Empirical Evaluation of Delta Debugging

Arpit Christi
Email: christia@onid.oregonstate.edu

Pratik Krishnamurthy
Email: krishnap@onid.oregonstate.edu

Panini Sai Patapanchala
Email: patapanp@onid.oregonstate.edu

Nitin Jogee Bella Subramanian
Email: subraman@onid.oregonstate.edu

SCHOOL OF EECS
OREGON STATE UNIVERSITY
Corvallis, OR 97331

***Abstract*—Input test cases that cause a program to fail are often large. Most of them are redundant that would not cause the program to fail. Thus, minimizing the number of input test cases could be handy while debugging. The Delta Debugging Algorithm is a general automated technique for minimizing the number of input test cases that cause failure of the code. In this paper, we analyze and present the results of how the Delta Debugging technique helps in the improvement of statistical fault localization and the extent to which it can help developers find the faults quickly and efficiently. We also show through our evaluation that Delta Debugger test case reduction produces the exact same faults with minimized coverage as compared to original failing test cases.**

I. Introduction

Programmers spend a significant amount of their time debugging programs. Software maintenance typically requires more time than any other programming activity. To pin-point a bug, programmers must determine which of the given test cases cause program failure. This search phase of the debugging process is slow and also tedious. Once the program's bug is understood, fixing the bug could be quick. A programmer is often given a large number of test cases that produce a failure. Reducing the number of test cases simplifies the debugging process because there are fewer irrelevant details that the programmers need to focus on, while debugging. Minimizing test cases has traditionally been an activity left for humans to do.

Delta Debugging[3] is an approach for automating test case minimization through its input minimization algorithms. It consists of two algorithms:

Simplification Algorithm:
In this algorithm[1,2], the failure-inducing input is simplified by examining smaller configurations of the input. The algorithm recursively analyzes the smaller failure configurations until it cannot produce a smaller configuration that still produces a failure.

Isolation Algorithm: This algorithm[1,2] attempts to find a passing configuration such that the addition of some element to it cause it to become a failing configuration. The algorithm works in the opposite direction too, finding bigger passing cases that are subsets of a failing case. Isolation produces outputs which are less intuitive as debugging aids to the programmer. A single element difference is not individually responsible for the failure. It merely guarantees that some symptom is exhibited. The programmer then has to go through the potentially large failure-inducing configuration to determine what else might be responsible for the failure. Although isolation may generally be faster than simplification, for large test cases, it may lead to worse running time because of the time spent testing the large configurations.

In this paper, we focus on the results given by the delta debugging algorithm to check for its effects on fault localization, code coverage and efficient debugging.

Our contributions are as follows. We successfully implemented delta debugger tests case to produce reduced input/test cases while still preserving the failure for 3 different projects, 5 different sub projects and 13 different seeded bugs. We measured effectiveness of reduction produced by delta debugger test case by measuring the reduction of test cases as well as reduction in coverage produced by delta debugger output. Our results suggest that 1. Delta debugger can significantly reduce test cases while preserving failure. 2. Such reduction/minimization can help developers to quickly localize the faults. (1 and 2 are hypothetical and not necessarily true as per our current results)

II. History and Related Work

A. History

In his work in delta debgging, [4] Zeller hints at intelligent partition choices as an idea that would significantly reduce the number of test cases run by the algorithm, though it falls short of realizing our motivational insight. Partition boundaries should be chosen with respect to a specific granularity of the input. It follows intuitively that we should start with the coarsest granularity and move towards the nest. Zeller and Hildebrandt[5] briefly mention applying Delta Debugging to specific domains including a short reference to context-free grammars. Rather than examining the input to a program, program slicing can be used to isolate relevant portions of

a program that are necessary to yield some result. Program slicing [8,9] can be performed either statically or dynamically (with respect to one concrete run). These techniques ease debugging by removing irrelevant portions of the failing program. It is worth noting that both Delta Debugging and program slicing can be used cooperatively. By first minimizing the input, we may significantly simplify the program slice and trace. These two techniques[6] are not necessarily competitors; they can be complementary. PSE[10] is a static analysis technique for diagnosing program failures. It can be viewed as a program slicing technique, however it is more precise because of its consideration of error conditions. A motivating example is de-referencing a NULL value. Bug isolation is related to simplifying failure-inducing input. Rather than focusing on minimizing the input, it focuses on finding the cause underlying the failure. Delta Debugging the program state space is used as the mechanism for this technique. The algorithm attempts to locate the state differences between passing and failing runs. This determines the relevant variables and values that infect the program to failure. A similar technique is applied to multi-threaded applications. Instead of focusing on state, the technique examines the thread schedule differences of passing and failing runs. Whalley's [2] work on isolating failure-inducing optimization is also related to our work. His approach automatically isolates errors in the vpo compiler system. The search is performed on the sequence of optimization performed by vpo to find the first improving optimization that causes incorrect output. The approach is fairly domain specific. Liblit et al [7]. use a sampling technique to reduce the run time overhead of collecting successful and failing runs. They also propose to use statistical learning techniques to infer the failures from many sampled runs. Work in error explanation for static analysis relates to our approach. Many tools produce error traces when a program violates its specification. However, understanding the error and locating the cause is usually left to the user.

B. Related Work

Researchers have previously attempted to measure effectiveness of automated debugging techniques. Jones and Harrold compared tarantula with four existing automated debugging technique of contemporary time based on ranking or scoring and reported that tarantula performs better in all the subjects under consideration [11]. Kai Yu et. al. evaluated effectiveness of Hierarchical Delta DebuggingIHDD) in minimizing and isolating changes while preserving the failure. They evaluated minimized changes by qualitative analysis, by actually looking inside the code, the repository changes and making sense out of it [12]. They figured out that, HDD can significantly reduce changes while preserving failure. Recently, Parnin and Orso used a controlled experiment to determine if automated debugging techniques actually help programmers. Their results were mixed and they could not determine if automated debugging techniques actually help developers [13].

III. Motivation

Does automated debugging techniques actually help developers? Most of the time though automated debugging techniques are developed with final goal of helping the developers to debug and localize faults faster, they are evaluated largely in the literature on the basis of some score that defines fault localization degree. Very little research has gone into studying significance of automated debugging techniques with respect to its ability to produce fault localization to help developers to debug and find faults faster. To determine this, we have come up with following research questions:

RQ1: Does delta debugging improve statistical fault localization and how much?

RQ2: Does the delta debugger test case reduction produce exact same faults with significantly reduced coverage?

RQ3: Does delta debugging output help developers to find faults quicker?

To answer research questions 1 and 2, we have applied delta debugger on 5 different subjects and measured and analyzed the output produced by delta debugger. To answer research question 3, we conducted a controlled experiment using 60 graduate students as our subjects. (Assuming that we conducted a controlled experiment with normal debugging vs delta debugging as our treatments, We have submitted a write up of such a study conducted on 5 participants as part of CS569 course.)

IV. Experimental Setup

We choose 3 random projects for our evaluation. For each project, We started with analyzing the existing test cases. We introduced defects to fail these test cases. Then, we use delta debugger to minimize these test cases and then checked the results to answer the above research questions.

A. Objects of Analysis

For the objects of analysis, we used 3 projects from apache commons 1. commons validator 2. CSV Parser and 3. commons CLI(Command Line Interface). Table 1 describes the size of each of these projects in terms of non empty, non commented LOC. We used two independent sub project in case of validator (Email validator and URL validator). Similarly we used two independent sub projects in case of CLI (POSIX CLI and GNU CLI). So, In total we have 5 different subjects under evaluation, 1 for CSV Parser, 2 for commons validator and 2 for commons CLI. Apache commons is widely used suite because it contains robust code base, well written test suite and well documented bug repository apart from being open source. Many software testing and fault localization research have used apache commons projects as their subjects[cite or remove]. Each of our subjects only executes very little part of the entire project. For example, running all the test cases for POSIX CLI only covers following java files : CommandLine, CommandLineParser, GnuParser, Option, Options, Parser, PosixPasrer, Util and ParserException. While doing any analysis on a subject, we instrumented only files that are

covered by test cases of the subject under consideration.. Table 1 contains effective LOC for each of the subjects that is total LOC of files instrumented for the analysis of the particular subject.

B. Faults

We used seeded bugs to produce fault in the program. Table 2 describes the seeded bugs, their description and number of test cases failed because of the bugs/faults. For our analysis, we introduced 13 faults. For each of the faults in the table, we also mentioned number of test cases failed because of the fault. Though, It is more desirable to reproduce faults from issue repository of apache commons project, we seeded meaningful mutants as our faults. By meaningful mutant we mean that though the bug is eventually a seeded mutant of existing program, it can be meaningfully described as a bug. For example, in case of EmailValidator, Bug 1, though a mutant, can be described as If an email address ends with a . character, it should be an invalid email address while the program lets it pass as valid email address without giving error or warning. Though apache commons believed to have robust test suite, to our surprise we found out that many of the meaningful mutants that we created were not killed by test suite.

C. Construction of Test Functions

We used Zellars delta debugger (java version) without any modification [3]. For input minimization or test minimization, one needs to implement the ITester interface and test method of delta debugger. We implemented test method for all of our subjects.For further discussion, delta debugger test or delta debugger test case means implementation of test method of ITester interface of Zellars delta debugger tool. Implementing delta debugger test can be time consuming process but one has to trade this for fault localization produced by input or test case minimization. How does a delta debugger test knows about occurrence of a fault and hence determine to proceed with minimization? Each delta debugger test is supplied with an oracle of its own. In some cases, the oracle was designed by researchers based on output of the correct version of the program. In some cases, we used a replica of correct program as our oracle. Delta debugger test then used faulty version of the program and oracle to determine existence of a fault. We set up time out threshold of 10 minutes for delta debugger test case. If delta debugger test couldn't report minimization within timeout period, the results are reported as Not Applicable. Our implementation of delta debugger tests for each of the subjects is available on GitHub. [provide a link in footnote]

V. Results and Conclusion

A. Results

We measured effectiveness of delta debugging with respect to fault localization by measuring reduction produced in code coverage and number of failing test cases produced by delta debugger test case for each of the subject. After introducing each fault, we run the original test suite and find out the tests being failed. We also measured the coverage for instrumented code while running the failing test in original test suite. We only used failing tests to measure coverage, because developers when fixing the fault concentrate on the failing test cases. Then we run delta debugger test and produced the minimization of input/test while keeping the failure. Then we designed another test suite based on minimized input or test cases produced by minimization. Though it is possible to produce the minimized test suite automatically using output of delta debugger, for this study, we wrote test suite by ourselves based on the minimization produced by delta debugger. To automate this remains as future work. We executed the delta debugger test suite and measured the coverage produced by delta debugger. Table 3 represents our results, for both original test suite(failing tests only) and the reduced test suite. Table 3 represents coverage for statement(SC), branch(BC), Loop(LC) and Term (TC) for both the test suite. It also mentions number of failing test cases in both the case. Table 4 represents percentage reduction produced for SC, BC, LC and TC. It also provides decrease in number of test cases produced by delta debugger. The average reduction (in percentage) produced by delta debugger for SC is 33% and branch coverage is 36% while the maximum reduction for SC is 62% and BC is 65%. Average reduction produced in terms of number of failing test cases (in exact number) is 5.53 and maximum reduction produced is 22. It is important to notice that delta debugger test was able to produce the minimization within the allowable threshold for all the faults except fault number 6. Most of the researchers have found delta debugging to be slow while using it for reduction of large test cases or large inputs [cite]. As our projects had very limited size and our test cases and inputs were quite small, we ran into this problem once only with URL Validator. It is important to note that URLValidator checks for 75000+ different possibilities of urls in single test case. URlValidator test suite is logically the biggest test suite we have.

B. Indirect Matrix Usage

We measured effectiveness of delta debugging by measuring reduction in test cases and reduction in code coverage produced by delta debugger. Though this is a very indirect metrics and does not directly associated with fault localization, one can use this as a starting point based on the qualitative fact that if a developer has less number of test cases or reduced test case that covers less parts of program, it will help him to remain within correct path of the fault and hence assist fault localization. :) . We should have used better matrix that is directly associated with fault localization and widely accepted in literature. A common practice in fault localization is assign some sort of score to fault localization and measure effectiveness of reduction based on the score[cite]. Fault localization score obtained by automated debugging techniques should be significantly higher than otherwise. Though Kai Yu et al. used a pure qualitative analysis of walking through the code and manually evaluating fault localization produced by Hierarchical Delta Debugger to minimize subversion changes

TABLE I
SUBJECTS OF ANALYSIS

Subject	Project	Total Non-empty Non-commented Lines of Code	Instrumented Non-empty Non-commented Lines of Code
URL Validator	Validator	2089	335
email Validator	Validator	2089	201
CSV-Parser	CSV	1233	1139
GNU-cli	commons-cli	3241	1109
POSIX-cli	commons-cli	3241	1188

TABLE II
FAULTS, DESCRIPTION OF FAULTS AND NUMBER OF FAILING TEST CASES

Fault	Subject	Description of Bug	Num of FTC
1	email-validator	If email ends with ., it is valid email	2
2	email-validator	Error with emails containing ip addresses	1
3	email-validator	Combination of both	3
4	url-validator	Error introduced for url with query string	1
5	url-validator	Error in url with inetaddress	3
6	url-validator	Error in url with localhost	1
7	csv	Error is size if csv record is empty	1
8	posix-cli	Error in list of arguments	17
9	posix-cli	Error if args contain certain character	10
10	posix-cli	If argument is single dash (-) it is not added to list of arguments	1
11	posix-cli	tokens are not added correctly	23
12	GNU-cli	Error in list of arguments	14
13	GNU-cli	Error if args contain certain character	8

TABLE III
TEST SUITE RUN,COVERAGE INFORMATION AND FAILING TEST CASES

Faults	Subject	Normal Run					Delta Run				
		SC	BC	LC	T	FTC	SC	BC	LC	T	FTC
1	email-Validator	74.6	42	N/A	N/A	2	31.3	14.3	N/A	N/A	1
2	email-Validator	37.3	25	N/A	N/A	1	37.3	25	N/A	N/A	1
3	email-Validator	79.1	50	N/A	N/A	3	37.3	25	N/A	N/A	1
4	URL-Validator	67.4	42	14.8	31.2	1	25.6	32.1	14.9	29	1
5	URL-Validator	46.5	66	37	63	3	20.9	25.5	11.1	23.9	1
6	URL-Validator	31.4	44.3	18.5	41.3	1	∞	∞	∞	∞	∞
7	CSV	48.1	31.7	12.5	31.5	1	37	22	9	23	1
8	POSIX-cli	28.9	26.3	21.5	23.8	17	17.6	13.5	5.4	9.9	1
9	POSIX-cli	22.4	21	14	18.6	10	17.6	13.5	5.4	9.9	1
10	POSIX-cli	35.8	36.1	33.3	34.3	1	21	17.1	6.2	13.1	1
11	POSIX-cli	30.2	30	23.5	27.6	23	21	17.1	6.2	13.1	1
12	GNU-cli	21	17.1	6.2	13.1	14	21	17.1	6.2	13.1	1
13	GNU-cli	44.8	45.5	46.9	43.7	8	21	17.1	6.2	13.1	1

TABLE IV
REDUCTION IN COVERAGE AND NUMBER OF FAILING TEST CASES

Fault	Subject	SC %	BC %	LC %	TC %	Difference in Number of Failing Cases
1	email-validator	58.04	65.95	NA	NA	1
2	email-validator	0	0	NA	NA	0
3	email-validator	52.84	50	NA	NA	2
4	url-validator	62.02	23.57	0	7.05	0
5	url-validator	55.05	61.36	70	62.06	2
6	url-validator	0	0	0	0	0
7	csv	23.08	30.6	28	26.98	0
8	posix-cli	41.34	52.63	81.38	61.8	16
9	posix-cli	30.46	43	73.62	52.54	9
10	posix-cli	0	0	0	0	0
11	posix-cli	53.13	62.42	86.78	70.02	22
12	GNU-cli	39.1	48.67	74.88	58.4	13
13	GNU-cli	21.43	35.71	61.43	46.77	7
Average		62.02	65.95	86.78	70.02	22
Max		33.58	36.45	47.61	38.56	5.54

while preserving the failure [12].

C. Conclusion

We answered each of the research questions based on our experiments and results as follows.

RQ1: Does delta debugging improve statistical fault localization and how much? Based on our current results, this question largely remained unanswered. We need to collect a large number of different matrix before arguing about the statistical fault localization produced by delta debugger. Currently, we just have minimized test cases/inputs and results describing reduction in number of test cases and significant reduction in coverage, though this information is largely meaningless at this point of time. It's just a starting point.

RQ2: Does the delta debugger test case reduction produce exact same faults with significantly reduced coverage? Our results suggest that delta debugger can significantly reduce coverage or test cases while producing exact same faults. Our implementation of delta debugger test case for subjects produce on average a decrease of 5.53 test cases per fault with maximum reduction in failing test cases being 22 in case of POSIX CLI fault 11. (This number needs to go significantly up for final paper). Also, average reduction in statement coverage and branch coverage are 33% and 34% respectively while corresponding maximum values are 62% and 65%.

RQ3: Does delta debugging output help developers to find faults quicker?
Based on the results of controlled experiment that we conducted, we can say that delta debugger helps developer to find out and fix bugs faster then otherwise.

VI. Future Work

1. We need to collect a large number of other matrix on current data set and figure out that if statistically significant fault localization is produced or not.
2. The size of subjects under consideration and produced reduction don't sound very compelling. The reason behind it that the original test suite itself doesn't cover large amount of code (surprisingly). If necessary use randoop or other tool so that you can start with 100% or close to 100% initial code coverage so that your results will amplify and look good.
3. Try to get the python and mozilla test case reduction thing working in next month or so, so that you can have strong additional data available.
4. Try to make the 569 delta debugging evaluation using controlled experiment a reality by summer.
5. Try to use atleast 2 or 3 faults from bug repository of apache commons instead of seeded bug immediately

VII. References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. Commun. ACM, 21(6):466471, 1978.

[2] Ghassan Misherghi, Zhendong Su. HDD: Hierarchical Delta Debugging Department of Computer Science University of California, Davis

[3] Philipp Bouillon. Diploma Thesis : A Framework for Delta Debugging in Eclipse.University of Saarlandes, Germany

[4] A. Zeller. Isolating cause-effect chains from computer programs. In Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering.

[5] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Trans. Software Engineering, 28(2), Febuary 2002.

[6] C. Sterling and R. A. Olsson. Automated bug isolation via program chipping. In Sixth International Symposium on Automated Debugging and Analysis-Deiven Debugging(AADEBUG'05), 2005.

[7] F. Tip. A survey of program slicing techniques. Journal of programming languages, 3:121.189, 1995.

[8] M. Weiser. Program slicing. In Proceedings of the 5th International Conference on Software Engineering, pages 439.449, 1981.

[9] M. Weiser. Programmers use slicing when debugging. Communications of the ACM, 25(7):446.452, 1982.

[10] R. Manevich, M. Sridharan, and S. Adams. PSE:explaining program failures via postmortem static analysis. In SIGSOFT'04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pages 63.72,2004.

[11] Jones, James A., and Mary Jean Harrold. "Empirical evaluation of the tarantula automatic fault-localization technique." Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005.

[12] Yu, Kai, et al. "Towards automated debugging in software evolution: Evaluating delta debugging on real regression bugs from the developers perspectives." Journal of Systems and Software 85.10 (2012): 2305-2317.

[13] Parnin, Chris, and Alessandro Orso. "Are automated debugging techniques actually helping programmers?." Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, 2011.