USING MUTATION IN FAULT LOCALIZATION

Chenglong Sun and Tian Huang

Institute of Software, Chinese Academy of Sciences, Beijing, China
suncl@ios.ac.cn huangt@ios.ac.cn

ABSTRACT

Fault localization is time-consuming and difficult, which makes it the bottleneck of the debugging progress. To help facilitate this task, there exist many fault localization techniques that help narrow down the region of the suspicious code in a program. Better accuracy in fault localization is achieved from heavy computation cost. Fault localization techniques that can effectively locate faults also manifest slow response rate. In this paper, we promote the use of pre-computing to distribute the time-intensive computations to the idle period of coding phase, in order to speed up such techniques and achieve both low-cost and high accuracy. We raise the research problems of finding suitable techniques that can be pre-computed and adapt it to the pre-computing paradigm in a continuous integration environment. Further, we use an existing fault localization technique to demonstrate our research exploration, and shows visions and challenges of the related methodologies.

KEYWORDS

Software testing, fault localization, continuous integration, mutation testing

1. INTRODUCTION

Software serves every corner of our lives. Nevertheless, software failures nowadays are still common and it is extremely difficult to thoroughly avoid software failures, even in final releases of software. Most software failures are caused by mistakes made by programmer in coding and debugging is an important activity in software engineering with the intent of locating and removing faults from programs. Conventionally, a debugging process involves three tasks, fault localization, fault repair and retesting of the revised program. Among them, fault localization is the most difficult and time-consuming task, and is often a bottleneck in the debugging process.

To facilitate the fault localization task, automatic fault localization techniques have been invented. Generally speaking, these techniques automate the identification of the suspicious code that may contain program faults. Tarantula [1] calculates the ratio of passed runs and ratio of failed runs that executes a program statement, and estimates the suspiciousness of that program statement correlated with program failures. A ranked list of program statements is constructed in descending order of the estimated suspiciousness of each statement. These kind of techniques, which contrasts the program spectra of passed and failed runs to predicate the fault relevance of individual program entities, is called coverage-based fault localization (CBFL).

CBFL has received much attention due to its applicability and is a popular research area on the exploration of effective evaluation formulas to assess the suspiciousness of program entities. The techniques in CBFL are of low-cost at ranking program entities because only testing results and program spectrum for software testing are taken into account [2][8]. A typical technique in CBFL first selects a set of program entities, and then collects the execution statistics of them for both passed and failed runs. By contrasting the similarities between two such sets of statistics for each entity, it estimates the extents of the program entities correlating to faults, and ranks the program entities accordingly.

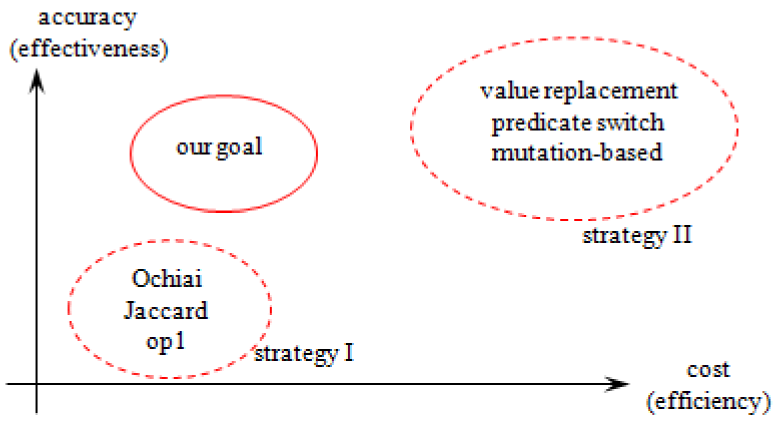


Figure 1. The cost and accuracy of different fault localization techniques

Apart from the lightweight CBFL techniques, there are many approaches aiming for high accuracy at the expense of high cost. Value replacement [10] searches for interesting value mapping pair, in which a corresponding replacement at a program site during the execution of a failing run results in correct output. Using such value mapping pairs, program statements are ranked in their likelihood of being faulty. Mutation-based fault localization techniques [7] mutate statements in programs systematically and estimate their likelihood of being faulty based on both coverage and how mutations affect the outcome of test runs. Comparing with CBFL techniques, such techniques make use of additional information like variable values, and experiments show significant improvements in fault localization effectiveness over the former.

Figure 1 illustrates the cost and accuracy differences of the above two strategies. Techniques following strategy I (CBFL techniques like Jaccard [1]) have low computation cost and relatively low locating accuracy. Techniques following strategy II have relatively high computation cost and high locating accuracy. Aiming for both low-cost and remarkable accuracy, in this paper, we integrate properties of the above two strategies to propose a low-cost and high-accuracy fault-localization direction.

Generally speaking, remarkable accuracy comes from sufficient computation to capture enough useful information to facilitate fault localization. As a result, we cannot reduce necessary computation but instead use pre-computing to move the time-intensive computing task at the testing moment to the code phase [11]. Thus, we can rapidly return fault localization results. This is the research problem proposed in this paper. We propose a two-step strategy to solve this problem: 1) To find fault localization techniques, which accuracy can be reserved by pre-computing; 2) To adapt fault localization techniques to the continuous integration scenario to get rapid response.

The contributions of this paper have at least two aspects. First, we propose a pre-computing paradigm in a continuous integration environment in order to effectively and efficiently run existing fault localization techniques. Second, we illustrate our preliminary idea by adapting an existing high-accuracy fault localization technique in the proposed paradigm.

The rest of this paper is organized as follows. Section II reviews related work. Section III motivates our work and Section IV explores this research direction, and shows visions and challenges of the research. Section V concludes this paper.

2. RELATED WORK

Coverage-based fault-localization (CBFL) techniques are one category of the most representative practices of fault localization. Jones et al. proposed Tarantula, which uses the propositions of passed and failed executions to calculate the suspiciousness of a statement to be faulty. Other fault localization techniques include Jaccard [1], Ochiai2 [3] and so on. These techniques are similar to Tarantula except that they use different formulas to compute the suspiciousness of statements. Apart from these statement-level techniques, there are many predicate-based techniques. Liblit et al. [4] developed CBI, which uses the number of times a predicate being evaluated true in passed and failed runs along with the specificity and sensitivity of the predicates in order to estimate the suspiciousness of predicates. SOBER [5] compares the distributions of evaluation biases between passed runs and failed runs to compute the suspiciousness of predicates.

Theoretically, Naish [6] showed that the optimal effectiveness of CBFL techniques is relatively not higher than that of Op1. However, the effectiveness of Op1 may not always be satisfactory. For example, in some programs, examining 200 lines of code to locate a fault can be too much for a programmer [6].

In order to achieve high accuracy in fault localization, additional information is referenced. X. Zhang et al. [9] proposed to forcibly switch a predicate's outcome at runtime and alter the control flow to estimate the position of fault statement, to accurately locate fault. Jeffrey et al. [3] proposed a value replacement technique, by looking at data-flow information and so on, which computational complexity is even higher. Papadakis and Le-Traon [7] proposed a mutation-based fault diagnosis approach, which uses mutation operators to generate different program versions, monitor testing outcome of them, and use the program version having most similar performance with the version in hand to estimate a fault's location.

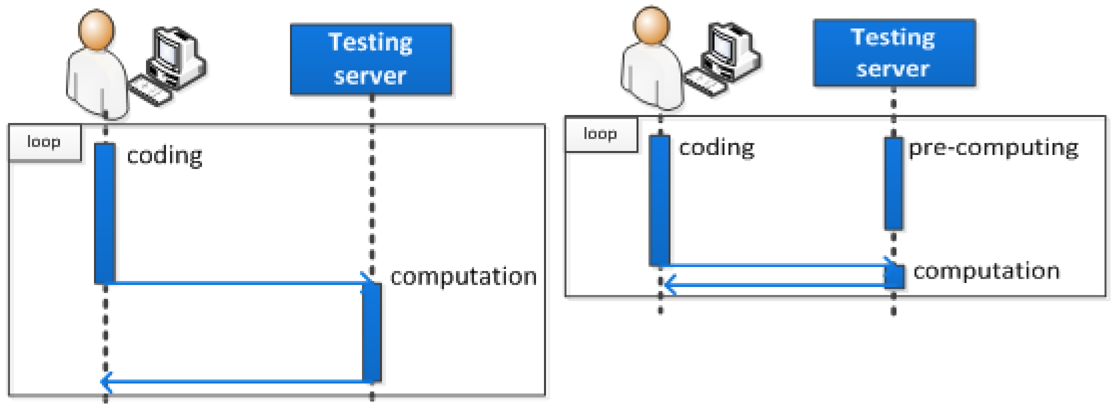
Generally speaking, high accuracy acquired in the above approaches means high computation costs. For example, the value replacement technique [7] is quite slow and its IVMP searching process may last for several hours. Such slow responses are unacceptable in practice. In this paper, we plan to use pre-computing to distribute heavy computation to the idle phase, to create high accuracy and low-cost fault localization approaches.

3. MOTIVATION

We revisit different fault localization techniques and quantify their cost and effectiveness in Table I. We collect data both from the reported result from the original paper of the mentioned techniques and from our experiment. We use the running time of a technique to output a fault localization result to measure its computation cost, and use the percentage of faults located in a date set within 5% of code examination efforts in each program to measure the accuracy of the fault localization technique.

Table 1. The cost and accuracy of different fault localization techniques (quantified)

	Avg. running time (seconds)	**Avg. accuracy (% of faults located when examining 5% of code)**
Group I / Strategy I (e.g., CBFL) (low-cost, low accuracy)	< 5	< 30%
Group II / Strategy II (e.g., value replacement) (high-cost, high accuracy)	> 500	> 60%



a) Convectional process (heave computation and slow response)

b) Pre-computing approach (computation distributed, rapid response)

Figure 2. Using pre-computing to enable rapid response to fault localization requests in the CI environment [11]

Our observations suggest that techniques are typically grouped into two categories. CBFL techniques (Group I) like Tarantula are time-saving but of relatively coarse accuracy. The utilization of these techniques simply depends on the program spectra and testing results, and the information can be easily acquired dynamically from software testing. They need less than five seconds on average in running time to produce a fault localization result, and can locate less than 30% of all faults when examining up to 5% of the code. In detail, the time cost of Tarantula is 0.04 seconds, and the measured accuracy is 47.15%, proportion of localized faults. On the other hand, the program state-based techniques (Group II) such as mutation are more accurate in locating faults but are also more time-consuming. In detail, the running time of [7] is 40.02 minutes in average and the measured accuracy is 60.16%.

It is natural to think about whether we can synthesize fault localization techniques that are both effective and efficient. Since high accuracy comes from sufficient computation. We notice that fault localization starts when a testing request is triggered. To wait for a heavy computation to finish, the response rate becomes too low to accept, for techniques like [7]. We therefore want to use pre-computing to provide rapid response rate, so that both high accuracy and seemingly low-cost can be achieved.

Figure 2 shows our basic idea. Plot (a) shows the original sequence process of the conventional testing paradigm in a continuous integration environment. The program evolves from a long coding phase and finally converges to a stable version. During testing, the program is run against an equipped test suite to collect the coverage information and test results. When failed runs appear, faults in the program are confirmed. Coverage information is input to a testing server running a fault localization service. After a complicated and long computing process, an accurate fault localization result is generated.

From plot (a), we observe that in the coding phase, testing server is always idle, and at the moment a testing request is triggered, the testing server is fully occupied and the programmer is waiting. If we could distribute the computation into idle period, the waiting time for a programmer to collect fault localization results can be shortened. Plot (b) of Figure 2 illustrates the idea. During coding, we want to adopt the pre-computing strategy to generate intermediate data, so that the testing server is reasonably used. At the moment a testing request is triggered, a fault localization result can be rapidly computed using the pre-computing results. Thus, the cost of adapted fault localization technique is reduced, without loss of its accuracy.

The proposed idea seems interesting and workable. However, we realize that pre-computing localization results in the coding phase means that locating faults for a future program P. The program P may be not available in the pre-computing process. On the other hand, there will be continuous program versions P' during the coding period and they can be used to provide useful information for the program P. As a result, the pre-computing process can only be carried on P', and at least two issues exist.

1. Since pre-computing is needed, what fault localization technique can be conducted on P' instead of P?

2. How can we make up for the locating accuracy loss caused by using P' instead of P in the pre-computing?

4. RESEARCH EXPLORATION

A. Fault localization in a CI environment

In the continuous integration (CI) [2] environment, program versions are continuously updated, and testing is requested at each commit, which forms a coding-testing-debugging loop. High-accuracy fault localization techniques are mostly time-consuming and cannot be done efficiently. It is deemed a bottleneck. It can be extremely significant to increase the response rate of such techniques.

B. Selecting promising techniques

Let us recall the two issues raised in Section III. We are aiming at localizing faults of the program P. Nevertheless, we need the testing server to pre-compute the information while the code is an intermediate version, say P'. Therefore, in our method, we often get the information from an in-complete and previous program version. However, getting the accurate running results for program version P is essential for the conventional CBFL technique. For instance, Tarantula uses the proportions of passed and failed executions to compute the suspiciousness of each statement in a program. The passed and failed executions must be from an actual program. Obviously, these CBFL techniques are not suitable for pre-computing.

In the next section, we will disclose that there exists at least one fault localization technique, which does not fully rely on the program P to produce fault localization results. Further, we will show how to conduct pre-computing with it and how to adapt it to a CI environment to synthesize a novel testing paradigm.

C. Mutation-enabled fault localization

Mutation testing is proposed to evaluate the quality of a program and measure the failure-revealing capacity of a test suite by manipulating a program using mutation operators.

Some research endeavors have explored mutation analysis to provide accurate fault-localization. Papadakis et al. [7] proposed a mutation-based fault localization technique. Basically, a mutation version, which corresponding test result is very similar to that of the original program, is deemed that the mutant statement is close to the fault position. Formally, given a program P and the associated test suite T. Mutant versions such as P1, P2 and so on are generated using mutation operators. The distinction between P and Pi (i=1, 2, …) can be decided by comparing the program outputs of them, say P(T) and Pi(T). If the output of some mutation version is very similar to that of P, the embedded mutant is more likely to be located on the faulty program statement.

Since most of the computation cost is due to executing mutation versions Pi, we need to pre-compute them. We design our methodologies with the following considerations.

1. During development, modification region of the program is often centralized. Therefore, the complete version P and intermediate version P' are close to each other.

2. According to above, we can generate mutation versions P1, P2, … from the intermediate version P' instead of from P at the coding phase.

We thus enable the mutation-based technique [7] in a pre-computing paradigm, and propose a four-step process, illustrated in Figure 3.

[Step 1]: Generating the mutation versions P1, P2, … from the intermediate version P'.

[Step 2]: Executing the mutation versions P1, P2, … and collect their execution results O1, O2, …., respectively.

[Step 3]: When coding finishes and a testing request is triggered, executing the stable version P to get result O.

[Step 4]: Use the distance between Oi and O to predict the distance between faults with the mutated statement in Pi.

D. Visions and challenges

We propose to use pre-computing to speed up a fault localization technique, which can rapidly respond testing request in a CI environment. Further, we apply the method on existing high-accurate techniques (e.g., [7]), which makes our approach to have adequate accuracy.

Challenges mainly relate to threats to the effectiveness of our approach and the preliminary solution of the problem, in the following aspects.

1. It is recognized in the software engineering research community that mutants may not couple well with real faults. We will integrate high-order mutants to simulate real faults.

2. Previous work indicates that better localization results are achieved by large number of mutation versions [7]. Our approach can be extended by increasing computing resources, since the executing of mutation versions can be in parallel.

3. The scale of program can be very large and seriously increases the complexity. We can focus on the critical region or modifying region to carry out mutations.

5. CONCLUSION

Accurate fault localization is achieved by high-cost computation. In this paper, we propose to make use of pre-computing to distribute time-intensive computation to idle period of coding phase. Our approach consists of two steps, to find fault localization techniques that can be pre-computed, and to adapt it to the pre-computing paradigm in a continuous integration environment. We demonstrate our exploration using an existing mutation-based fault localization technique, and thus found a fault localization technique both effective and efficient. We are now developing our method gradually. Future work includes development of mechanisms integrating other fault localization techniques like value replacement.

ACKNOWLEDGEMENTS

Thanks to Zhenyu Zhang and Jeffrey Hu for their help on sharpening the idea.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J C Van Gemund. On the accuracy of spectrum-based fault localization. In Testing: Academic and Industrial Conference Practice and Research Techniques, pages 89-98, 2007.

[2] B. Jiang, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization?. In Computer Software and Applications Conference, pages 99-106, 2009.

[3] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In Proceedings of the 2008 International Symposium on Software Testing and Analysis, pages 167-178, 2008.

[4] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programing Language Design and Implementation, pages 15-26, 2005.

[5] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statitical debugging: A hypothesis testing-based approach. IEEE Transactions on Software Engineering, 2006.

[6] L. Naish, H. J. Lee, and L. Ramamohanarao. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering Methodology, pages 20-23, 2011.

[7] M. Papadakis and Y. Le-Traon. Using Mutants to Locate "Unknown" Faults. Software Testing, Verification and Validation, pages 691-700, 2012.

[8] X. Xie, T. Chen, F. C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology, 2013

[9] Y. Yu, J. A Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In the 30th International Confference on Software Engineearin, 2008.

[10] X. Zhang, and R. Gupta. Locating faults through automated predicate switching. In proceedings of the 28th International Conference on Software Engineering, pages 272-281, 2006.

[11] Z. Zhang, H. Li, A Technical Report to Continuously Predict the Dynamic Behaviors of Evolving Programs under Development, State Key Laboratory of Computer Science. Technical reports ISCAS-SKLCS-14-11.