

A Linear Programming Approach for Automated Localization of Multiple Faults

Brian C. Dean, William B. Pressly, Brian A. Malloy and Adam A. Whitley
School of Computing
Clemson University
{bcdean, wpressl, malloy, awhite}@cs.clemson.edu

Abstract

In this paper, we address the problem of localizing faults by analyzing execution traces of successful and unsuccessful invocations of the application when run against a suite of tests. We present a new algorithm, based on a linear programming model, which is designed to be particularly effective for the case where multiple faults are present in the application under investigation. Through an extensive empirical study, we show that in the case of both single and multiple faults, our approach outperforms a host of prominent fault localization methods from the literature.

1. Introduction

Identifying the faulty lines of code in a program is one of the most important and time consuming aspects of the software development cycle [1, 16, 17]. In order to reduce the cognitive burden of the developer in localizing faults, many researchers have investigated techniques for automating the fault localization process [3, 4, 10, 14, 15, 17, 19, 18]. Ideally, a fault localization algorithm should *separate* distinct faults from each-other (without being told the number of faults to expect), *localize* each fault by listing a small set of lines that comprise the fault, and *explain* the nature of each fault in terms of the nature of the input data that causes the fault to manifest itself. In its full generality, this problem is extremely difficult, and it is perhaps overly optimistic to believe that any automated procedure can offer a complete solution to this problem.

One of the most prominent models for automatic fault localization appearing in the literature is a *spectrum-based* model: a program is run on a number of test inputs, with each execution characterized abstractly in terms of its “spectrum” — a feature vector describing the characteristics of the execution. Unfortunately, spectrum-based fault localization is quite challenging from a computational stand-

point, being closely-related to the NP-hard MINIMUM TEST COLLECTION problem [8]. In a recent survey of state-of-the-art methods for fault localization using binary code coverage spectra, Abreu et al. [2] observe that for half the programs they tested, even the best techniques currently available still require examination of at least 10% of a program in order to locate faults successfully; the same observation is also made in [10]. As would be expected, localization in the presence of multiple faults is even more difficult and costly than localization of a single fault [13].

In this paper, we describe a novel approach to automated fault localization based on a linear programming framework. Our method is specifically designed to be adept at accommodating multiple faults, although it often outperforms all of the prominent spectrum-based approaches recently surveyed by Abreu et al. [2] in both single-fault and multiple-fault experiments. As opposed to the majority of the techniques discussed in [2], which rank all the lines of code in a program according to their “suspiciousness”, our method identifies a very small maximally-suspicious *set* of lines of code that collectively explains all of the failing test cases in terms of its aggregate coverage.

2. Background and Motivation

Consider the simple C program in Figure 1, adapted from [12], that attempts to compute the median of three input numbers x , y , and z . This program contains two faults: line 7 should read “ $m = x$ ” and line 11 should read “else if ($x < y$)”. To the right of the program we see a binary *code coverage matrix*, where the rows correspond to lines of code, the columns correspond to test cases in a test suite, and the (i, j) entry is a one or zero depending on whether line i is executed in test case j . The matrix is partitioned according to failing and passing test cases; otherwise, the ordering of its columns is arbitrary. Each row of the code coverage matrix gives us a feature vector, or *spectrum*, characterizing the execution profile of a single line of code.

	Failing cases						Passing cases					
	1,7,4	3,6,4	7,3,8	2,1,4	4,2,6		2,3,8	6,4,3	9,5,6	7,3,2	2,9,1	
1: read (x, y, z);	1	1	1	1	1		1	1	1	1	1	
2: m = z;	1	1	1	1	1		1	1	1	1	1	
3: if (y<z) {	1	1	1	1	1		1	1	1	1	1	
4: if (x<y)	0	0	1	1	1		1	0	1	0	0	
5: m = y;	0	0	0	0	0		1	0	0	0	0	
6: else if (x<z)	0	0	1	1	1		0	0	1	0	0	
7: m = y;	0	0	1	1	1		0	0	0	0	0	
8: } else {	1	1	0	0	0		0	1	0	1	1	
9: if (x>y)	1	1	0	0	0		0	1	0	1	0	
10: m = y;	0	0	0	0	0		0	1	0	1	0	
11: else	1	1	0	0	0		0	0	0	0	1	
12: m = x;	1	1	0	0	0		0	0	0	0	1	
13: }	0	0	0	0	0		0	0	0	0	0	
14: print ("Median = ", m);	1	1	1	1	1		1	1	1	1	1	

Figure 1. Code coverage matrix of a simple program, with two buggy lines highlighted.

One view of our fault localization problem is, given the code coverage matrix as input, locate the lines of code that are most “suspicious” in terms of the likelihood that they contain faults. A number of authors have proposed various measurements of the suspiciousness s_i of a line i of code based on its spectrum; each of the following evaluates to a number in the range $[0, 1]$, with 1 being the most suspicious:

$$\begin{aligned}
\text{AMPLE [6]:} & \quad |f_i - p_i| \\
\text{Tarantula [11, 12]:} & \quad f_i / (f_i + p_i) \\
\text{Jaccard [5]:} & \quad f_{i,1} / (p_{i,1} + f_{i,0} + f_{i,1}) \\
\text{Correlation [2]:} & \quad f_{i,1} / \sqrt{(p_{i,1} + f_{i,1})(f_{i,1} + f_{i,0})},
\end{aligned}$$

where $f_{i,1}$ and $p_{i,1}$ ($f_{i,0}$ and $p_{i,0}$) denote the number of failing and passing test cases that execute (don’t execute) line i , and where f_i and p_i respectively denote the fraction of failing and passing cases executing line i . Fault localization according to these metrics proceeds by ranking all lines in a program in decreasing order of suspiciousness, then by examining lines one by one until all bugs are found. See [2] for a detailed comparison of the performance of these metrics.

Note that two lines having the same spectrum are effectively indistinguishable for our purposes. Similarly, two test cases with identical execution profiles (binary code coverage vectors in their corresponding columns) are indistinguishable. As a consequence, any fault localization algorithm based solely on binary spectra can merge lines of code with identical spectra into *equivalence classes*, and likewise for the test cases. For example, Figure 1 indicates the equivalence class to which each line of code belongs. Lines

within equivalence classes cannot be differentiated, so if a fault is 1 of 100 lines in a large equivalence class, we cannot hope to find the fault without examining all 100 lines in the worst case (or 50 lines in the average case). This is an inherent limitation of any fault localization algorithm based solely on binary spectra. We henceforth generally assume that we are dealing with equivalence classes of lines rather than single lines by themselves, and likewise for the test cases; note that it is a simple matter to compress our original code coverage matrix so its rows and columns represent equivalence classes. We use the term *lineEC* to denote an equivalence class of lines. Baudry et al. [4] refer to these as *dynamic basic blocks*.

3. Our New Method

Our algorithm strives to find a small set of suspicious lineECs that explains, or “covers” all of the failing test cases. To represent such a solution, we introduce a decision variable $x_i \in [0, 1]$ for each lineEC i , whose value indicates the extent to which lineEC i belongs to our solution. We denote by L the set of all lineECs, and by x the vector of all x_i ’s for $i \in L$. We can represent an exact, “crisp” set of lineECs S by taking x to be the binary incidence vector of S , so that $x_i = 1$ for all $i \in S$. We can also represent a “fuzzy” set S by including some lineECs to a fractional extent.

We now show how to find a set of lineECs x having large “suspiciousness”. If L_j denotes the set of lineECs executed by some test case j , we define the *coverage* of case j by a solution x as $y_j = \min\left(1, \sum_{i \in L_j} x_i\right)$. In the common case where x is an integral solution, we will have $y_j = 1$ if some lineEC i with $x_i = 1$ is executed by test case j . Let us now define the following quantities for any solution x :

- The aggregate failure ratio, $f(x) = \sum_i f_i x_i$
- The aggregate passing ratio, $p(x) = \sum_i p_i x_i$
- The total coverage, $c(x) = \frac{1}{|F|} \sum_{j \in F} y_j$, where F is the set of all failing test cases.
- The generalized suspiciousness of x ,

$$s(x) = \frac{A + f(x)}{1 + f(x) + p(x)},$$

where A is a constant to be determined shortly.

We can now state our problem as a mathematical program, where we wish to maximize $s(x)$ subject to the constraint $c(x) \geq 1$. Even though its objective involves a ratio, we can solve this mathematical program by solving a series of linear programs as follows. Consider the simpler problem of determining whether or not $s(x^*) \geq \lambda$, for some specified λ , where x^* is an optimal solution. That is,

we wish to determine if there exists some solution x with $c(x) \geq 1$ for which $(A + f(x))/(1 + f(x) + p(x)) \geq \lambda$. By rearranging terms, this is equivalent to asking if we can find some solution x with $c(x) \geq 1$ for which $A + f(x) - \lambda p(x) - \lambda f(x) - \lambda \geq 0$, and this can be answered by solving a linear program:

$$\begin{aligned} \text{Maximize} \quad & A - \lambda + \sum_{i \in L} (f_i - \lambda p_i - \lambda f_i) x_i \\ \text{Subject to} \quad & \sum_{j \in F} \frac{1}{|F|} y_j \geq 1 \\ & \sum_{i \in L_j} x_i \geq y_j \\ & x_i \in [0, 1] \quad \forall \text{ lineECs } i \in L. \\ & y_j \in [0, 1] \quad \forall \text{ test cases } j \in F. \end{aligned}$$

If the optimal solution to the program above is non-negative, then our guess λ was lower than the optimal objective z for the original problem; otherwise, it was higher. We can therefore solve the original problem by binary search on λ . Observe that the constraints in the linear program above express the coverage constraint $c(x) \geq 1$ as a system of linear inequalities. The value of A is set to be just large enough so that $s(x^*) \geq 1$; otherwise, all lineECs i for which $f_i/(f_i + p_i) > s(x^*)$ would be automatically included in our solution, even if they were not necessary for coverage; further elaboration on this subtlety will appear in the full version of this paper.

After producing a set of lineECs S as output, our algorithm performs one final postprocessing step: it augments this set with all single lines of code containing conditional expressions that are immediate parents of lines in S in the control-flow graph of our program. For example, if S contained lines in the body of an “if” statement, we would add the single line containing the “if” conditional to our solution. The intuition behind this step is that if the parent conditional contains a bug, then this will manifest itself not in the execution profile of the conditional, but rather in the execution profiles of its children in the control-flow graph. Our experimental studies have shown that this postprocessing step contributes significantly to the quality of our solutions, and moreover it does not inflate the size of our solutions by too much, since most lineECs have only zero or one parent conditionals. It is worth noting that it is not immediately obvious how one would apply a similar postprocessing step in an effective manner to a ranking-based fault localization approach, say based on Tarantula or the correlation metric; this might be a good direction for future research.

4. Case Study

This section describes our study to investigate the efficiency and utility of our linear programming algorithm.

The test suite for our study is derived from eight C programs. Seven of these come from the Siemens suite, a

set of small programs developed at the Siemens Research Corporation [9]. These programs, each with documented “real world” faults, have been used extensively in the literature for the purpose of testing spectrum-based techniques for fault localization (see, e.g., [3, 9, 10, 15]). The eighth program, **Space**, is an interpreter for the Array Definition Language (ADL), a language that controls and coordinates the function of antenna arrays in satellites, written by the European Space Agency [7].

The source code package for the Siemens suite and the Space program consists of multiple versions of each of the programs, each containing a single fault. By analyzing these, we first created a “clean” version of each program containing no faults, then performed a `diff` of the clean version and each faulty version to isolate the faulty lines of code. By enclosing faulty lines with preprocessor directives, we created a single source package for each of our eight test programs, which could be compiled with any desired subset of faults enabled.

All executions were run on a single-processor 64-bit 2.2 Ghz Opteron processor with 4 Gb RAM. Linear programs were solved using CPLEX 10.0. For the 6218-line **Space** program, our algorithm requires roughly 2 seconds on average to run. For the smaller programs in the Siemens suite, running time is well under one second. This is significantly less than the amount of time required to run the program against its test suite in the first place.

One of the principal strengths of our approach is that it always appears to produce a small set of lines as output, making it easy for a human practitioner to check the resulting lines for faults. As shown in Figure 2, our average estimated solution size is typically no more than 10 lines for the programs in the Siemens suite, and at most 46 lines for the larger 6218-line Space program. As might be expected, we see that the average size of a solution tends to increase slightly with program size and also when we seed our program with more faults. However, even with three faults, average solution size tends to be only a tiny percentage of total program size.

We now discuss the comparison of our algorithm to the prominent ranking approaches reviewed in [2], based on a suspiciousness metric that is either Tarantula, AMPLE, Jacard, or correlation. Note that this is somewhat of an “apples to oranges” comparison, since we cannot make a true comparison between our algorithm (which outputs a set of lineECs) and a ranking-based approach. However, a reasonably fair method of comparison is to take the top X most

⁰Recall that our solution set is expanded during postprocessing by including parent conditionals, but this expansion is not likely to increase the solution size appreciably, since each lineEC typically has only zero or one parent conditional. To obtain an estimate of this expansion (since our prototype testing infrastructure does not enable recovery of parent pointers from lineECs in a convenient fashion), we have charged ourselves one extra line for each lineEC in our solution.

suspicious lines from the ranking-based approach, where X denotes the number of lines returned by our algorithm. That is, both methods are compared based on the quality of the lines they return, each set having the same size. For each algorithm, we compute the expected number of faults located in a single trial by computing the average number of faults located over a long series of trials (by trial, we mean taking one of our test programs, seeding it with faults, and running both methods to see how many faults are located by each). The number of trials run for each program depends on the number of faulty versions available for that program, and the number of faults seeded; for tests involving multiple faults, we performed up to 100 trials on randomly chosen combinations of compatible faults.

When running a ranking-based approach, lines with equal suspiciousness are assumed to be randomly ordered, allowing for us to find only a fraction of a fault in expectation. For example, if we are comparing against a set of $X = 4$ lines, and the most suspicious lineEC contains 1 faulty line and 5 other lines, then we would say that the expected number of faults found within the top X lines is $2/3$.

The table in Figure 2 summarizes the results of our comparison. Some of the entries are not as meaningful from a statistical point of view, since they are computed from a small number of runs (e.g., one should not try to draw any meaningful conclusions by just examining the results for the 5 runs of `print_tokens`, when seeded with one fault). We note that while our algorithm occasionally falters, its performance is consistently better than all other ranking-based approaches when we aggregate all the runs for the Siemens suite programs — and this is true when we seed one faults, two faults, and three faults. For the larger `Space` program, our algorithm improves its standing with larger numbers of faults present.

References

- [1] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund. Refining spectrum-based fault localization rankings. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 409–414, New York, NY, USA, 2009. ACM.
- [2] R. Abreu, P. Zoetewij, and A. J. C. v. Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC '06: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.
- [3] R. Abreu, P. Zoetewij, and A. J. van Gemund. A dynamic modeling approach to software multiple-fault localization. In A. Grastien and M. Stumptner, editors, *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08)*, pages 7–14, Blue Mountains, NSW, Australia, September 2008.
- [4] B. Baudry, F. Fleurey, and Y. L. Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 82–92, 2006.
- [5] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, 2002.
- [6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization in java. In *ECOOP 2005: 19th European Conference*, pages 528–550, 2005.
- [7] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
- [8] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of books in the mathematical sciences. W. H. Freeman, San Francisco, 1979.
- [9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [10] J. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.
- [11] J. Jones, M. J. Harrold, and J. Stasko. Visualization for fault localization. In *Proceedings of the Workshop on Software Visualization*, 2001.
- [12] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 191–200, 2002.
- [13] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26, New York, NY, USA, 2007. ACM.
- [14] H. Pan, R. A. DeMillo, and E. H. Spafford. Failure and fault analysis for software debugging. In *Proceedings of COMPSAC 97*, pages 515–521, 1997.
- [15] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 30–39, 2003.
- [16] I. Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.
- [17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.
- [18] A. X. Zhang, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous isolation of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning (ICML)*, pages 1105–1112, 2006.
- [19] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems (NIPS) 16*, 2003.

	# Runs	AvgSize	AMPLE	Tarantula	Jaccard	Correlation	LinearProg
schedule	6	3.50	0.208	0.167	0.167	0.380	0.500
schedule2	8	3.38	0.250	0.250	0.250	0.250	0.250
tot_info	19	6.16	0.246	0.205	0.216	0.246	0.474
tcas	36	3.08	0.171	0.168	0.171	0.171	0.111
print_tokens	5	5.40	0.600	0.400	0.400	0.800	0.600
print_tokens2	10	3.70	0.417	0.342	0.342	0.442	0.500
replace	29	3.24	0.241	0.224	0.224	0.241	0.379
aggregate-SS	113	-	0.250	0.220	0.223	0.270	0.327
space	33	19.24	0.496	0.387	0.518	0.536	0.485

1 fault seeded

	# Runs	AvgSize	AMPLE	Tarantula	Jaccard	Correlation	LinearProg
schedule	14	3.93	0.253	0.393	0.164	0.274	0.500
schedule2	36	8.44	0.222	0.307	0.307	0.224	0.167
tot_info	100	13.73	0.531	0.484	0.467	0.485	0.750
tcas	99	8.92	0.541	0.594	0.601	0.567	0.323
print_tokens	9	9.89	1.139	0.694	0.917	1.361	1.111
print_tokens2	43	5.16	0.479	0.556	0.641	0.670	0.651
replace	100	5.96	0.317	0.435	0.375	0.402	0.630
aggregate-SS	401	-	0.451	0.492	0.481	0.493	0.551
space	100	35.52	0.477	0.644	0.718	0.721	0.790

2 faults seeded

	# Runs	AvgSize	AMPLE	Tarantula	Jaccard	Correlation	LinearProg
schedule	19	4.42	0.364	0.353	0.237	0.435	0.632
schedule2	84	6.08	0.583	0.667	0.667	0.643	0.571
tot_info	100	11.09	0.640	0.531	0.621	0.708	0.980
tcas	100	4.01	0.396	0.437	0.446	0.385	0.350
print_tokens	7	13.29	1.571	1.518	1.536	1.572	1.286
print_tokens2	100	6.14	0.634	0.780	0.807	0.898	0.770
replace	100	8.23	0.465	0.708	0.567	0.622	0.970
aggregate-SS	510	-	0.550	0.625	0.618	0.656	0.737
space	100	46.00	0.580	0.776	0.711	0.744	1.060

3 faults seeded

Figure 2. Comparison between our approach and ranking-based approaches. The # runs column tells how many trails we ran of each program, and the AvgSize column reveals the estimated average number of lines of code returned by our program over all these trials. The remaining columns indicate the expected number of faults located by the AMPLE, Tarantula, Jaccard, and correlation methods, as well as our own method, shown in the final column.