# An Approach for Classifying Program Failures

Burcu Ozcelik, Kubra Kalkan, and Cemal Yilmaz
*Faculty of Engineering and Natural Sciences*
*Sabanci University*
*Istanbul, Turkey*
{*burcuoz, kubrakalkan, cyilmaz*}@*sabanciuniv.edu*

***Abstract*—In this work, we leverage hardware performance counters-collected data to automatically group program failures that stem from closely related causes into clusters, which can in turn help developers prioritize failures as well as diagnose their causes. Hardware counters have been used for performance analysis of software systems in the past. By contrast, in this paper they are used as abstraction mechanisms for program executions. The results of our feasibility studies conducted on two widely-used applications suggest that hardware counters-collected data can be used to reliably classify failures.**

***Keywords*-failure classification; debugging aids; hardware performance counters.**

## I. INTRODUCTION

Many data-driven program analysis approaches have been proposed in the literature. These approaches instrument the source code and/or binaries of programs, collect execution data from program runs every time the instrumentation code is exercised, and analyze the collected data to help shape future software development efforts. Some example applications of this general approach include predicting failures, detecting failures, and identifying likely causes of failures [5], [9], [11], [13], [14], [16].

Another application, which is also the focus of this paper, is concerned with automatically classifying failed program executions. The classifications are often obtained by collecting a specific type of execution data from failing runs, which from now on is referred to as a program spectrum, and then feeding this data to an unsupervised learning algorithm, such as a cluster analysis algorithm. The result is a set of clusters grouping similar failed executions together. Many empirical studies suggest that individual clusters obtained from these types of analyses tend to contain failures that stem from closely related causes [6], [7], [8], [11]. Therefore, the resulting clusters can help developers prioritize failures as well as diagnose their causes.

A fundamental assumption behind these approaches is that there are repeatable and identifiable patterns in failed program executions and similarities to these patterns are highly correlated with the causes of failures. Previous empirical studies reported in the literature support this assumption [5], [9], [11]. Consequently, one factor that affects the accuracy of classifications is the type of program spectrum collected to identify patterns in executions. In general, it is possible to collect quite detailed information at runtime to identify the patterns. However, the overhead cost both in terms of the runtime overhead required to collect the spectra and the space overhead required to store them often makes this approach an impractical one. Therefore, it is still an open question what to collect and how to analyze the collected data to identify meaningful patterns in executions [16].

In this work a novel approach is proposed, in which most of the data collection work is carried out by fast hardware performance counters. Hardware performance counters are CPU-resident counters that record various types of events occurring on a CPU. They have been frequently used to perform performance analysis of software systems in the past (e.g., hot-spot analysis). By contrast, we leverage them for a functionality-related purpose to classify failed program executions.

The contributions of this paper can be summarized as follows:

- A hardware performance counters-based program spectrum as an abstraction mechanism for program executions to classify failures is proposed.
- The proposed approach is empirically evaluated by conducting experiments using two widely-used, real life applications as our subject applications.
- The accuracy of our hardware counters-based spectra in classifying failed executions to those of three different types of program spectra are compared.

The remainder of the paper is organized as follows: Section II presents related work; Section III introduces the proposed approach; Section IV describes the experiments and analyzes the results obtained; and Section V draws some concluding remarks.

## II. RELATED WORK

Several researchers have studied various types of program spectra to predict program execution behavior. Podgurski et al. [6], [7], [11] present a set of techniques for clustering program executions. Bowring et al. [2] introduce a technique based on Markov models to distinguish failed executions from successful executions using branch coverage

information. Haran et al. [8] present several methods for classifying execution data as belonging to one of several classes. Brun and Ernst [3] identify dynamically discovered likely program invariants to reason about program execution behavior. Liblit et al. [10] use similar types of invariants for fault localization. Agrawal et al. [1] and Jones et al. [9] use statement coverage information to identify likely causes of failures. Chen et al. [4] keep track of components exercised during executions to pinpoint faulty components. Santelices et al. [13] empirically evaluate the performance of several types of program spectra in locating defects.

All of these approaches use program spectra collected purely by software instrumentation. On the other hand, in this work, substantial parts of the profiling task are pushed onto the hardware by leveraging hardware performance counters.

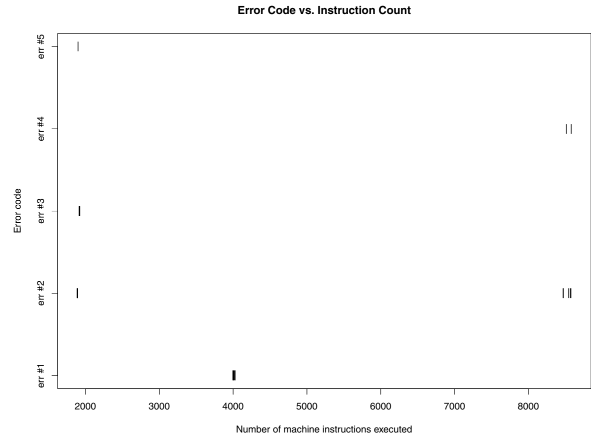


Figure 1. The number of machine instructions executed in the `socket` function.

## III. Proposed Approach

In this work, our aim is to automatically group similar failed executions into clusters. We instrument programs, collect hardware counters-based execution data at runtime, and then analyze the collected data to identify the similarities among executions.

Hardware performance counters are CPU-resident counters that record various events occurring on a CPU. Although the types of events recognized by CPUs may vary across platforms, today's general-purpose CPUs are capable of recording a wide variety of events, such as the number of instructions executed, the number of branches taken, the number of cache hits and misses experienced, etc. To activate hardware performance counters, special-purpose assembly-level instructions are used to pair a physical counter with an event of interest and to instruct the CPU to start profiling. After having activated hardware counters, the counter values can be read and reset at runtime, and the counters can be deactivated at will.

In this work use hardware performance counters-collected data are used as abstraction mechanisms for program executions. We conjecture that counter values can capture patterns occurring in executions and these patterns can then be used to automatically classify similar failed executions.

### A. Feasibility Study

To evaluate the plausibility of our hypothesis, we conducted a simple feasibility study using the `socket` system call as our subject.

The `socket` function creates an endpoint (i.e., socket) for communication. We experimented with two input parameters of this function: `domain` and `type`. The domain parameter specifies a communication domain (e.g., INET and INET6), and the type parameter specifies the communication semantics (e.g., connection-based and connectionless). These parameters take a value from a set of eleven and six discrete settings, respectively. Furthermore, the socket function returns a file descriptor on success and an error code on error. Note that this function is implemented at the OS kernel level.

All combinations of the parameters settings are exhaustively tested using 66 test cases. For each input combination, the number of machine instructions executed during the invocation of the `socket` function are measured. This was performed by reading the value of the corresponding hardware counter before and after calling the function and then computing the difference. Since not all input combinations were supported on our test platform, 51 of the test cases failed with five unique error codes; 30 of them with error #1, 11 with error #2, 7 with error #3, 2 with error #4, and 1 with error #5. Figure 1 visualizes the data obtained.

An initial question was: Can the total instruction counts be used to automatically cluster the function calls that return the same error code? To answer this question, we fed the instruction counts to the Weka's EM cluster analysis algorithm [15]. This algorithm provided us with three clusters. It was observed that each cluster was predominantly composed of failures with error code #1, #2, or #3. The rest of the failures with the error codes #4 and #5 was scattered among the clusters.

As is the case with all spectrum-based approaches, our approach depends on having an adequate number of executions to capture patterns and identify similarities. An inadequate number of test cases, or, similarly, an uneven distribution of failures across classes may degrade the accuracy of clusters. This is because clustering techniques themselves often perform poorly in such cases. in our feasibility study, an instantiation of this phenomenon was observed. Since there were only two failures with error #4 and only one failure with error #5, the EM clustering algorithm was not able to reliably cluster them. Since our goal is not to evaluate clustering techniques themselves, we in this work use failures as evenly distributed across classes as possible.

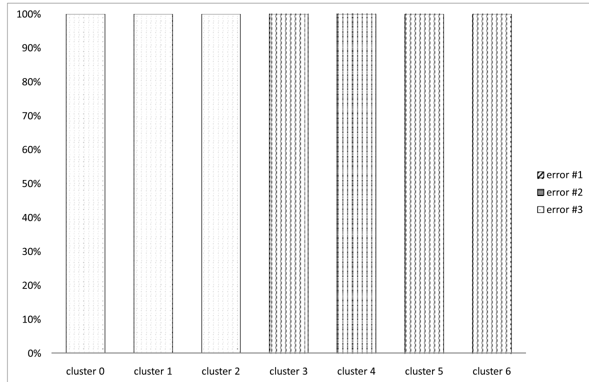


Figure 2. Clustering results after eliminating error #4 and #5

Then analysis are repeated without using the failures with the error codes #4 and #5. Figure 2 visualizes the resulting clusters. In this figure, the horizontal axis denotes the clusters obtained, and the vertical axis depicts the percentages of failures with the same error code in each cluster. As can be seen from this figure, all the clusters obtained were pure. That is, each cluster was composed only of failures with the same error code.

An in-depth analysis of the implementation of the socket function revealed that failed invocations that executed around 2000 machine instructions did so because of a simple check at the very top of the function that immediately returns an error code if the parameters provided are not within the supported range. Furthermore, the failed invocations that executed more number of instructions did so because if an error occurs after a certain point into the execution, the OS kernel releases all the resources that have been allocated so far. The farther into the execution an error occurs, the more activities there are to be performed to release the resources. For example, for the failed invocations that executed around 4000 instructions, only the kernel resources related to the socket data structure were released, whereas, for the failed invocations that executed more than 8000 instructions, more resources were reclaimed on behalf of the kernel loadable modules that implement the communication protocols. The instruction counts data collected from hardware performance counters was able to identify these patterns.

However,some cases where the failures with the same error code were distributed among multiple clusters, were observed. Automatic failure classifications obtained by using data-driven analysis approaches, such as the one presented here, are often not exact [11]. This is because *correlations* may not always indicate *causations*. However, many empirical studies strongly suggest that, despite the caveats, such analysis results are of great practical importance to practitioners [2], [5], [9], [10], [11], [16].

Although the results of this feasibility study are by no means conclusive, it increased our beliefs that hardware counters-collected data can be used to identify patterns in executions. More details about the proposed approach are provided in the remainder of this section.

Table I
AN EXAMPLE DATA TABLE

| Test | f1 | f2 | ... | pass/fail |
|---|---|---|---|---|
| 1 | 6660 | 3805 | ... | pass |
| 2 | -1 | 53445 | ... | fail |
| ... | ... | ... | ... | ... |

### B. Program Spectra

In this work, executions at the level of function invocations are profiled using a hardware performance counter called, TOT_INS. The TOT_INS counter records the number of machine instructions executed. To map the counter values with function invocations, the value of the counter is read before and after an invocation and attribute the difference to the invocation.

Given a program and its test suite, all the test cases on the program are executed and the program spectra are collected. Table I depicts, as an example, a portion of program spectra collected in a study for a subject application using the TOT_INS counter. Each row in this table represents a single execution of the application with a test case. The first and the last columns depict the test index and whether the test passed or failed, respectively. Each remaining column corresponds to a function and the values stored in the table depict the total number of machine instructions executed in functions. The counter values mapped to a function are aggregated across all invocations of the function. For example, the first row depicts a successful execution of test #1, in which there were a total of 6660 and 3805 machine instructions executed in functions `f1` and `f2`, respectively. The value of `-1` is used in these tables to indicate the functions that were not invoked in executions. These tables are referred as *data tables* in the remainder of the paper.

### C. Feature Selection and Cluster Analysis

The accuracy and performance of clustering techniques often suffer as the dimension of the data being analyzed increases. To alleviate this problem, a feature selection algorithm is leveraged to select a small, but highly relevant subset of functions. By following a similar approach presented in [11], this step is carried out by identifying those functions that are most capable of reliably distinguishing failed executions from successful executions.

For the feature selection step, a classifier subset evaluator (CSE) algorithm [15] is used. This algorithm iteratively evaluates potential function subsets, estimates the predictive power of each subset using a classifier, and outputs the best function subset that has been encountered. As the classification algorithm to be used with the CSE algorithm the J48 classification tree algorithm [15] is utilized. Once

Table II
SUBJECT APPLICATIONS USED IN THE EXPERIMENTS

| subject application | number of LOC | functions | number of defects | total tests | passing tests | failing tests |
|---|---|---|---|---|---|---|
| flex | 10459 | 162 | 16 | 396 | 318 | 78 |
| sed | 14427 | 255 | 6 | 370 | 354 | 16 |

the set of highly relevant functions is identified, the rest of the functions is ignored.

Then the remaining data is fed to a clustering algorithm, called Expectation Maximization (EM) clustering algorithm [15]. This algorithm is used, since, unlike many other clustering algorithms, it does not require the number of clusters to be known in advance. The EM algorithm is a probabilistic algorithm working in two alternating steps; expectation and maximization. In the expectation step, the probabilities of cluster memberships are computed. In the maximization step, the overall likelihood of the data (given the clusters) is maximized by reconfiguring the clusters. The algorithm stops when the difference between successive iterations is smaller than a threshold value. The output is a set of clusters grouping similar failed executions together.

## IV. EXPERIMENTS

We conducted an initial set of feasibility studies to evaluate the accuracy of our proposed approach in classifying failures.

### A. Subject Applications

In these experiments two open source applications are used as our subject applications: `flex` and `sed`. These applications are widely-used utility applications on UNIX/Linux platforms. The `flex` application generates lexical analyzers, and the `sed` application filters and transforms text as a stream editor

Our subject applications were taken from an independent defect repository, called Software-artifact Infrastructure Repository (SIR) (sir.unl.edu). Each subject application had its own test suite and test oracles, which were utilized in our experiments. Table II provides some statistics about the subject applications used in the experiments.

### B. Experimental Setup

The SIR repository also provided us with a set of known defects for our subject applications. Each defect was identified with a unique defect identifier, and the defects were able to be activated individually as needed. The `flex` and `sed` applications used in the experiments had 16 and 6 known defects, respectively.

First all the defects are activated in our subject applications and their test suites are executed . Then it is determined whether the tests were successful or not with the help of the test oracles that came with the test suites.

To collect the program spectra from these executions, the Scalasca tool (www.scalasca.org) is used. This tool provided us with dynamic call trees observed in executions. Each function invocation in these trees was annotated with the number of machine instructions executed in it. These call trees are parsed and a data table is created for each subject application, such as the one presented in Table I.

Once the data tables were ready, first the feature selection step is performed.Then all the successful executions are excluded from our data tables and only the functions chosen in the feature selection step are kept. Finally, the failed executions are automatically clustered as explained in Section III-C. These steps were carried out by using the Weka data mining tool [15]. All the experiments were performed on a Pentium D machine with 1GB of RAM, running the CentOS 5.2 operating system.

### C. Comparative Studies

In our experiments, the accuracy of our hardware counters-based spectrum is compared (TOT_INS) to that of three different types of spectra, namely CALL_SWT, TIME, and VISIT.

The CALL_SWT spectrum records the functions invoked during executions. Each execution record in the CALL_SWT spectrum reflects whether a function is invoked (indicated by 1) or not (indicated by -1) during the execution. Note that our hardware counters-based TOT_INS spectrum also leverages function coverage information, in addition to the instruction counts. By comparing the TOT_INS spectrum to the CALL_SWT spectrum, we aim to single out the effect of using instruction counts in clustering failures from the effect of using function coverage information. Since the only difference between the TOT_INS and CALL_SWT spectra is the presence of instruction counts, differences between the performance of these spectrum types can safely be attributed to using instruction counts.

The TIME spectrum is the trace of function execution times measured at the level of nanoseconds in terms of the CPU allocation times. In an earlier study [16], time spectra was used to locate defects. Empirical evaluations suggested that execution times can be used to capture patterns in program executions. However, one downside of this approach was the inaccuracies and imprecisions associated with measuring execution times. In this work, we conjecture that, being a related, but more precise and accurate spectrum, the TOT_INS spectrum is capable of clustering failures better than the TIME spectrum.

Table III
EMPIRICAL RESULTS

| sut | | TOT_INS | | | VISIT | | | CALL_SWT | | | TIME | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | class1 | class2 | class3 | class1 | class2 | class3 | class1 | class2 | class3 | class1 | class2 | class3 |
| | cluster1 | 0 | 100 | 0 | 100 | 0 | 0 | 0 | 50 | 50 | 54,5 | 0 | 45,5 |
| | cluster2 | 12,5 | 0 | 87,5 | 0 | 0 | 100 | 100 | 0 | 0 | 56 | 0 | 44 |
| | cluster3 | 0 | 0 | 100 | 100 | 0 | 0 | - | - | - | 0 | 83,8 | 16,2 |
| flex | cluster4 | 100 | 0 | 0 | 0 | 100 | 0 | - | - | - | - | - | - |
| | cluster5 | 100 | 0 | 0 | 100 | 0 | 0 | - | - | - | - | - | - |
| | cluster6 | 0 | 0 | 100 | 0 | 0 | 100 | - | - | - | - | - | - |
| **RI** | | | 0.87 | | | 0.89 | | | 0.78 | | | 0.74 | |
| | cluster1 | 0 | 100 | - | 50 | 50 | - | 50 | 50 | - | 50 | 50 | - |
| sed | cluster2 | 0 | 100 | - | - | - | - | - | - | - | - | - | - |
| | cluster3 | 100 | 0 | - | - | - | - | - | - | - | - | - | - |
| **RI** | | | 0.87 | | | 0.47 | | | 0.47 | | | 0.47 | |

The VISIT spectrum, on the other hand, records the number of times each function is invoked during executions. Podgurski et al. [11] use this spectrum type to classify failures. In our experiments, it is observed that the TOT_INS and VISIT spectra are highly correlated. An analysis of the data collected in our experiments revealed that the total number of machine instructions executed in a function during an execution is correlated with the number of times the function is called with a correlation coefficient of $0.98$ on average. Although the data values are correlated, we hypothesize that the TOT_INS spectrum convey more information for reliably classifying failures.

### D. Evaluation Framework

To be able to evaluate the quality of the resulting clusters, we required to pinpoint the causes of failures so that we could quantify to which extent the clusters contain failures caused by the same defect. Since each subject application used in our experiments had several defects, this posed an issue for us. The information about which subset of all activated defects was responsible for the manifestation of a given failure was not known to us.

In this work the delta debugging algorithm [17] is leveraged to identify failure inducing defects. In our implementation of this algorithm, each atomic change (a concept defined in delta debugging) corresponds to the activation of a single defect. At a very high level, this algorithm explores the subsets of all the defects in an iterative fashion until it finds a "minimal" set of defects that, once activated, causes the program to produce the same faulty output as is produced when all the defects are activated. The output is a subset of all the defects such that deactivating a single defect in this subset would make the program produce a different output.

The failure inducing defects for each failed execution are identified in our experiments using the delta debugging algorithm. Then the failures that are caused by the same set of defects are grouped into classes. Finally these classes are used to evaluate the accuracy of automatically identified clusters by leveraging a well-known information retrieval-based metric, called rand index [12].

Rand index (RI) is computed as follows;

$$RI = \frac{TP + TN}{TP + FP + FN + TN}$$

A true positive (TP) decision assigns two similar failures (i.e., failures with the same class) to the same cluster, a true negative (TN) decision assigns two dissimilar failures (i.e., failures with different classes) to different clusters, whereas a false positive (FP) decision assigns two dissimilar failures to the same cluster, and a false negative decision (FN) assigns two similar failures to different clusters. In effect, RI measures the ratio of decisions that are correct. It takes a value between 0 and 1. The higher the value of RI, the better the clustering is.

### E. Data & Analysis

The delta debugging algorithm identified three classes of failures for the `flex` application, and two classes of failures for the `sed` application. The number of clusters obtained for the various types of program spectra used in the experiments varied. Table III summarizes the results of our experiments.

In this table, columns indicate the classes of failures, and rows depict the clusters. The cell values are the percentages of failure classes included in automatically identified clusters. For example, `cluster1` obtained by using the TIME spectrum on the `flex` application is composed of $54,5\%$ `class1` and $45.5\%$ `class3` failures. Furthermore, `class1` failures are distributed across two clusters, namely `cluster1` and `cluster2`. The dash characters indicate data not applicable. The RI values are provided for each pair of subject application and spectrum type.

Comparing the TOT_INS spectrum to the CALL_SWT spectrum revealed that using total instruction counts provided by hardware performance counters improved the accuracy of clustering by about 38% on average over not using them. The average RI value was 0.87 for the TOT_INS spectrum and 0.63 for the CALL_SWT spectrum.

Comparing the TOT_INS spectrum to the TIME spectrum exposed that, being a more precise spectrum compared to the time spectrum, our hardware counters-based spectrum

improved the accuracy by 43% on average. The average RI values were 0.87 and 0.61 for the TOT_INS and TIME spectrum, respectively.

Then the accuracy of the clusters obtained from TOT_INS spectrum is compared to those obtained from VISIT spectrum. It was observed that TOT_INS spectrum increased the accuracy by 28% on average. The average RI values were 0.87 and 0.68 for the TOT_INS and VISIT spectrum, respectively. Although, as discussed in Section IV-C, the TOT_INS spectrum was highly correlated with the VISIT spectrum, visual investigation of the data and the resulting clusters revealed that instruction counts via summarizing computational activities occurring in function invocations provided more information compared to using invocation frequencies only. As our results indicate, the additional information was helpful in improving the accuracy.

## V. Concluding Remarks

In this work we leveraged a hardware performance counters-based program spectrum to automatically group failures that stem from closely related causes into clusters. Hardware counters have been used for performance analysis of software systems in the past. By contrast, we in this work use them as abstraction mechanisms for program executions.

We conducted a feasibility study to evaluate the proposed approach. In this study we used two widely-used applications as our subject applications and compared the accuracy of hardware counters-based spectra to those of three different types of program spectra.

All empirical studies suffer from threats to their internal and external validity. For this work, we are primarily concerned with threats to external validity since they limit our ability to generalize our results. One threat concerns the representativeness of the subject applications used in the experiments. Although they are all real-life applications, they only represent two data points. A related threat concerns the representativeness of the defects used in the experiments. Although our subject applications were taken from an independent defect repository which has been leveraged by many related studies in the literature, they only represent a subset of defects.

Keeping these limitations in mind, we believe that our results, although preliminary, supports our basic hypothesis: Hardware performance counters-collected data can be used to classify program failures.

Many software quality assurance approaches, such as fault localization, failure prediction, software security assurance, and in-the-field quality assurance, depend on finding patterns in program executions and identifying similarities to these patterns and/or deviations from them. Our ultimate goal beyond the scope of this work is to provide reliable means of capturing patterns in executions as unobtrusively as possible. Our motivation behind leveraging hardware performance counters for this purpose is a simple one: Hardware is one of the best places to profile executions with as little runtime overhead as possible. We, therefore, are continuing to investigate how hardware counters-collected data can serve as abstraction mechanisms for program executions in various software quality assurance approaches.

## VI. Acknowledgments

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme (FP7-PEOPLE-IRG-2008).

## References

[1] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE Conference Proceedings*, 1995.

[2] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *ISSTA Proceedings*, pp. 195-205, 2004.

[3] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE Conference Proceedings*,pp. 480-490, 2004.

[4] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *International Conference on Dependable Systems and Networks*, 0:595–604, 2002.

[5] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE Conference Proceedings*, pp. 34-44, 2009.

[6] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *FSE Conference Proceedings*, pp. 246-255, 2001.

[7] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE Conference Proceedings*, pp. 339-348, 2001.

[8] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. *SIGSOFT Softw. Eng. Notes*, 30(5):146–155, 2005.

[9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE Conference Proceedings*, pp. 467-477, 2002.

[10] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI Conference Proceedings*, pp. 141-154, 2003.

[11] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang. Automated support for classifying sw failure reports. In *ICSE Conference Proceedings*, pp. 465-474, 2003.

[12] W. R. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[13] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE Conference Proceedings*, pp. 56-66, 2009.

[14] S. Singer, K. Gross, J. Herzog, S. Wegerich, and W. King. Model-based nuclear power plant monitoring and fault detection: theoretical foundations. In *ISAP Conference Proceedings*, pp. 60-65, 1997.

[15] I. H. Witten and E. Frank. Data Mining: Practical Machine Learning Tools and Techniques (Second Edition). *Morgan Kaufmann Publishers*, San Francisco, 2005.

[16] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In *ICSE Conference Proceedings*, pp. 81-90, 2008.

[17] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.