Using Hardware Performance Counters for Fault Localization

Cemal Yilmaz
Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul, Turkey
cyilmaz@sabanciuniv.edu

***Abstract*—In this work, we leverage hardware performance counters-collected data as abstraction mechanisms for program executions and use these abstractions to identify likely causes of failures. Our approach can be summarized as follows: Hardware counters-based data is collected from both successful and failed executions, the data collected from the successful executions is used to create normal behavior models of programs, and deviations from these models observed in failed executions are scored and reported as likely causes of failures. The results of our experiments conducted on three open source projects suggest that the proposed approach can effectively prioritize the space of likely causes of failures, which can in turn improve the turn around time for defect fixes.**

***Keywords*-debugging aids; fault localization; hardware performance counters.**

I. Introduction

Identifying root causes is the hardest, thus the most expensive, component of program debugging. Developers often observe symptoms of failures, hypothesize a set of potential causes in an ad hoc manner, and then iteratively verify and refine their hypotheses until root causes are located. Obviously, this process can be quite tedious and time consuming.

Many fault localization approaches have been proposed in the literature to help developers quickly pinpoint the root causes of failures. These approaches aim to automatically reduce and/or prioritize the space of likely causes so that developers have a promising (and hopefully small) set of starting points to reliably form their initial failure hypotheses.

One type of automated fault localization approach, which is also the focus of this paper, compares successful and failed program executions to identify potential causes of failures. These approaches operate in the same way: Specific types of execution information, called program spectra, are collected from both successful and failed program executions, and similarities in the failed executions and/or deviations from the successful executions are identified and scored as potential causes of failures. The fundamental assumption behind these approaches is that there are identifiable and repeatable patterns in the behavior of successful and failed executions and that similarities and deviations from these patterns are highly correlated with the causes of failures.

One factor that affects the success of program spectrum-based fault localization approaches is the type of program spectrum used to identify patterns in executions. A common characteristic of most existing types of program spectra [6], [10], [11] is that they focus only on a very specific feature of program executions, such as code statements covered [10] or function call sequences [6] observed, and collect precise information about that feature at the expense of others. For example, information about function call sequences [6] would tell which sequences of function invocations were observed in executions, but cannot tell what happened in each invocation.

One reason behind leveraging such simple types of program spectra is that a program execution is a complex event. It consists of a sequence of state transformations each of which comes to existence as a result of complex interactions between many factors. Although, in theory, one may collect quite detailed data from executions, the runtime overhead cost (both time and space) of collecting such large amount of heterogeneous data often makes this approach an impractical one. Consequently, it is still an open question what to collect and how to analyze the collected data to identify meaningful patterns in executions.

In a previous work [16], rather than individually dealing with every factor that affects executions, we, in a sense, dealt with all of them at once by leveraging a different type of program spectrum, called a time spectrum. A time spectrum can be considered as a trace of function invocation times. Note that, at a coarse level, the execution time of a function accounts for everything that happens in the function, including all complex interactions. We call this approach Time Will Tell (TWT). Our empirical evaluations of the TWT approach conducted on three real-life applications suggest that the time spectra can effectively capture patterns occurring in executions [16].

One downside of the TWT approach, however, was that the resolution of the time measurements is limited by the resolution of the software/hardware clocks available. We observed that a low resolution in measurements could prevent us from detecting some important patterns in executions. Another shortcoming of the approach was that the time to execute a piece of code may vary from execution to execution because of the noise imposed by the underlying platform. We observed that, if not taken into account, noise in measurements could lead to detecting spurious patterns.

To overcome these shortcomings, we in this work leverage hardware performance counters to collect a related, but more precise and accurate type of program spectrum than the time spectrum. A hardware performance counter is a CPU-resident counter that precisely and accurately records various events occurring on a CPU. Hardware performance counters have been extensively leveraged for performance analysis of software systems (e.g., *hot spot* analysis). By contrast, we in this work use them in a novel way to locate functionality-related defects. The results of our experiments conducted on three open source projects suggest that, for the systems used, hardware counters-based program spectra effectively prioritized the space of likely causes of failures; and they provided better diagnosis reports than those provided by time spectra.

The remainder of this paper is organized as follows: Section II briefly discusses the related work; Section III introduces the proposed approach; Section IV describes the experiments we conducted; and Section V presents concluding remarks.

II. Related Work

Several types of program spectra have been proposed in the literature for fault localization. Agrawal et al. leverage statement coverage information to identify potential causes of failures [1]. They report statements that appear in a failed execution but not in any of successful executions as potential sources of failures. Pan et al. provide similar heuristics by leveraging dynamic program slices [12]. Jones et al. improve Agrawal's work by allowing some tolerance for faulty statements to be occasionally executed by successful runs [10]. Dallmeier et al. identify and score suspicious sequences of method invocations occurring in failed executions [6]. Reps et al. study the use of path spectra in program debugging with applications to the year 2K problems [14]. Harrold et al. [9] and Santelices et al. [15] empirically evaluate the performance of several types of program spectra in locating defects. Renieris et al. claim that comparing a failed execution only to those successful executions that most resemble the failed execution can improve the accuracy of fault localization approaches, as opposed to comparing it to all or arbitrary successful executions [13].

Liblit and colleagues introduce the concept of cooperative bug isolation [11]. They leverage partial execution traces collected from different program runs to locate defects. They use this idea, in one work, to detect violations of dynamically identified likely program invariants [11], and, in another work, to identify suspicious paths followed by failed executions [3].

Cleve and Zeller introduce an algorithmic approach to debugging, called delta debugging [5]. Zeller and colleagues later use the delta debugging algorithm to simplify failure inducing program inputs [18], to capture cause-effect chains [17], and to isolate failure-inducing thread schedules [4].

Our work is different than the ones discussed above in its use of a novel, hardware performance counters-based program spectrum to identify likely causes of failures.

III. Proposed Approach

A hardware performance counter is a part of a CPU that counts various events occurring on the CPU. 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.

Hardware performance counters have been traditionally used in performance debugging to identify hotspots in the program code. For this purpose, the values of counters serve as numeric values. By contrast, we use hardware performance counters in many different ways. First, we leverage them as abstraction mechanisms for program executions, not just as numeric values as is the case with performance debugging. Second, rather than looking for hotspots, we identify those code segments that perform "suspicious" amounts of computational activities and report them as potential sources of failures.

Hardware performance counters are by default inactivated. To activate them, a code indicating the type of event to be counted and the physical counter to be used for counting is written to a register and then the CPU is instructed to start the counting process. There are often many physical counters present in a CPU and these counters can individually be paired with any event known to the CPU. Once activated, hardware counters count the events of interest during program executions and store the counts in a set of special purpose registers. These registers can then be read and reset, and the performance counters can be deactivated as needed. Since the counting process is performed on an instruction-by-instruction basis, hardware counters provide quite precise and accurate values.

One challenge we faced in leveraging hardware counters for the task of fault localization was that the counters do not distinguish between the instructions issued by different processes. To overcome this issue, we used an OS kernel driver, called `perfctr` (linux.softpedia.com). This driver provided us with virtual counters that count instructions on a per-process basis.

Another challenge we faced was that, since hardware counters carry out the profiling task within the hardware, they are not aware of the program entities being executed at the time of counting. For example, they cannot tell to which program function the instructions being counted belong. In this work, we associate hardware counters with function invocations. We chose to profile executions at this granularity level, since functions provide well-defined code and functionality boundaries.

Our approach takes as input a set of successful executions and a failed execution. At a very high level, it can be summarized as follows: 1) hardware counters-based program spectra are collected from the successful and failed executions; 2) behavior models that capture the patterns observed in the successful executions are created using the spectra collected from the successful executions; 3) deviations from these models that occurred in the failed execution are identified and scored as potential causes of the failure. The output of the approach is a ranked list of function invocations observed in the failing run, sorted in descending order by the level of their suspiciousness.

The remainder of this section describes how we addressed each step of the approach.

A. *Program Spectra Collected*

In this work we chose to leverage a hardware performance counter, called TOT_INS, to profile executions. The TOT_INS counter records the number of machine instructions executed (i.e., retired).

Although numerous types of hardware counters exist, we decided to use only the TOT_INS counter for a number of reasons. First, being an initial study in the area, we in this work opted to keep the design of our experiments simple by leveraging only one counter. Second, the values of the TOT_INS counter reflect all computational activities occurring in executions. All programs in one form or another are compiled into machine instructions and then executed. The amount of computational activities performed in a piece of code is directly proportional to the number of instructions the code is compiled into. Consequently, the value of the TOT_INS counter associated with a piece of code can provide clues about the branches taken, methods invoked, and paths exercised during the execution of the code. Functions, for example, depending on their input arguments as well as the state of the program, may exercise different paths which may potentially be compiled into different number of instructions. Therefore, in theory, these differences should be reflected on the values of the TOT_INS counter. While we chose to use the TOT_INS counter, the proposed approach is equally applicable to other types of events.

As discusses in Section III, the granularity of our analysis is at the level of a function invocation. To associate the TOT_INS counter with function invocations, we read the value of the counter before and after a function invocation and attribute the difference to the invocation.

In an earlier study, we observed that, since hardware counters provide coarse level information about executions, augmenting them with some context information generally improves their ability to identify patterns in executions. Therefore, by following a similar strategy we presented in [16], we augment the hardware counters-collected data

Table I
AN EXAMPLE DATA TABLE.

body	**f68**	**f89**	**...**
15082	2595	4800	...
3612	1480	1455	...
...	...	...	...

with caller-callee information in this work. That is, we itemize the number of machine instructions executed in a function invocation to reflect the number of instructions executed in the body of the function and in each callee function. The instruction counts of a callee function are aggregated over all the invocations of the callee. If the callee is called multiple times, the sum of its associated counter values is used.

As an example, Table I provides a portion of the TOT_INS spectrum collected for a function (`f86`) in a study. Each row in this table represents a single invocation of function `f86` in a successful execution. For example, the first row depicts an invocation of this function where it executed a total of 15082 instructions in its body, a total of 2595 instructions in the callee function `f68`, and a total of 4800 instructions in the callee function `f89`. These tables are referred to as *data tables* in the remainder of the paper.

B. *Creating Behavior Models and Scoring Deviations*

In this work we create one behavior model for each function defined in a program. Hardware performance counters-based spectra are collected from successful executions of the program, all invocation records obtained for the same function are gathered in a data table, such as the one presented in Table I, and the resulting data table is used to create a behavior model for the function. The behavior model of the program is then considered to be the collection of these individual models. In effect, these models capture the normal behavior of the program as it is observed in successful executions.

For a given failed execution, we first create the data tables in the same manner as with successful executions (i.e., one data table per function). We then feed these data tables to their corresponding normal behavior models. For each invocation record (i.e., each row in a data table), the output of a model is a score quantifying the deviation of the invocation from the model.

To create the normal behavior models and compute the deviations from them, we use a density-based outlier detection technique, called local outlier factor (LOF) [2]. LOF assigns to each invocation record collected from a failed execution a score of being an outlier. The score of an invocation record depends on how isolated the record is with respect to its k-distance neighboring invocation records collected from successful executions ($k = 3$ in our case). Higher scores signal deviations, whereas lower scores are a sign of normality.

Table II
SUBJECT APPLICATIONS USED IN THE EXPERIMENTS.

subject application	**LOC**	**number of functions**	**number of versions**	**total tests**	**passing tests**	**failing tests**
flex	10459	162	30	16679	16007	672
sed	14427	255	24	9640	9016	624
grep	10068	146	13	9537	8847	690

Once the score of every function invocation encountered in a failed execution is computed, they are sorted in descending order and presented as a diagnosis report for the failure.

C. Evaluation Framework

As with all the fault localization approaches, we assign a score to every diagnosis report we generate to evaluate the accuracy of our approach.

In our scoring scheme, by following a similar approach presented in [6], [10], [11], [15], suspicious function invocations are examined starting with the most suspicious one and working down to the least. The score of a diagnosis report for a failed execution is then defined as the percentage of the function invocations observed in the failed execution that need to be examined before the defect(s) can be located, assuming that defects are recognized on sight. The lower the score, the better the diagnosis report is.

IV. EXPERIMENTS

We conducted a set of feasibility studies to evaluate the proposed approach.

A. Subject Applications

In these experiments, we used three open source real-life applications as our subject applications: `grep`, `flex`, and `sed`. These widely used applications print lines matching a pattern, generate fast lexical analyzers, and filter and transform text as a stream editor, respectively. Table II provides some statistics for the subject applications.

All the subject applications were taken from an independent defect repository, called Software-artifact Infrastructure Repository (SIR) [7]. The SIR repository provided us with several versions of these subject applications with known defects, and a test suite for each version.

B. Operational Model

In our experiments, we compare the performance of our hardware counters-based spectra (TOT_INS) to that of two different types of spectra, namely TIME and CALL_SWT spectra.

As discussed in Section I, the roots of this research stem from one of our previous works [16] in which we leveraged execution times of functions to identify likely causes of failures. One objective of ours in this work is to improve the quality of the failure diagnostics by replacing time measurements with more precise and accurate measurements obtained from hardware performance counters. Therefore, to evaluate the gains from our improved approach, we compare the performance of the TOT_INS spectrum to that of the TIME spectrum. TIME spectra used in the experiments are collected as traces of function execution times measured at the level of nanoseconds in terms of the CPU allocation times.

The CALL_SWT spectrum, on the other hand, records the functions invoked during executions. For each callee function, an invocation record in the CALL_SWT spectrum reflects whether the callee function is called (indicated by 1) or not (indicated by -1) in the invocation. Note that our hardware counters-based TOT_INS spectrum leverages the caller-callee 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 localizing defects from the effect of using caller-callee 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.

Furthermore, to demonstrate that our results are not by chance, we create random diagnosis reports. For each failure, we create 100 random reports, score them, and use the average scores in our analysis.

C. Experimental Setup

We evaluated the performance of various types of program spectra used in the experiments in the same way. Although the data tables created for each spectrum type were structurally the same, the type of information stored in them varied depending on the spectrum type.

To collect the program spectra, we used the `Scalasca` tool [8]. This tool provided us with all the information needed to create the data tables. To create normal behavior models and to compute deviations from them, we used the `dprep` library of the `R` statistical computing application (www.r-project.org). This library provided us with an implementation of the LOF algorithm. All the experiments were performed on a dual Intel Xeon machine with 2GB of RAM, running the CentOS 5.2 operating system.

D. Data and Analysis

We evaluated our approach using a total of 33870 successful executions and 1986 failed executions (Table II). For each

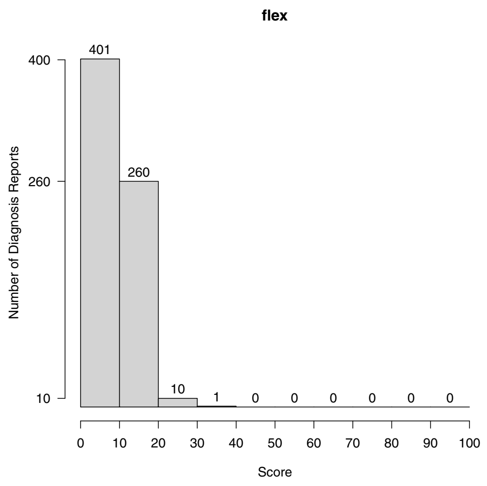

Figure 1. Histogram of the scores obtained on `flex`.

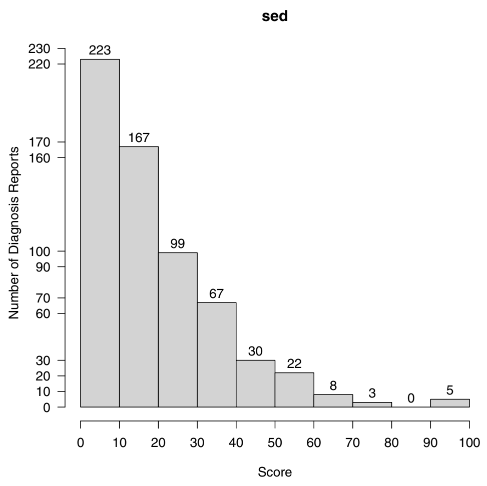

Figure 2. Histogram of the scores obtained on `sed`.

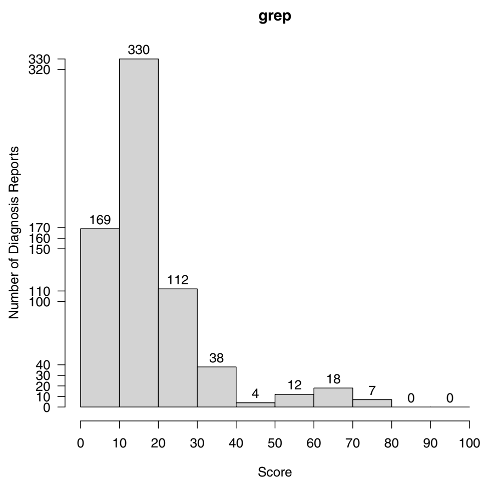

Figure 3. Histogram of the scores obtained on `grep`.

Table III
AVERAGE SCORES OBTAINED USING DIFFERENT TYPES OF SPECTRA

subject	CALL_SWT	TIME	TOT_INS
flex	45.91	14.22	**8.28**
sed	50.91	21.69	**19.09**
grep	81.69	21.38	**18.54**

failed execution, we automatically generated a diagnosis report, and then assigned a score to it.

We first assessed the performance of the TOT_INS spectrum in isolation. Figure 1 - 3 report the histograms of scores we obtained by using the TOT_INS spectrum. In these plots, the horizontal axis denotes a score interval and the vertical axis denotes the number of diagnosis reports that fall into each score interval.

As these figures depict, out of 1986 failures across our subject applications, 40% of the defects were located by examining up to 10% of the function invocations, 78% by examining up to 20%, and 89% by examining up to 30% of the function invocations in the failed executions. Comparing these results to the results of our random experiments revealed that our results are not by chance. Only 0.3% and 6% of the diagnosis reports obtained in the random experiments received a score below the cutoff scores of 20% and 30%, respectively. None of the scores was below 10%.

We then compared the performance of the TOT_INS spectrum in locating defects to that of CALL_SWT spectrum. Table III provides the average scores we obtained for different types of spectra used in the experiments. We observed that using the information of number of instructions executed in function invocations (the TOT_INS spectrum) greatly improved the quality of diagnosis reports compared to not using it (the CALL_SWT spectrum). The scores of the diagnosis reports were improved by about 63% to 82% (74% on average).

Once our results strongly suggested that the success of our hardware counters-based TOT_INS spectrum is neither by chance nor by the use of the caller-callee information, we compared the performance of the TOT_INS spectrum to that of the TIME spectrum. We observed that, for all the subject applications used in the experiments, the TOT_INS spectrum on average provided better diagnosis reports than the TIME spectrum (Table III). For example, the average score of the diagnosis reports obtained for `flex` was 8.28 for the TOT_INS spectrum and 14.22 for the TIME spectrum. This corresponds to about 42% improvement in the scores. Overall, the TOT_INS spectrum provided about 12% to 42% (22% on average) improvement in the scores of the diagnosis reports.

V. CONCLUDING REMARKS

The root of this research stem from one of our previous research [16], where we leveraged function execution times (time spectrum) to identify likely causes of failures. One downside of using time, however, was the imprecisions and inaccuracies associated with measuring execution times.

To overcome this shortcoming, we in this work leverage hardware performance counters-collected data as abstraction mechanisms for program executions, which provides us with related, but more precise and accurate program spectra than time spectra. Hardware performance counters have been extensively used for performance debugging in the past. By contrast, this work leverages them in a novel way to locate functionality-related defects.

We conducted a set of experiments using three subject applications to evaluate our approach. In these experiments, we leveraged a hardware counter (TOT_INS) that records the number of machine instructions executed.

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 three data points. A related threat concerns the representativeness of the defects used in the experiments. Although all the defects studied were taken from an independent defect repository that has been leveraged by many related studies in the literature [3], [6], [10], [13], they represent only a subset of all potential defects. Keeping these limitations in mind, we believe that our study supports our basic hypothesis: Hardware performance counters-based program spectra can be used to prioritize likely causes of failures, and, being a more precise and accurate spectrum, they can provide better diagnostics than time spectra.

We arrived at this conclusion by comparing the performance of our hardware performance counters-based TOT_INS spectrum to that of two different types of spectra; time spectra (TIME) and function coverage spectra (CALL_SWT). Compared to the CALL_SWT spectrum, the TOT_INS spectrum improved the scores of the diagnosis reports by 74% on average, suggesting that it is the instruction counts not the function coverage information leveraged in the TOT_INS spectrum that helped locate the defects. Compared to the TIME spectrum, the TOT_INS spectrum improved the scores by 22% on average, suggesting that the TOT_INS spectrum provided better diagnosis reports than the TIME spectrum.

We believe that this line of research is novel. As a next step, we are currently in the process of leveraging different types of hardware counters for fault localization. We also investigate how hardware counters-based spectra can serve as abstraction mechanisms for program executions in various software quality assurance approaches, such as failure detection, failure prediction, security assurance, and in-the-field quality assurance approaches.

Acknowledgment

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*, pages 143–151, 1995.

[2] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. Lof: identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, 2000.

[3] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. HOLMES: Effective statistical debugging via efficient path profiling. In *ICSE Proceedings*, pages 34–44, 2009.

[4] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. *SIGSOFT Softw. Eng. Notes*, 27(4):210–220, 2002.

[5] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

[6] V. Dallmeier, C. Lingding, and A. Zeller. Lightweight defect localization for java. In *ECOOP Conference Proceedings*, pages 528–550, 2005.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Soft. Eng.*, 10(4):405–435, 2005.

[8] M. Geimer, F. Wolf, B. J. Wylie, E. Abraham, D. Becker, and B. Mohr. The scalasca performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing*, pages 51–65, 2008.

[9] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between fault-revealing test behavior and differences in program spectra. *STVR Journal of Software Testing, Verification, and Reliability*, 10(3):171–194, 2000.

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

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI Conference Proceedings*, pages 12–15, 2005.

[12] H. Pan and E. Spafford. *Technical Report SERC-TR-116-P, Purdue University*, 1992.

[13] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE Proceedings*, page 30, 2003.

[14] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *ESEC/FSE Conference Proceedings*, pages 432–449, 1997.

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

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

[17] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT/FSE Proceedings*, pages 1–10, 2002.

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