0.005	0.004	0.005
50	0.002	0.002	0.002
60	0.001	0.001	0.001
70	0.001	0.001	0.001
Max	0.001	0.001	0.001

*4.1.2 Answering RQ*2

To answer RQ2, we compute the overall fault localization effectiveness of using a particular percentage of a test suite for a commit build (regardless of the choice of test case prioritization or fault localization techniques), as shown in Fig. 6. The *x*-axis represents the percentages of a test suite being used to conduct a commit build. The *y*-axis represents the expenses. Each box in the figure represents the distribution of expenses over all test case prioritization techniques for a given percentage of test suite used to conduct a commit build, as indicated by the *x*-value in the plot.

We observe from Fig. 6 that with an increase in the percentage of a test suite used for statistical fault localization, the decrease in expense is very noticeable, especially when the percentage is small. For instance, when using the first 10% of a test suite for fault localization, the median expense is 0.42. When using the first 20%, the median expense decreases to 0.36. When using the entire test suite for fault localization, the resultant expense is only 0.24. However, the variances of the expenses in all three scenarios are almost the same, which is around 0.04. The result indicates that although code examination efforts can be reduced by using more high priority test cases, the variance does not improve when more test cases are used for fault localization.

If we look closer at Fig. 6, we find that when a higher percentage of a test suite is used for a commit build, the decrease in expense (or increase in fault localization effectiveness) gradually become slower. In fact, the difference in the median expense between using 60% and 100% is only 0.02. We further conduct an ANOVA test, validating the null hypothesis that there is no significant difference between the mean expenses of using different percentages of test suite for fault localization at a significance level of 5%. The ANOVA test returns a *p*-value of 0.0037, which successfully rejects the null hypothesis and shows that there are significant differences among the groups.

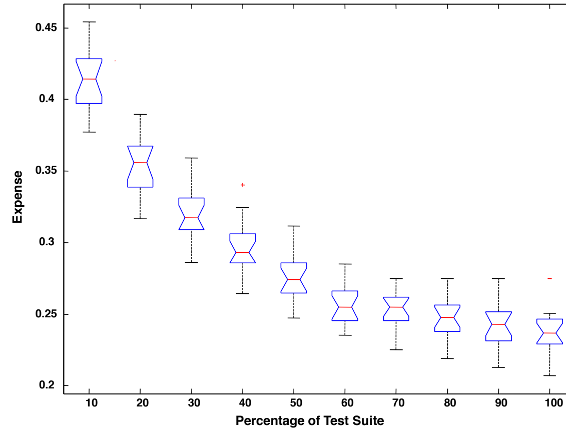

Fig. 6. Distributions of expenses for different percentages of test suites on Siemens programs.

In an ANOVA test, we compare the means and variances of several groups to validate statistically the hypothesis that all these groups are drawn from the same population, as against the general alternative that they are not from the same population. We further want to know which pairs of the means are significantly different and which are not. A test that can provide such information is called a *multiple comparison* procedure.

Thus, to verify whether the mean expenses for two different percentages of a test suite differ significantly, we further conduct multiple comparisons to compare the mean expenses of using different percentages of a test suite for a commit build at a significance level of 5%. For each multiple comparison procedure conducted in this work, we use Tukey's *Honestly Significant Difference* (*HSD*) as the alpha adjustment procedure, which is the default option available in MATLAB. The result is shown in Fig. 7. The multiple comparisons are conducted between using 100% of a test suite for a commit build and using other percentages of the test suite for the commit build. The *x*-axis shows the expenses and the *y*-axis shows the different percentages of test suites. Each horizontal line in the figure represents the distributions of expenses using a specific percentage of the test suite for the commit build, and the central point indicating the mean expense. The horizontal lines cut by the two dotted vertical lines indicate that their mean expenses are *not* significantly different from that of using 100% of test suites for commit build at a significance level of 5%.

The result of multiple comparisons in Fig. 7 shows that there is no statistical difference between the mean expense of using an entire test suite for a commit build and that of using 60% or more of the test suite at a significance level of 5%. However, there is a significant difference between the mean expenses of using an entire test suite and using no more than 50% of the test suite. The result also shows that for programs of the scale represented by the Siemens suite, the CI process can reduce the number of test case executions by up to 40% while maintaining a fault localization effectiveness result similar to that when the entire test suite is being used.

4.2 Scalability of single-fault subjects

We want to know whether our finding on the Siemens programs can be observed in other real-life scenarios. Therefore, we further analyze the results for single-fault and multi-fault UNIX programs. In this section, we present the results for single-fault UNIX programs and re-examine the two research questions. In the next section, we will report the result on multi-fault UNIX programs.

*4.2.1 Answering RQ*1

Similarly to the experiment on the Siemens programs, we first study the scenario of developers stopping a commit build whenever a specific number of failed test cases have been observed.

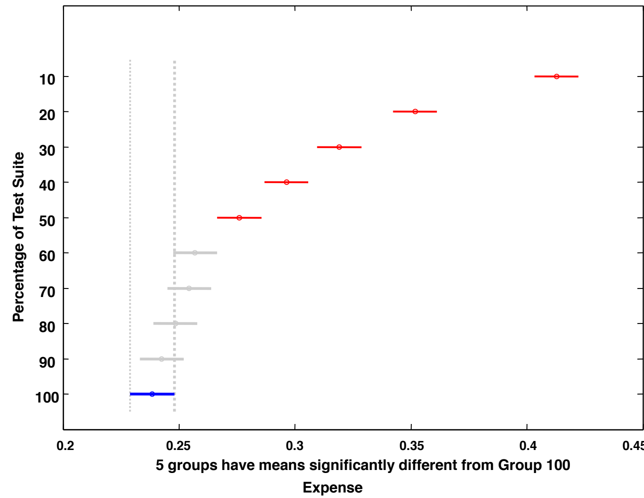

Fig. 7. Multiple comparisons of expenses for different percentages of test suite on Siemens programs.

4.2.1.1 Comparison of test case prioritization strategies

This section reports on the comparison of different test case prioritization strategies as shown in Fig. 8. The *x*- and *y*-axes of Fig. 8 can be interpreted similarly to Fig. 4. The expenses in each curve are computed in the same way as in Section 4.1.1.1, except that we use single-fault UNIX programs instead of the Siemens programs as subjects.

Similarly to what we did for Fig. 4, we divide the plot in Fig. 8 into three regions using two vertical lines. We observe that during the initial range (i.e., the leftmost region), the fault localization effectiveness of all the strategies studied (except Total) improves rapidly when more failed test cases are encountered. In the middle region, roughly speaking, the fault localization effectiveness of all the strategies deteriorates towards the end of the range. Total fluctuates more seriously than the other strategies. We observe also that there are quite a number of peaks and troughs along the curve of Total in the middle region. Finally, all the curves converge to the same value in the final range. This general trend is similar to what we have observed from Fig. 4. By comparing these curves in Fig. 8, we also find that Total is the least effective among the strategies studied. The trends of the Random, ART, and Additional strategies are observably comparable to one another, and quite different from the trend of the Total strategy.

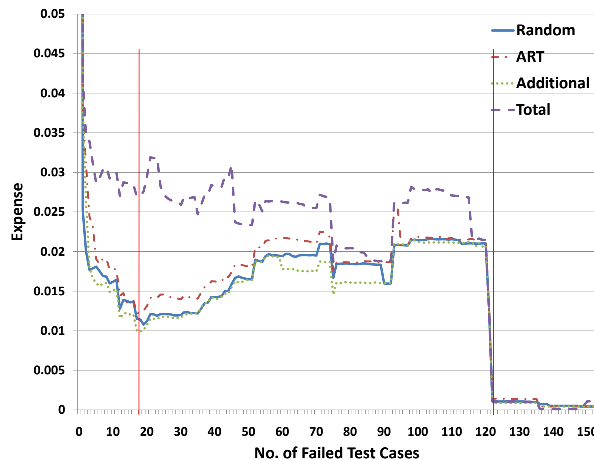

Fig. 8. Comparison of test case prioritization strategies on fault localization effectiveness for single-fault UNIX programs.

Encouragingly, the effectiveness of the ART and Additional strategies catches up with that of the Random strategy. However, the former strategies do not outperform the latter noticeably. In order to understand this observation better, we further compute the area under curve of each strategy, as shown in Table 8. We observe that, in general, the Random (2.14), ART (2.33), and Additional (2.05) strategies only differ by about 2 to 13%, and they all perform significantly better than the Total strategy (3.16) by at least 35%. Random ordering can be applied at low cost. On the other hand, consistent with the time cost reported in Jiang et al. [23][25], the Additional strategy is the least efficient among the studied techniques, and the time cost of the ART strategy is much lower than that of the Additional strategy. The result echoes the result in Section 4.1.1.1 that owing to its low prioritization cost, the Random ordering strategy is useful if the project concerned is buggy; whereas the ART or Additional strategy can be viable options if the goal is to reassure all the existing features of a baseline version.

Table 8. Areas under curve for different strategies on single-fault UNIX programs.

Strategy	Random	ART	Additional	Total
AUC	2.14	2.33	2.05	3.16

4.2.1.2 Comparison of coverage granularity levels

Similarly to our analysis on the raw data of Fig. 5 for the Siemens programs, we report our result on the single-fault UNIX programs in Fig. 9. We find the trends of the curves for these subject programs are similar to that for the Siemens programs. We also compute the area under curve for each level of granularity, as shown in Table 9. We find that different levels of coverage granularity do not result in any significant difference in the cumulative fault localization effectiveness. This is confirmed by an ANOVA test that returns a p-value of 0.145, which cannot reject the null hypothesis that there is no difference in expenses between different coverage granularity levels at a significance level of 5%. Thus, when the test case prioritization strategy (other than Random) is fixed, a coarse (function-level) coverage granularity can be a better choice from the perspective of cost effectiveness. When Random ordering is used, coverage granularity is not relevant.

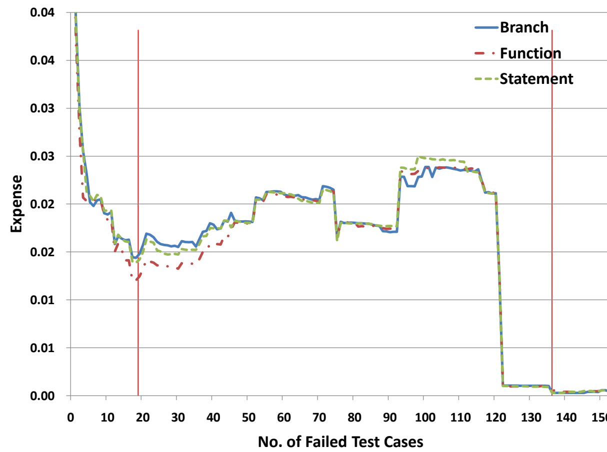

Fig. 9. Comparison of coverage granularity levels on fault localization effectiveness for single-fault UNIX programs.

Table 9. Areas under curve for different coverage granularity levels on single-fault UNIX programs.

Coverage granularity	Branch	Function	Statement
AUC	2.40	2.32	2.40

4.2.1.3 Comparison of distributions of different test case prioritization techniques

In this section, we report the variances of the expenses of using the test suites generated by different test case prioritization techniques for statistical fault localization on single-fault UNIX programs. Table 10 shows the results. Note that a value smaller than 0.001 is represented as "< 10^{-3}" in the table.

Table 10. Variances of expenses on single-fault UNIX programs.

Number of failed test cases	random	ART-br-maxmax	ART-br-maxavg	ART-br-maxmin	ART-fn-maxmax	ART-fn-maxavg	ART-fn-maxmin	ART-st-maxmax	ART-st-maxavg	ART-st-maxmin	addtl-br	addtl-fn	addtl-st	total-br	total-fn	total-st
10	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.003	0.002	0.002	**0.009**	0.006	0.007
20	0.003	0.003	0.003	0.002	0.003	0.003	0.002	0.003	0.003	0.002	0.002	0.002	0.002	0.006	0.004	0.006
30	0.002	0.003	0.003	0.002	0.003	0.003	0.002	0.003	0.003	0.002	0.002	0.002	0.002	0.005	0.004	0.005
40	0.002	0.003	0.003	0.002	0.003	0.003	0.003	0.003	0.003	0.002	0.003	0.002	0.002	**0.006**	0.004	0.005
50	0.003	0.004	0.004	0.003	0.004	0.004	0.003	0.004	0.004	0.003	0.003	0.003	0.003	**0.006**	0.004	0.005
60	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.005	0.005	0.005
70	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.005	0.005	0.005
80	0.004	0.004	0.004	0.003	0.004	0.004	0.003	0.004	0.004	0.003	0.003	0.004	0.003	0.004	0.004	0.004
90	0.004	0.004	0.003	0.003	0.004	0.004	0.003	0.004	0.004	0.003	0.003	0.003	0.003	0.004	0.004	0.004
100	0.003	0.004	0.003	0.003	0.004	0.004	0.003	0.004	0.004	0.003	0.003	0.003	0.003	0.004	0.004	0.004
110	0.004	0.005	0.006	0.004	0.005	0.006	0.004	0.005	0.006	0.004	0.004	0.005	0.004	0.006	0.006	0.006
120	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$
130	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$
140	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$
150	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$
160	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$
Max	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-3}$

We can find that all the 13 Random, ART and Additional techniques are comparable to one another in terms of the variances of the expenses, which are comparably smaller than those of the three Total techniques when the number of failed test cases is small.

If we group different test case prioritization techniques according to strategies as shown in Table 11, we find that for the Random, ART, and Additional strategies, the variances of the expenses are comparable to one another, and all of them have smaller variances than the Total strategy when the number of failed test cases is up to 50. If we classify test case prioritization techniques in terms of coverage granularity as shown in Table 12 (which excludes the Random technique), we find that the variances of the expenses for the Statement, Branch, and Function levels are comparable to one another. As the expense and variance of using a coarse coverage granularity are comparable to those for fine granularities, function coverage is a better choice from the perspective of cost-effectiveness as it incurs less program instrumentation, coverage profile, and test case prioritization overheads.

*4.2.2 Answering RQ*2

Similarly to our analysis shown in Fig. 6 for the Siemens programs, we compute the distributions of expenses in the use of different percentages of a test suite for a commit build of a single-fault UNIX program. The result is shown in Fig. 10. The x-axis and y-axis can be interpreted similar to those of Fig. 6.

Like the results for the Siemens programs, if more test cases are used for a commit build, the expense decreases gradually. For instance, when using 10% of a test suite, the median expense is only 0.21. When using 20% of a test suite, the median expense improves to 0.148. When using the entire test suite for fault localization, it is about 0.04.

We further observe that the variances for single-fault UNIX programs, as indicated by vertical bars, are larger than those for Siemens programs. For example, if 10%, 20%, and 30% of the test cases in a test suite are used for a commit build, the variances are 0.1, 0.15, and 0.2, respectively. They correspond to 2.5, 3.75, and 5 times of the observed variances for the Siemens programs.

We conduct an ANOVA test to validate whether there is no significant difference between the mean expenses of using different percentages of test suite for fault localization at a significance level of 5%. The test returns a p-value of 0.0026, which successfully rejects the null hypothesis and shows that there are significant differences among the groups.

We further conduct multiple mean comparisons on these data. The result, as shown in Fig. 11, can be interpreted similarly to those in Fig. 7. It indicates that there is no statistical difference (at a significance level of 5%) between the expense of using an entire test suite for a commit build and that of using 50% or more of the test suite. This finding is the same as our earlier result for the Siemens suite that we can reduce the number of test case executions by up to 50% in a commit build while maintaining a fault localization effectiveness result as if the whole test suite were used.

Table 11. Variances of expenses for prioritization strategy on single-fault UNIX programs.

Number of failed test cases	Random	ART	Additional	Total
10	0.003	0.003	0.002	**0.007**
20	0.003	0.003	0.002	**0.005**
30	0.002	0.003	0.002	**0.005**
40	0.002	0.003	0.002	**0.005**
50	0.003	0.004	0.003	**0.005**
60	0.004	0.004	0.004	0.004
70	0.004	0.004	0.004	0.004
80	0.004	0.004	0.003	0.004
90	0.004	0.004	0.003	0.004
100	0.003	0.004	0.003	0.004
110	0.004	0.005	0.004	**0.006**
120	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
130	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
140	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
150	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
160	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
Max	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$

Table 12. Variances of expenses for coverage granularity on single-fault UNIX programs.

Number of failed test cases	Branch	Function	Statement
10	**0.0042**	0.0034	0.0036
20	0.0032	0.0028	0.0032
30	0.0030	0.0028	0.0030
40	**0.0036**	0.0032	0.0030
50	**0.0040**	0.0036	0.0038
60	0.0042	0.0042	0.0042
70	0.0042	0.0042	0.0042
80	0.0036	**0.0038**	0.0036
90	0.0034	0.0036	0.0036
100	0.0034	0.0036	0.0036
110	0.0050	**0.0052**	0.0050
120	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
130	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
140	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
150	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
160	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$
Max	$<10^{-3}$	$<10^{-3}$	$<10^{-3}$

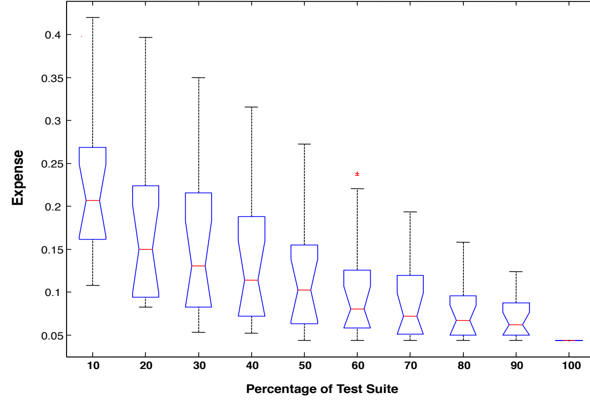

Fig. 10. Distributions of expenses for different percentages of test suite on single-fault UNIX programs.

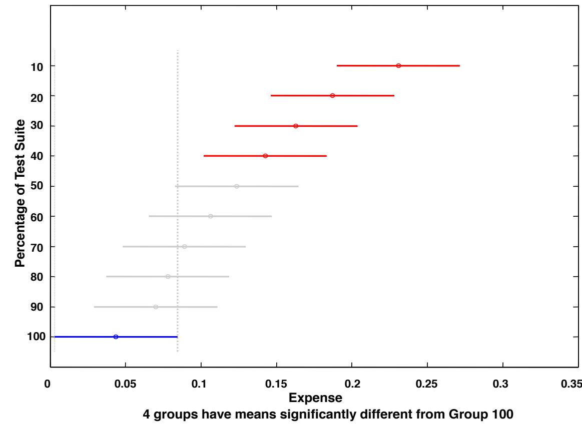

Fig. 11. Multiple comparisons of expenses for different percentages of test suite on single-fault UNIX programs.

4.3 Scalability of multi-fault subjects

In this section, we present the results for multi-fault UNIX programs. To reduce the number of repetitions, for brevity, we directly discuss the results without elaborating again on how to read the corresponding plots and tables, or how to compute the statistics.

*4.3.1 Answering RQ*1

4.3.1.1 Comparison of test case prioritization strategies

From Fig. 12, we observe a trend similar to what we have found on the Siemens programs and single-fault UNIX programs. The variations in the effectiveness of fault localization (in terms of expense) are observed to be consistent in all three kinds of subjects.

We also measure the area under curve of each strategy. The result is shown in Table 13. We find that the Random (2.48), ART (2.60), and Additional (2.45) strategies have comparable fault localization effectiveness, and they all perform better than the Total (3.51) strategy significantly by at least 35%.

4.3.1.2 Comparison of coverage granularity levels

We also report the comparison among different levels of coverage granularity in Fig. 13. The corresponding cumulative fault localization effectiveness at each level is shown in Table 14. We find that different coverage granularity levels do not show significant differences in terms of fault localization effectiveness in Fig. 13, and the difference in terms of AUC among different levels are marginal. Furthermore, an ANOVA test returns a large *p*-value of 0.112, which cannot reject the null hypothesis that various coverage granularities are not significantly different. Thus, when the test case prioritization strategy (other than Random) is fixed, a coarse (function-level) coverage granularity is a better

choice due to its low cost. Random ordering does not, of course, involve coverage information. These results also echo the findings we have observed in previous sections.

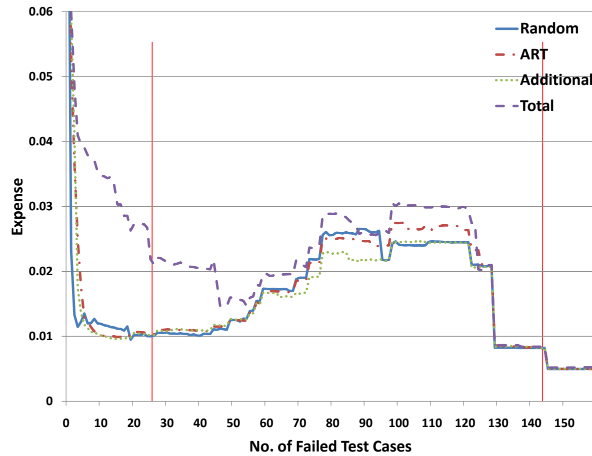

Fig. 12. Comparison of test case prioritization strategies on fault localization effectiveness for multi-fault UNIX programs.

Table 13. Areas under curve for different strategies on multi-fault UNIX programs.

Strategy	Random	ART	Additional	Total
AUC	2.48	2.60	2.45	3.51

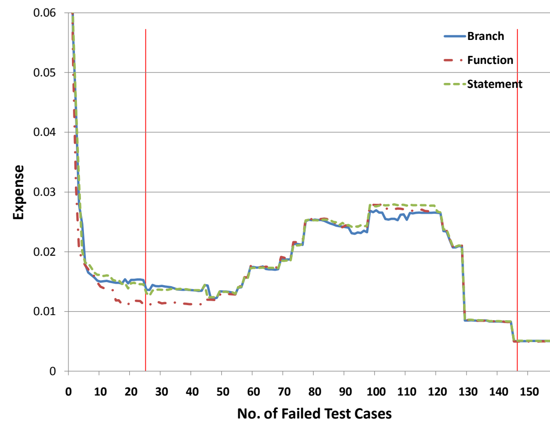

Fig. 13. Comparison of coverage granularity levels on fault localization effectiveness for multi-fault UNIX programs.

Table 14. Areas under curve for different coverage granularity levels on multi-fault UNIX programs.

Coverage granularity	Branch	Function	Statement
AUC	2.77	2.68	2.81

4.3.1.3 Comparison of Distributions of Different Test Case Prioritization Techniques

Table 15 summarizes the variances of the expenses of using the test suites generated by different test case prioritization techniques for statistical fault localization on multi-fault UNIX programs. We find that all the 13 Random, ART, and Additional techniques are close to one another in terms of the variances of their expenses, while the variances for the three Total techniques are comparably larger than the former techniques when the number of failed test cases is small. We should add, of course, that the Random technique is much simpler to implement.

Table 15. Means and variances of expenses on multi-fault UNIX programs.

Number of failed test cases	random	ART-br-maxmax	ART-br-maxavg	ART-br-maxmin	ART-fn-maxmax	ART-fn-maxavg	ART-fn-maxmin	ART-st-maxmax	ART-st-maxavg	ART-st-maxmin	addtl-br	addtl-fn	addtl-st	total-br	total-fn	total-st
10	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.007	0.004	0.007
20	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.007	0.006	0.007
32	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.006	0.005	0.006
40	0.002	0.002	0.002	0.002	0.002	0.003	0.002	0.002	0.003	0.002	0.002	0.002	0.002	0.006	0.004	0.006
50	0.002	0.002	0.003	0.002	0.002	0.003	0.002	0.003	0.003	0.002	0.002	0.002	0.002	0.006	0.006	0.006
60	0.004	0.003	0.004	0.003	0.004	0.004	0.003	0.003	0.004	0.003	0.003	0.003	0.004	0.004	0.004	**0.005**
70	0.004	0.004	0.005	0.004	0.004	0.005	0.004	0.004	0.005	0.004	0.004	0.004	0.004	0.005	0.005	0.005
80	0.005	0.005	0.006	0.004	0.005	0.006	0.004	0.005	0.006	0.005	0.005	0.005	0.004	0.006	0.006	0.006
90	0.005	0.005	0.005	0.004	0.005	0.005	0.005	0.005	0.006	0.005	0.004	0.004	0.004	0.006	0.006	0.005
100	0.005	0.005	0.004	0.004	0.005	0.005	0.005	0.005	0.006	0.004	0.004	0.004	0.005	0.006	0.005	0.005
110	0.005	0.005	0.006	0.005	0.005	0.006	0.005	0.006	0.006	0.005	0.005	0.005	0.005	0.006	0.006	0.006
120	0.005	0.005	0.005	0.004	0.005	0.005	0.005	0.004	0.005	0.005	0.004	0.005	0.005	0.006	0.006	0.006
130	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004	0.004
140	0.003	0.004	0.002	0.003	0.002	0.004	0.003	0.002	0.004	0.003	0.003	0.002	0.004	0.003	0.004	0.003
150	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002
160	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001
Max	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001

Table 16. Variances of expenses for prioritization strategy on multi-fault UNIX programs.

Number of failed test cases	Random	ART	Additional	Total
10	0.002	0.002	0.002	**0.006**
20	0.002	0.002	0.002	**0.007**
30	0.002	0.002	0.002	**0.006**
40	0.002	0.002	0.002	**0.005**
50	0.002	0.002	0.002	**0.006**
60	0.004	0.003	0.003	0.004
70	0.004	0.004	0.004	**0.005**
80	0.005	0.005	0.005	**0.006**
90	0.005	0.005	0.004	**0.006**
100	0.005	0.005	0.004	0.005
110	0.005	0.005	0.005	**0.006**
120	0.005	0.005	0.005	**0.006**
130	0.004	0.004	0.004	0.004
140	0.003	0.003	0.003	0.003
150	0.002	0.002	0.002	0.002
160	0.001	0.001	0.001	0.001
Max	0.001	0.001	0.001	0.001

Similarly to the results on the Siemens and single-fault UNIX programs, Table 16 and Table 17 show the corresponding results of Table 15, which summarize the variances of the expenses of using the test suite generated by different test case prioritization techniques for statistical fault localization on multi-fault UNIX programs at the strategy and coverage granularity levels. From Table 16, we find that the variances of the expenses for the Random, ART, and Additional strategies are close to one another, and all of them have smaller variances than those of the Total strategy when the number of failed test cases is up to 50. From Table 17 (which excludes the Random strategy), we

find that the variances of the expenses for Statement, Branch, and Function levels are comparable to one another, which agrees with the results that we have observed on the Siemens and single-fault UNIX programs.

Table 17. Variances of expenses for coverage granularity on multi-fault UNIXprograms.

Number of failed test cases	Branch	Function	Statement
10	0.003	0.002	0.003
20	0.003	0.003	0.003
30	0.003	0.003	0.003
40	0.003	0.003	0.003
50	0.003	0.003	0.003
60	0.003	0.004	0.004
70	0.004	0.004	0.004
80	0.005	0.005	0.005
90	0.005	0.005	0.005
100	0.005	0.005	0.005
110	0.005	0.005	**0.006**
120	0.005	0.005	0.005
130	0.004	0.004	0.004
140	0.003	0.003	0.003
150	0.002	0.002	0.002
160	0.001	0.001	0.001
Max	0.001	0.001	0.001

*4.3.2 Answering RQ*2

Lastly, for completeness, we report the distributions of expenses on the use of different percentages of a test suite for a commit build. The result is shown in Fig. 14, and that for their multiple mean comparisons is shown in Fig. 15.

We observe that the difference in median expenses between using the entire test suite and using 30% is only 0.003, which is small. We conduct an ANOVA test to check whether there is no significant difference between the mean expenses of using different percentages of test suite for fault localization at a significance level of 5%. However, the ANOVA test returns a p-value of 0.0039, which rejects the null hypothesis, and shows that there are significant differences among the groups. We further perform a multiple-comparison procedure to find out which pairs of techniques differ significantly. Interestingly, the result of multiple comparisons in Fig. 15 shows that there is *no* statistical difference (at a significance level of 5%) between the mean expense of using the whole test suite for a commit build and that of using 30% or more of the test suite. We conjecture that the presence of multiple faults in a program may reduce the effectiveness of locating the first fault among all the faults in the same version. Nonetheless, more studies are required to confirm the conjecture.

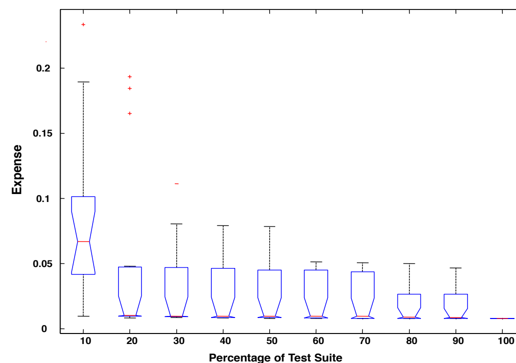

Fig. 14. Distributions of expenses for different percentages of test suite on multi-fault UNIX programs.

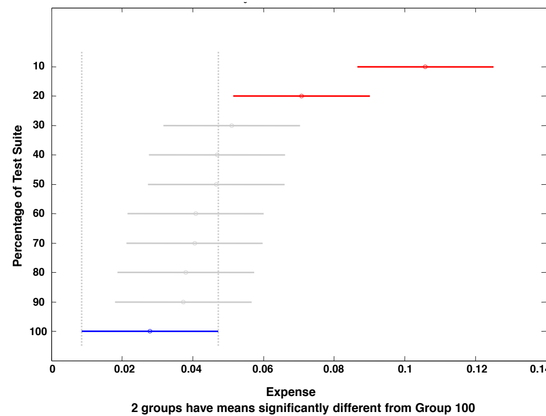

Fig. 15. Multiple comparisons of expenses for different percentages of test suite on multi-fault UNIX programs.

4.4 Summary

In summary, the results of our empirical study provide new information to better configure the continuous integration process to make the best use of test case prioritization techniques and statistical fault localization techniques under different scenarios.

With respect to research question RQ1, we have the following findings: (a) Random ordering can be a cost-effective strategy to reorder test suites for statistical fault localization. It is a low cost prioritization strategy to assist developers to locate faults. However, Random may require executing more test cases to obtain the same number of failed test cases than the other strategies studied. (b) The ART and Additional strategies have quite comparable results with Random ordering in terms of cumulative fault localization effectiveness. If covering different program elements (depending on the coverage granularity) is the prioritization requirement, these two are good strategies. Total is the least effective among the strategies studied. (c) There is no statistically significant difference among various levels of coverage granularity. Intuitively, since the collection of statistics on coarse-grained program elements is less intrusive and more lightweight, it can be more attractive to software projects (except when Random ordering is used, in which case coverage information is not relevant). (d) Last but not least, despite small variations, there is a gradual increase in fault localization effectiveness as the number of failed test cases increases. The underlying reason for the variations remains to be uncovered. In a nutshell, we find that *strategy* can be a factor for fault localization effectiveness, and yet *coverage granularity* is not a significant factor.

With respect to research question RQ2, we find that fault localization effectiveness will be compromised only if a small fraction (≤ 40% in the experiment) of the test suite is used for a commit build. In addition, for all our subject programs, when the scale of a program increases and multi-faults are involved, we observe more savings in terms of the percentage of test cases used in a commit build. Real-life applications are usually large and contain many faults. Our result provides the first piece of empirical evidence that the integration of test case prioritization and fault localization techniques can save efforts in test case execution in a commit build. We find that *time cost* expressed in terms of the number of test cases is a factor for fault localization effectiveness.

Our empirical findings surprisingly show that Random is effective for continuous integration if our objective is to locate faults by using a prioritized test suite with a fixed number of failed test cases. Random can be attractive because it can be efficiently implemented and is non-intrusive. Researchers may wish to study deeper on the role of Random in such integration.

While our study provides empirical findings on the integration of test case prioritization techniques with statistical fault localization, further studies are need to relate such findings to underlying models or theories.

4.5 Threats to validity

4.5.1 Internal validity

To conduct the empirical study, we used many tools, which could have added variability to our results and introduced threats to internal validity. We used several procedures to control potential sources of variation.

All our subject programs and their faulty versions were downloaded from SIR. The fault seeding process used by SIR followed a precise specification so that each program was handled in a similar way [37]. Various previous work [16][17][37] also used the Siemens suite of programs from SIR to study test case prioritization and other problems. We have verified that in our experiment, each version produced the same test verdicts as indicated in the fault matrix files provided by SIR.

We evaluated four fault localization techniques and 16 test case prioritization techniques in our empirical study. We chose these specific techniques because they have been studied in recent regression testing and fault localization papers. They are chosen because they are representative.

When analyzing the results of the empirical study, we chose to group the subjects into small-sized programs, medium-sized single fault programs, and medium-sized multi-fault programs rather than discussing each subject program individually. Our rationale is an attempt to provide a useful summary rather than overwhelming readers with unnecessary details. We also carefully verified the results of every subject program to ensure that they are consistent with those of the overall group.

We carefully tested our regression testing tools, which were also used in Jiang and Chan [23]. The result for ART-based techniques can be repeated by the experiment presented in [25]. We also systematically tested our fault localization tools, which were also used in Zhang et al. [47][48][48]. More specifically, we tested our tools on the SIR subjects and compared the results with the fault matrix provided by SIR. Our tools produced exactly the same fault matrix as that from SIR. To evaluate Tarantula, CBI, Jaccard similarity coefficient, and Ochiai, we carefully studied their papers on the technical details and implemented them on our platform. For the ranking results produced by a fault localization technique, we spot-checked them manually to verify the correctness of the implementation. To conduct ANOVA test and multiple comparisons, we used MATLAB rather than any toolset of our own. This is because MATLAB's statistics package is mature and tested by many users to be reliable. We also made use of gcov, a popular code coverage collection tool used with gcc. The gcov tool has been demonstrated to work well by many researchers and software engineers. We have also double checked the reported figures. We use an unweighted average to aggregate the measured value from the outputs of the techniques of the same dimension because they have the same probability to happen in our experiment setting. Using other aggregation formulas may obtain different findings.

One important component of our study was how to use a prioritized test suite for fault localization. In other words, how could we define the interface between test case prioritization and fault localization? To do it correctly, we obtained a prioritized test suite generated by a test case prioritization technique, collected the execution profile of different percentages (or numbers) of test cases within the test suite to emulate the commit build process, and fed the execution profile to each fault localization technique. We carefully verified the results with invariant properties to ensure correctness. We also manually worked out small examples to check whether our implementations output the results correctly.

4.5.2 External validity

There were several issues that may affect the external validity of our experiment. The first issue was concerned with the subject programs studied.

We used the Siemens and UNIX programs as subjects in the study. They ranged from small scale to real-life programs with single and multiple faults. While the Siemens programs were small in scale, all the subject UNIX programs were real-life programs with 4756 to 10,124 statements. We used them to study the integration problem between test case prioritization and statistical fault localization because they were also used by other studies on regression testing and statistical fault localization [1][16][17][23][25][27][31][34][40][47][48][50]. Nonetheless, compared to the scale of software development today, the programs are still relatively small. Case studies of industrial-scale programs would help to verify the results further.

Another potential threat to validity for the Siemens and UNIX programs is that seeded faults were the only modifications between the original versions and the faulty versions. In real life, code evolution by developers may

include both benign and faulty modifications. Since benign and faulty modifications may interact with each other, a further study on subject programs with both kinds of changes may further strengthen the validity of our study.

KLEE [4] has demonstrated that it is feasible to generate test cases automatically to achieve high branch coverage for selected industrial-strength and complex programs. In general, not all test suites may fully satisfy a specific test adequacy criterion. We used the test suites already developed for Siemens programs because all the test suites are based on the branch coverage criterion. The use of other test case coverage criteria to construct test cases for regression testing is also an interesting topic for future study.

We only chose C programs in our empirical study because they are still widely used in many real-life applications such as Web servers, UNIX tools, and database servers. A further investigation on subject programs written in other programming languages may help generalize our findings.

The current results show that if a tester starts debugging after a given number of failed test cases have been observed, Random may, on average, generate a more effective test suite for fault localization than the techniques under each of the other strategies studied. There are many interesting follow-up questions: First, it is unclear (to us) how the current results can be generalized if the tester chooses to start debugging after (i) a given number of test cases (regardless of whether they have failed) have been executed, (ii) a given test adequacy criterion has been fulfilled, (iii) some criterion that is determined not only by the test results has been satisfied, or (iii) other feedback related to fault prediction has been obtained. Second, the current study has only explored the strategy dimension (which is at an aggregated level of the underlying techniques). Future research may be conducted to examine the effect of, say, one particular prioritization technique coupled with one particular fault localization technique. While it will be a valuable option, the result of such research may have even more significant threats to external validity than the present study, because there is a vast number of test case prioritization techniques and fault localization techniques proposed in the literature. At the other extreme, a further study covering more techniques will surely alleviate the threats to external validity of our study.

4.5.3 Construct validity

APFD was originally proposed to measure the fault detection rates of test case prioritization techniques. In this paper, however, we focused on the integration of test case prioritization and fault localization, whose efficacy was demonstrated by debugging effectiveness. As a result, we used *expense* as the primary metric in our empirical study. It was widely adopted in many previous studies on fault localization techniques. This is because it corresponds to a typical way software engineers use fault localization techniques, by examining the individual entries in a ranked list from top to bottom until they find a faulty statement. On the other hand, we are aware that software engineers may also use the ranked list of suspicious statements in various other ways. For example, when developers examine a particular statement $s1$, they may happen to spot problems in another statement $s2$ close to $s1$. Statement $s2$ may have a much lower rank, however. This indicates that more code or less code may be examined to locate faults than what an effectiveness metric may indicate. In any case, a study of the behavior of software engineers on the use of ranked list is beyond the scope of this paper.

They were also other studies [13][34][36] that used T-score to evaluate the effectiveness of fault localization techniques. We did not use T-score because it was a metric for predicate-based fault localization techniques rather than statement-based fault localization techniques. Furthermore, T-score relies heavily on the assumption of "an ideal programmer who is able to distinguish defects from non-defects at each location, and can do at the same cost for each location considered" [13]. As a result, we used *expense* rather than T-score to evaluate the effectiveness of fault localization techniques.

We studied the impact of using different portions of a reordered test suite on fault localization for the integration problem. Therefore, we also wanted to know the overall effectiveness of a strategy. We thus used area under curve to compute the cumulative *expense* for the full range of a test suite for fault localization. The use of more metrics may help strengthen the study.

5. Related work

This section reviews projects that are related to our work. We firstly review the work on test case prioritization and fault localization, followed by their integration.

Our work is related to studies on the time-cost dimension of test case prioritization. In our empirical study, we used the ART [25], Additional [16], and Total [16] strategies to prioritize a test suite based on code coverage information obtained from the executions of the test suite over previous versions of a program. Researchers have proposed a number of variants of these basic strategies. For instance, a binary matching technique was proposed by Srivastava and Thiagarajan [38] to identify program modifications between versions, and reorder test cases to optimally cover such modifications.

However, such optimization can be inefficient. Walcott et al. [39] adopted a genetic algorithm approach, which usually runs fast, to propose a time-aware test case prioritization technique that generates approximations to an optimized permutation efficiently. Li et al. [31] empirically evaluated various search-based algorithms for test case prioritization in a systematic way. However, they concluded from their experiments that meta-heuristics approaches to optimizing the rate of coverage might not outperform the greedy algorithms proposed by Elbaum et al. [16]. Zhang et al. [46] used integer linear programming to optimize time-aware test case prioritization. They found that such coverage-based optimization could help improve the fault detection rate for selected Java programs. You et al. [43] evaluated time-aware test case prioritization on the Siemens suite and the program *space*. They found that the differences among techniques in terms of AFPD were not statistically significant. Li et al. [30] further showed that the Additional strategy and the 2-optimal greedy algorithms could be more effective than generic hill-climbing algorithms and the Total strategy. Based on previous results, therefore, we chose to use the Additional strategy [16] in our experiment to study this time-related dimension, rather than using a meta-heuristics approach. We also included the Total strategy [16] in our experiment because it is often used in pair with the Additional strategy.

Jiang et al. [25] showed that an ART-based approach to test case prioritization can be slightly less effective but more scalable than the Additional strategy. We note that even without a comparison with a genetic algorithm such as those proposed in [30], the result is still valid. Their experiments found that coverage granularity is not a consistent factor that affects the goal of detecting failures as early as possible, which is consistent with our finding on fault localization that coverage granularity is not a factor that affects fault localization effectiveness significantly. Thus, our result is consistent with their observation, even though we focus on fault localization rates. Rather than using the Jaccard distance as in Jiang et al., Zhou [51] uses the Manhattan similarity coefficient to measure the test case differences from the codebase perspective.

The related work above differed from the present paper in that their techniques primarily focused on the goal of increasing the rate of fault detection or the rate of code coverage, while our study focuses on the goal of increasing the fault localization rate primarily measured by *expense* [27][48]. Our target is not to measure the effectiveness of test case prioritization techniques, but whether test case prioritization may affect the use of statistical fault localization techniques to help developers locate faults.

All the ART, Additional, and Total techniques in our study prioritize a test suite based on code coverage information obtained from the executions of the test suite using previous versions of the program. However, the code coverage information for different versions may be different, which may especially affect the effectiveness of the greedy techniques for test suite prioritization. Chittimalli and Harrold [9] proposed techniques that use algorithms for regression test selection to compute accurate, up-to-date coverage data on a modified version of the software without having to rerun test cases that do not execute the modified statements. Zhang et al. [45] proposed a regression test selection technique that manages to choose a subset of an original test suite while maintaining high precision and recall rates. They proposed to use a clustering approach to achieve this objective. Leon et al. [29] proposed failure-pursuit techniques for test case generation and prioritization. Their techniques first cluster test cases based on inputs and then use sampling to select an initial sample. If a failure is revealed by any test case in the initial sample, its *k* nearest neighbors are selected and checked. If additional failures are found, the process will be repeated. This is an online technique that uses execution results to guide the selection of follow-up test cases. Yan et al. [42] proposed a new sampling strategy to select test cases. One may also adapt their idea to conduct online test case prioritization to support more statistical significant execution information. However, the degree of fault localization support is unclear. Moreover, in the preliminary version of this paper [26], we showed that such techniques could be more sensitive than random ordering. We therefore did not include such clustering-based techniques in our empirical study.

To support debugging, researchers have proposed many individual techniques. Delta Debugging [13] automatically isolated failure-inducing inputs, produced cause-effect chains, and identified the faults. It located faults by systematically manipulating the input to isolate the minimum failure inducing input, and analyzing when an erroneous program state is produced and propagated to the final output. Since their techniques changed the test cases

systematically, it would be hard to directly integrate it with test case prioritization techniques. Renieris and Reiss [36] used the difference in the execution traces between a failed run and its nearest neighboring passed run for fault localization. Jeffrey et al. [22] proposed a value replacement approach to ranking program statements according to their suspiciousness of being faulty. Zhang et al. [48] proposed to represent passed executions and failed executions using edge profiles, and compared them in order to model how each basic block contributes to failures and how the infected program states propagate along control flow edges. Wong et al. [41] proposed a family of code-coverage-based heuristics for fault localization. Abreu et al. [2] also proposed to combine spectrum-based fault localization (SFL, which correlate failures with abstractions of program traces) and model-based fault diagnosis to support fault localization in programs with multiple faults. Their techniques used ideas similar to the four techniques studied in this paper. Zhang et al. [50] proposed to conduct fault localization at evaluation sequence level to take into consideration the short-circuit evaluations of individual predicates. Zhang et al. [48] also studied whether the feature spectra of program elements can be safely considered as normal distributions, so that parametric fault localization techniques can be soundly and powerfully applied. These techniques used different ranking formulas in statistical analyses to rank the faulty statements. Since our study is an integration of test case prioritization techniques and fault localization techniques, the experiment may grow exponentially if we evaluate more fault localization techniques. We have therefore focused on four well-studied fault localization techniques to make the empirical study both comparable to previous work and manageable.

Yu et al. [44] studied the effect of applying different *test suite reduction* techniques on the effectiveness of fault localization. In regression testing research, test suite reduction and test case prioritization are two different categories of techniques. On the other hand, at an abstract level, both of them can select a subset of test cases from a test suite, and can therefore be considered as test case selection approaches. However, applying every technique studied by Yu et al. on a pair of a test suite and a program generates exactly one subset of the test suite. Owing to the redundancy elimination nature of every such test suite reduction technique, in general, such a technique cannot generate two subsets, say *A* and *B*, from the same pair of test suite and program such that *A* is a proper superset of *B*. Moreover, each technique also has an assumption of the availability of a coverage adequacy criterion. Test case prioritization has neither of these limitations. Without such limitations, we can explore in this paper the effects of adding or removing test cases to fault localization effectiveness at a level of detail unprecedented in previous studies.

Wong et al. [40] proposed to select test cases based on a ratio of cost to incremental coverage. Their approach combined both test suite minimization and prioritization techniques. Baudry et al. [3] used a bacteriologic approach to create test suites that aim at maximizing the number of dynamic basic blocks and use the ranking algorithm in Jones and Harrold [27] to conduct fault localization. It would be interesting to study how test case generation techniques may support fault localization. Gonzalez-Sanchez et al. [20][21] proposed to directly incorporate fault localization estimations into test case prioritization techniques. They did not study the factors of test case prioritization techniques that might affect the integration. In our previous work [23], we reported a postmortem analysis of the support by test case prioritization strategies on the capability of fault localization. Different from the current study, our previous work first set a threshold effectiveness value for the metric *expense* to select criterion-adequate test suites. It then conducted a postmortem analysis on the test case prioritization strategies that may generate these test suites.

Adaptive random testing [5][6][8] improved the effectiveness of random testing in detecting the first failure by recommending diversity guidelines to the test case generation process. Chen and Merkel [7] proposed the use of quasi-random sequences for testing in a high-dimensional input space. Ciupa et al. [10][11][12] investigated how to define distance among objects for ART. Their experimental results showed that ART based on object distance can significantly increase the fault detection rate for object-oriented programs. Chen et al. [5] further proposed a more general category and choice method of using ART in non-numeric applications. They have not studied test case prioritization. However, these distance measures and the idea of handling higher dimensional inputs may be used to enhance adaptive test case prioritization [25].

6. Conclusion and future work

In this paper, we have examined the integration of test case prioritization and fault localization techniques. Our primary interest is motivated by whether such an integration supports commit build in continuous integration. We have conducted and reported the first empirical study in this area, and have examined three factors, namely strategy, coverage granularity, and time cost. Our empirical study has used 11 subject programs, including the Siemens suite as

well as single-fault and multi-fault UNIX programs. It has involved 16 test case prioritization techniques, four statistical fault localization techniques, tens of thousands test cases, and different percentages of test suites. The empirical results have concluded that strategy and time cost of test case prioritization techniques can be factors that affect the effectiveness of statistical fault localization techniques, while there is no statistical evidence to conclude that coverage granularity is a factor. Last but not least, we have found that the addition of more failed test cases will generally improve the fault localization effectiveness in such integration, and yet there are variations (namely, deteriorations in fault localization effectiveness) in terms of expenses during the process. In other words, statistical fault localization can be fully effective only after sufficient failed test cases have been added to get over such variation periods.

Based on the empirical results, we have the following interpretations: If testers want to stop regression testing of the commit build after a certain number of failed test cases have been observed, the test suites produced by Random ordering can be a cost-effective option to integrate with (existing) statistical/spectrum-based fault localization techniques. The ART and Additional strategies have comparable fault localization effectiveness as Random ordering. The Total test case prioritization techniques are the least effective in ordering test suites for effective statistical fault localization. We have also found that different coverage granularity levels do not result in significant differences in supporting effective fault localization by the studied techniques. If testers have a resource concern in terms of the number of test cases for regression testing, the use of test case prioritization can save up to 40% of test case executions for commit builds without significantly deteriorating the fault localization effectiveness from the statistics point of view. The finding shows that such savings are more noticeable on medium-sized programs than small-scale programs, and more on multi-fault programs than single-fault programs.

For future work, it is interesting to study how to achieve a tighter integration between regression testing and debugging techniques. We also want to further study test case generation techniques and test case adequacy selection criteria that enable the original test suite to provide better support for effective fault localization. We would also like to know why there is a variation of fault localization effectiveness after the addition of failed test cases. Our study only reveals some empirical findings in this integration process. The answers on why the integration leads to such empirical findings are still open to further investigation.

In previous work, random ordering was reported to be slow in revealing faults when compared with Greedy and ART. In studying RQ1, we set the scenario that testers start debugging on observing a given number of failed test cases. Random probably requires more test cases than either Greedy or ART to reveal the same number of faults. Moreover, statistical fault localization techniques can be more effective if they work on a larger test suite than a smaller one. However, our finding on RQ1 also shows that even for the same strategy (such as Additional), there is a deterioration period in which the addition of more test cases worsens the average fault localization effectiveness. Based on such findings, we conjecture that the effect of failed test cases is more significant than the size of a test suite. We suggest studying these two observations in more detail in the future. Fault detection capability of a test suite is also closely related to the concept of test adequacy. Hence, for RQ2, we suggest further studies on the effect of test adequacy. In addition, special attention should be given to marginal savings rates rather than simply on whether there are savings if less test cases are applied.

Acknowledgments

This research is supported in part by a grant of the Basic Research Fund of Beihang University, a grant of the Natural Science Foundation of China (Project No. 61003027), a strategy research grant of City University of Hong Kong (Project No. 7002673), a grant of the General Research Fund of the Research Grants Council of Hong Kong (Project No. 717308), and a discovery grant of the Australian Research Council (Project No. DP120104773).

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, A.J.C. van Gemund, A practical evaluation of spectrum-based fault localization, Journal of Systems and Software 82 (11) (2009) 1780–1792.

[2] R. Abreu, P. Zoeteweij, A.J.C. van Gemund, Spectrum-based multiple fault localization, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), IEEE Computer Society Press, Los Alamitos, CA, 2009, pp. 88–99.

[3] B. Baudry, F. Fleurey, Y. Le Traon, Improving test suites for efficient fault localization, in: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), ACM Press, New York, NY, 2006, pp. 82–91.

[4] C. Cadar, D. Dunbar, D.R. Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008), USENIX Association, Berkeley, CA, 2008, pp. 209–224.

[5] T.Y. Chen, F.-C. Kuo, R.G. Merkel, T.H. Tse, Adaptive random testing: the ART of test case diversity, Journal of Systems and Software 83 (1) (2010) 60–66.

[6] T.Y. Chen, H. Leung, I.K. Mak, Adaptive random testing, in: Advances in Computer Science: Proceedings of the 9th Asian Computing Science Conference (ASIAN 2004), Lecture Notes in Computer Science, vol. 3321, Springer, Berlin, Germany, 2004, pp. 320–329.

[7] T.Y. Chen, R.G. Merkel, Quasi-random testing, IEEE Transactions on Reliability 56 (3) (2007) 562–568.

[8] T.Y. Chen, R.G. Merkel, An upper bound on software testing effectiveness, ACM Transactions on Software Engineering and Methodology 17 (3) (2008) 1–27.

[9] P.K. Chittimalli, M.J. Harrold, Recomputing coverage information to assist regression testing, IEEE Transactions on Software Engineering 35 (4) (2009) 452–469.

[10] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, Object distance and its application to adaptive random testing of object-oriented programs, in: Proceedings of the 1st International Workshop on Random Testing (in conjunction with the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006)), ACM Press, New York, NY, 2006, pp. 55–63.

[11] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, Experimental assessment of random testing for object-oriented software, in: Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007), ACM Press, New York, NY, 2007, pp. 84–94.

[12] I. Ciupa, A. Leitner, M. Oriol, B. Meyer, ARTOO: adaptive random testing for object-oriented software, in: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), ACM Press, New York, NY, 2008, pp. 71–80.

[13] H. Cleve, A. Zeller, Locating causes of program failures, in: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), ACM Press, New York, NY, 2005, pp. 342–351.

[14] H. Do, G. Rothermel, On the use of mutation faults in empirical assessments of test case prioritization techniques, IEEE Transactions on Software Engineering 32 (9) (2006) 733–752.

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

[16] S.G. Elbaum, A.G. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Transactions on Software Engineering 28 (2) (2002) 159–182.

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

[18] D. Farley, The deployment pipeline: extending the range of continuous integration <http://www.scribd.com/doc/196739/The-Deployment-Pipeline-by-Dave-Farley-2007>, 2007.

[19] M. Fowler, Continuous integration <http://www.martinfowler.com/articles/continuousIntegration.html>, 2006.

[20] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, A. van Gemund, A diagnostic approach to test prioritization, Technical Report TUD-SERG-2010-007, Software Engineering Research Group, Delft University of Technology, Delft, the Netherlands, 2010.

[21] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, A.J.C. van Gemund, Prioritizing tests for software fault localization, in: Proceedings of the 10th International Conference on Quality Software (QSIC 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 42–51.

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

[23] B. Jiang, W.K. Chan, On the integration of test adequacy: test case prioritization and statistical fault localization, The 1st International Workshop on Program Debugging in China (IWPDC 2010), in: Proceedings of the 10th

International Conference on Quality Software (QSIC 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 377–384.

[24] B. Jiang, T.H. Tse, W. Grieskamp, N. Kicillof, Y. Cao, X. Li, Regression testing process improvement for specification evolution of real-world protocol software, in: Proceedings of the 10th International Conference on Quality Software (QSIC 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 62–71.

[25] B. Jiang, Z. Zhang, W.K. Chan, T.H. Tse, Adaptive random test case prioritization, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009), IEEE Computer Society Press, Los Alamitos, CA, 2009, pp. 233–244.

[26] B. Jiang, Z. Zhang, T.H. Tse, T.Y. Chen, How well do test case prioritization techniques support statistical fault localization, in: Proceedings of the 33rd Annual International Computer Software and Applications Conference (COMPSAC 2009), vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 2009, pp. 99–106.

[27] J.A. Jones, 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), ACM Press, New York, NY, 2005, pp. 273–282.

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

[29] D. Leon, W. Masri, 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), ACM Press, New York, NY, 2005, pp. 412–421.

[30] S. Li, N. Bian, Z. Chen, D. You, Y. He, A simulation study on some search algorithms for regression test case prioritization, in: Proceedings of the 10th International Conference on Quality Software (QSIC 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 72–81.

[31] Z. Li, M. Harman, R.M. Hierons, Search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (4) (2007) 225–237.

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

[33] A.H. Lipkus, A proof of the triangle inequality for the Tanimoto distance, Journal of Mathematical Chemistry 26(1) (1999) 263–265.

[34] C. Liu, X. Yan, L. Fei, J. Han, 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 Foundations of Software Engineering (ESEC 2005/FSE-13), ACM Press, New York, NY, 2005, pp. 286–295.

[35] L. Naish, H.J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on Software Engineering and Methodology 20 (3): article no. 11 (2010).

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

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

[38] A. Srivastava, J. Thiagarajan, Effectively prioritizing tests in development environment, in: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), ACM Press, New York, NY, 2002, pp. 97–106.

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

[40] W.E. Wong, V. Debroy, B. Choi, A family of code coverage-based heuristics for effective fault localization, Journal of Systems and Software 83 (2) (2010) 188–208.

[41] W.E. Wong, J.R. Horgan, S. London, H. Agrawal, A study of effective regression testing in practice, in: Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE 1997), IEEE Computer Society Press, Los Alamitos, CA, 1997, 264–274.

[42] S. Yan, Z. Chen, Z. Zhao, C. Zhang, Y. Zhou, A dynamic test cluster sampling strategy by leveraging execution spectra information, in: Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation (ICST 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 147–154.

[43] D. You, Z. Chen, B. Xu, B. Luo, C. Zhang, An empirical study on the effectiveness of time-aware test case prioritization techniques, in: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC 2011), ACM Press, New York, NY, 2011.

[44] Y. Yu, J.A. Jones, 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), ACM Press, New York, NY, 2008, pp. 201–210.

[45] C. Zhang, Z. Chen, Z. Zhao, S. Yan, J. Zhang, B. Xu, An improved regression test selection technique by clustering execution profiles, in: Proceedings of the 10th International Conference on Quality Software (QSIC 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 171–179.

[46] L. Zhang, S.-S. Hou, C. Guo, T. Xie, H. Mei, Time-aware test-case prioritization using integer linear programming, in: Proceedings of the 2009 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009), ACM Press, New York, NY, 2009, pp. 213–224.

[47] Z. Zhang, W.K. Chan, T.H. Tse, P. Hu, X. Wang, Is non-parametric hypothesis testing model robust for statistical fault localization?, Information and Software Technology 51 (11) (2009) 1573–1585.

[48] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, 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 Foundations of Software Engineering (ESEC 2009/FSE-17), ACM Press, New York, NY, 2009, pp. 43–52.

[49] Z. Zhang, W.K. Chan, T.H. Tse, Y.T. Yu, P. Hu, Non-parametric statistical fault localization, Journal of Systems and Software 84 (6) (2011) 885–905.

[50] Z. Zhang, B. Jiang, W.K. Chan, T.H. Tse, X. Wang, Fault localization through evaluation sequences, Journal of Systems and Software 83 (2) (2010) 174–187.

[51] Z.Q. Zhou, Using coverage information to guide test case selection in adaptive random testing, The 7th International Workshop on Software Cybernetics (IWSC 2010), in: Proceedings of the 34th Annual Computer Software and Applications Conference (COMPSAC 2010), IEEE Computer Society Press, Los Alamitos, CA, 2010, pp. 208–213.