This article is part of the IEEE Reliability Society 2009 Annual Technology Report.

# Software Fault Localization

W. Eric Wong, and Vidroha Debroy
Department of Computer Science
University of Texas at Dallas
{ewong,vxd024000}@utdallas.edu

## 1. INTRODUCTION

Regardless of the effort spent on developing a computer program,[1] it may still contain bugs. In fact, the larger, more complex a program, the higher the likelihood of it containing bugs. It is always challenging for programmers to effectively and efficiently remove bugs, while not inadvertently introducing new ones at the same time. Furthermore, to debug, programmers must first be able to identify exactly where the bugs are, which is known as *fault localization*; and then find a way to fix them, which is known as *fault fixing*. In this article, we focus only on fault localization.

Software fault localization is one of the most expensive activities in program debugging. It can be further divided into two major parts. The first part is to use a technique to identify suspicious code that may contain program bugs. The second part is for programmers to actually examine the identified code to decide whether it indeed contains bugs. All the fault localization techniques referenced in the following text focus on the first part such that suspicious code is prioritized based on its likelihood of containing bugs. Code with a higher priority should be examined before code with a lower priority, as the former is more suspicious than the latter, i.e., more likely to contain bugs. As for the second part, we assume *perfect bug detection*, i.e., programmers can always correctly classify faulty code as faulty, and non-faulty code as non-faulty. If such perfect bug detection does not hold, then the amount of code that needs to be examined may increase.

There is a high demand for automatic fault localization techniques that can guide programmers to the locations of faults with minimal human intervention. This demand has led to the proposal and development of various techniques over recent years. While these techniques share similar goals, they can be quite different from one another, and often stem from ideas which themselves originate from several different disciplines. No article, regardless of breadth or depth, can hope to cover all of them. Therefore, while we aim to cover as much ground as possible, we primarily focus on state of the art techniques and discuss some of the key issues and concerns that are relevant to fault localization.

## 2. SOME INTUITIVE TECHNIQUES & TOOLS

One intuitive way to locate bugs when a program execution fails is to analyze its memory dump. Another way is to insert *print* statements around code that is thought to be suspicious such that the values of certain variables, which may provide hints towards the bugs, are printed and examined. The first approach suffers from inefficiency, and intractability due to the tremendous amount of data that would need to be examined. The second approach places the burden on programmers to decide where to insert *print* statements, as well as decide on which variable values to print. Such choices are subjective, and may not be meaningful. As a result, it is also not an ideal technique for identifying the locations of faults.

Debugging tools such as DBX and Microsoft VC++ debugger have been developed, which allow users to set *break* points along a program execution, and examine values of variables, as well as internal states at each break point, if so desired. This approach is similar to inserting *print* statements into the program, except that it does not require the physical insertion of any *print* statements. Break points can be pre-set before the execution, or dynamically set on the fly by the user in an interactive way during program execution. Executions may be a continuous execution from one break point to the next break point, or a

[1] We use “bugs” and “faults” interchangeably. We also use “program” and “software” interchangeably. In addition, “a statement is covered by a test case” and “a statement is executed by a test case” are used interchangeably.

stepwise execution starting from a break point. However, a significant disadvantage of using these tools is that users must develop their own strategies to examine only meaningful information.

## 3. MORE ADVANCED FAULT LOCALIZATION TECHNIQUES

There are several ways to classify fault localization techniques including, but not limited to, the following.

### Static, Dynamic, and Execution Slice-Based Techniques

Program slicing is a commonly used technique for debugging. Reduction of the debugging search domain via slicing is based on the idea that if a test case fails due to an incorrect variable value at a statement, then the bug should be found in the static slice associated with that variable-statement pair [22]. Lyle & Weiser extended the above approach by constructing a program dice to further reduce the search domain for possible locations of a fault [15]. A disadvantage of this technique is that it might generate a dice with certain statements which should not be included. To exclude such extra statements from a dice (as well as a slice), we need to use dynamic slicing instead. Studies such as [2], [19], [31] use the dynamic slicing concept to program debugging.

An alternative is to use execution slicing and dicing to locate program bugs [24], where an execution slice with respect to a given test case contains the set of code executed by this test. There are two principles:

- The more successful tests that execute a piece of code, the less likely for it to contain any fault.
- The more that failed tests with respect to a given fault execute a piece of code, the more likely for it to contain this fault.

The problem of using a static slice is that it finds statements that could possibly have an impact on the variables of interest for *any* inputs instead of statements that indeed affect those variables for a *specific* input. Stated differently, a static slice does not make any use of the input values that reveal the fault. The disadvantage of using dynamic slices is that collecting them may consume excessive time and file space, even though algorithms have been proposed to address these issues. On the other hand, the execution slice for a given test can be constructed easily if we know the coverage of the test.

### Program Spectrum-based Techniques

A program spectrum records the execution information of a program in certain aspects such as how statements and conditional branches are executed with respect to each test. When the execution fails, such information can be used to identify suspicious code that is responsible for the failure.

Tarantula [10] is a popular fault localization technique based on the *executable statement hit* spectrum. It uses the execution trace information in terms of how each test covers the executable statements, and the corresponding execution result (success or failure) to compute the suspiciousness of each statement as $X/(X+Y)$, where X = (number of failed tests that execute the statement)/(total number of failed tests), and Y = (number of successful tests that execute the statement)/(total number of successful tests). One problem with Tarantula is that it does not distinguish the contribution of one failed test case from another, or one successful test case from another. To overcome this problem, Wong et al. [23] propose that, with respect to a piece of code, the contribution of the $n$th failed test in computing its suspiciousness is larger than or equal to that of the $(n+1)$th failed test. The same applies to the contribution provided by successful tests. In addition, the total contribution of the failed tests is larger than that of the successful.

Renieris & Reiss [18] propose a program spectrum-based technique, nearest neighbor, which contrasts a failed test with another successful test that is most similar to the failed one in terms of the "distance" between them. The execution of a test is represented as a sequence of basic blocks that are sorted by their execution counts. If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug that is not contained in the difference set, the technique continues by first constructing a program dependence graph, and then including and checking adjacent un-checked nodes in the graph step by step until the bug is located. The set union, and set intersection techniques are also reported in [18].

Additional examples include *predicate count spectrum* (PRCS) based techniques, which are often referred to as *statistics-based* techniques; *program invariants hit spectrum* (PIHS) based techniques, such as the study by Pytlik, et al. [17]; and *method calls sequence hit spectrum* (MCSHS) based techniques such as [7], and [14].

**Statistics-based Techniques**

Several statistical fault localization techniques have also been proposed, such as Liblit05 [11], and SOBER [12], which rely on the instrumentations and evaluations of predicates in programs to produce a ranking of suspicious predicates, which can be examined to find faults. However, these techniques are constrained by the sampling of predicates. They are also limited to bugs located in predicates, and offer no way to attribute a suspiciousness value to all executable statements.

In light of such limitations, Wong et al. propose a cross tabulation (crosstab) based statistical technique [27] which uses only the coverage information of each executable statement, and the execution result with respect to each test case. It does not restrict itself to faults located only in predicates. More precisely, a crosstab is constructed for each statement with two column-wise categorical variables of "covered," and "not covered;" and two row-wise categorical variables of "successful execution," and "failed execution." The exact suspiciousness of each statement depends on the *degree of association* between its coverage (number of tests that cover it) and the execution results.

**Program State-based Techniques**

A program state consists of variables, and their values at a particular point during the execution. A general approach for using program states in fault localization is to modify the values of some variables to determine which one is the cause of erroneous program execution.

Zeller, et al. propose a program state-based debugging approach, delta debugging [29], to reduce the causes of failures to a small set of variables by contrasting program states between executions of a successful test and a failed test via their memory graphs. Based on delta debugging, Cleve & Zeller [6] propose the cause transition technique to identify the locations and times where the cause of failure changes from one variable to another. A potential problem is that the cost is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow the causes. Another problem is that the identified locations may not be where the bugs reside. Gupta et al. [8] try to overcome these issues by introducing the concept of failure-inducing chops.

Predicate switching [30] proposed by Zhang, et al. is another program state-based fault localization technique where program states are changed to forcefully alter the executed branches in a failed execution. A predicate whose switch can make the program execute successfully is labeled as a critical predicate. Wang & Roychoudhury [20] present a technique that automatically analyzes the execution path of a failed test, and alters the outcome of branches in that path to produce a successful execution. The branch statements whose outcomes have been changed are recorded as bugs.

**Machine Learning-based Techniques**

Machine learning techniques are adaptive, and robust; and have the ability to produce models based on data, with limited human interaction. The problem at hand can be expressed as trying to learn or deduce the location of a fault based on input data such as statement coverage, etc.

Wong et al. [25] propose a fault localization technique based on a back-propagation (BP) neural network, which is one of the most popular neural network models in practice. The statement coverage of each test case, and the corresponding execution result, are used to train a BP neural network. Then, the coverage of a set of *virtual* test cases that each covers only one statement in the program are input to the trained BP network, and the outputs can be regarded as the likelihood of the statements being faulty. However, as BP

neural networks are known to suffer from issues such as paralysis, and local minima, Wong et al. [26] also propose an approach based on radial basis function (RBF) networks, which are less susceptible to these problems, and have a faster learning rate.

Briand et al. [4] use the C4.5 decision tree algorithm to construct a set of rules that might classify test cases into various partitions such that failed test cases in the same partition most likely fail due to the same fault. The statement coverage of both the failed, and successful test cases in each partition is then used to form a ranking based on each partition using a heuristic similar to Tarantula [10]. These individual rankings are then consolidated to form a final statement ranking which can then be examined to locate the faults. There are other studies in this category such as those reported in [3].

**Other Techniques**

There are other fault localization techniques including, but not limited to, data mining-based (e.g., Cellier et al. [5] which discuss a combination of association rules and Formal Concept Analysis (FCA) to assist in fault localization), and model-based (e.g., [28]). Similarity-based coefficients such as Ochiai & Jaccard are also used in [1], [9]. Studies also examine the impact of coincidentally-successful tests on the effectiveness of fault localization techniques [21].

## 4. IMPORTANT ASPECTS

**Effectiveness, efficiency, and robustness of a Fault Location Technique**

One important criterion to evaluate a fault localization technique is to measure its effectiveness in terms of the percentage of code, such as statements that have to be examined by programmers to locate the bug(s). In addition, a fault localization technique should also be efficient in that it should be able to present quality results in a reasonable amount of time without consuming extensive resources.

Also, a test set when executed against the same program, but in two different environments, may result in two different sets of failed test cases. For a fault localizer relying on the coverage and test case execution results as its input, its effectiveness may therefore also vary depending on which environment it is employed in (or rather depending on which environment its input data is collected in). A fault localization technique should be robust to such variations in input (noise), and still perform effectively irrespective of environment.

**Impact of Test Cases**

All empirical studies independent of context are sensitive to the input data. Similarly, the effectiveness of a fault localization technique also depends on the set of failed, and successful test cases employed. Using all the test cases to locate faults may not be the most efficient approach. Therefore, an alternative is to select only a subset of these tests. An important question that remains to be answered is how to select an appropriate set of test cases to maximize the effectiveness of a given fault localization technique.

**Faults introduced by missing code**

One critique against all the fault localization techniques discussed is that they are incapable of locating a fault that is the result of missing code. However, the omission of the code may have triggered some adverse effect elsewhere in the program, such as the traversal of an incorrect branch in a decision statement. This abnormal program execution path may possibly assign certain code with unreasonably high suspicious values that provides a clue to programmers that some omitted code may be leading to control flow anomalies. Still, a more robust approach should be included in any fault localization technique to handle such faults.

**Programs with multiple bugs**

The majority of current research on fault localization focuses on programs with a single bug. A possible extension to programs with multiple bugs can be achieved as follows. When two or more test cases result in a failed program execution, it is not necessary that all the failures are caused by the same fault(s). However, if there is a way to segregate or rather cluster failed executions together such that failed tests in

each cluster are related to the same fault(s), then these failed tests, along with some successful tests, can be used to localize the corresponding causative fault(s). However, there are two significant challenges that need to be overcome. First, there may be more than one possible fault responsible for a failed execution. Second, a precise "due to" relationship between execution failures and causative fault(s) may not even be found without expensive manual investigation. Different clustering approaches have been proposed to address these challenges [9], [13], [16], [32]. However, significant research still needs to be done before such problems can be completely overcome.

## 5. CONCLUSIONS

Choosing an effective debugging strategy usually requires expert knowledge regarding the program in question. In general, an experienced programmer's intuition about the location of the bug should be explored first. If this fails, an appropriate fallback would be a systematic fault localization technique based on solid reasoning, supported by case studies, rather than an unsubstantiated ad hoc approach. However, even with the presence of so many different techniques, fault localization is far from perfect. While these techniques are constantly advancing, software too is becoming increasingly more complex, which means the challenges posed by fault localization are also growing. Thus, there is a significant amount of research still to be done, and a large number of breakthroughs yet to be made.

## REFERENCES

1. R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "A Practical Evaluation of Spectrum-based Fault Localization," *Journal of Systems and Software*, 82(11):1780-1792, November 2009

2. H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software – Practice & Experience*, 23(6):589-616, June 1993

3. L. C. Ascari, L. Y. Araki, A. R. T. Pozo, and S. R. Vergilio, "Exploring Machine Learning Techniques for Fault Localization," *Proc. of the 10th Latin American Test Workshop*, pp. 1-6, Buzios, Brazil, March 2009

4. L. C. Briand, Y. Labiche, and X. Liu, "Using Machine Learning to Support Debugging with Tarantula," *Proc. of the 18th IEEE International Symposium on Software Reliability*, pp. 137-146, Trollhattan, Sweden, November 2007

5. P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux, "Formal Concept Analysis Enhances Fault Localization in Software," *Proc. of the 4th International Conference on Formal Concept Analysis*, pp. 273-288, Montréal, Canada, February 2008

6. H. Cleve and A. Zeller, "Locating Causes of Program Failures," *Proc. of the 27th International Conference on Software Engineering*, pp. 342-351, St. Louis, Missouri, USA, May 2005

7. V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight Defect Localization for Java," *Proc. of the 19th European Conference on Object-Oriented Programming*, pp. 528-550, Glasgow, UK, July 2005

8. N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 263-272, Long Beach, California, USA, November 2005

9. J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in Parallel,"*Proc. of the 2007 International Symposium on Software Testing and Analysis*, pp. 16-26, London, UK, July 2007.

10. J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. of the 20th IEEE/ACM Conference on Automated Software Engineering*, pp. 273-282, Long Beach, California, USA, December, 2005

11. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, Illinois, USA, June 2005.

12. C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical Debugging: A Hypothesis Testing-based Approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, October 2006

13. C. Liu and J. Han, "Failure Proximity: A Fault Localization-based Approach," *Proc. of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 286-295, Portland, Oregon, USA, November 2006

14. C. Liu, X. Yan, H. Yu, J. Han and P. Yu, "Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs," *Proc. of 2005 SIAM International Conference on Data Mining*, pp.286-297, Newport Beach, California, USA, April 2005
15. J. R. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," *Proc. of the 2nd International Conference on Computer and Applications*, pp. 877-883, Beijing, China, June 1987
16. A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated Support for Classifying Software Failure Reports," *Proc. of the 25th International Conference on Software Engineering*, pp. 465-475, Portland, Oregon, USA, May 2003
17. B. Pytlik, M. Renieris, S. Krishnamurthi, and S. P. Reiss, "Automated Fault Localization Using Potential Invariants," *Proc. of the 5th International Workshop on Automated and Algorithmic Debugging*, pp. 273-276, Ghent, Belgium, September 2003
18. M. Renieris and S. P. Reiss, "Fault Localization with Nearest Neighbor Queries," *Proc. of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30-39, Canada, October 2003
19. C. D. Sterling and R. A. Olsson, "Automated Bug Isolation via Program Chipping," *Proc. of the 6th International Symposium on Automated Analysis-Driven Debugging*, pp. 23-32, Monterey, California, USA, September 2005
20. T. Wang and A. Roychoudhury, "Automated Path Generation for Software Fault Localization," *Proc. of 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 347-351, Long Beach, California, USA, November 2005
21. X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, "Taming Coincidental Correctness: Refine Code Coverage with Context Pattern to Improve Fault Localization," *Proc. of the 31st International Conference on Software Engineering*, pp. 45-55, Vancouver, Canada, May 2009.
22. M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, 25(7):446-452, July 1982
23. W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization." *Journal of Systems and Software*, 83(2):188-208, February 2010
24. W. E. Wong and Y. Qi, "Effective Program Debugging based on Execution Slices and Inter-Block Data Dependency," *Journal of Systems and Software* , 79(7):891-903, July 2006
25. W. E. Wong and Y. Qi, "BP Neural Network-based Effective Fault Localization," *International Journal of Software Engineering and Knowledge Engineering*,19(4):573-597, June 2009
26. W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF Neural Network to Locate Program Bugs," *Proc. of the 19th IEEE International Symposium on Software Reliability Engineering*, pp. 27-38, Seattle, Washington, USA, November 2008
27. W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization," *Proc. of the 1st International Conference on Software Testing, Verification and Validation*, pp. 42-51, Lillehammer, Norway, April 2008
28. F. Wotawa, M. Stumptner, and W. Mayer, "Model-based Debugging or How to Diagnose Programs Automatically," *Proc. of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence*, pp. 746-757, Cairns, Australia, June 2002.
29. A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, 28(2):183-200, February 2002
30. X. Zhang, N. Gupta, and R. Gupta, "Locating Faults through Automated Predicate Switching," *Proc. of the 28th International Conference on Software Engineering*, pp. 272-281, Shanghai, China, May 2006
31. X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental Evaluation of Using Dynamic Slices for Fault Location," *Proc. of the 6th International Symposium on Automated Analysis-driven Debugging*, pp. 33-42, Monterey, California, USA, September 2005
32. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical Debugging: Simultaneous Isolation of Multiple Bugs," *Proc. of the 23rd International Conference on Machine Learning*, pp. 26-29, Pittsburgh, Pennsylvania, June 2006.