Combining Hardware and Software Instrumentation to Classify Program Executions

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

Adam Porter
University of Maryland
College Park, MD 20742
aporter@cs.umd.edu

ABSTRACT

Several research efforts have studied ways to infer properties of software systems from program spectra gathered from the running systems, usually with software-level instrumentation. While these efforts appear to produce accurate classifications, detailed understanding of their costs and potential cost-benefit tradeoffs is lacking. In this work we present a hybrid instrumentation approach which uses hardware performance counters to gather program spectra at very low cost. This underlying data is further augmented with data captured by minimal amounts of software-level instrumentation. We also evaluate this hybrid approach by comparing it to other existing approaches. We conclude that these hybrid spectra can reliably distinguish failed executions from successful executions at a fraction of the runtime overhead cost of using software-based execution data.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Reliability, Measurement, Experimentation

Keywords

Hardware performance counters, Failure detection, Software quality assurance.

1. INTRODUCTION

In recent years, numerous researchers have proposed data-driven techniques to improve the quality of deployed software. At a high level these techniques typically follow the general approach of lightly instrumenting the software system, monitoring its execution, analyzing the resulting data, and then acting on the analysis results. Some example applications of this general approach include identifying likely fault locations, anticipating resource exhaustion, and categorizing crash reports as instances of previously reported bugs [24, 13, 23, 22, 12, 18, 15].

Another specific application, which is the focus of this paper, is determining whether execution data taken from a deployed system comes from a failed or a successful execution. Following the approach described above, existing solutions to this problem typically add instrumentation to the system source code or binaries that collect specific types of execution information, called program spectra, whenever the system runs. This data is periodically collected and analyzed. These analysis techniques attempt to identify patterns in the returned program spectra that are highly correlated with failed executions. The resulting models can later be applied to new executions, taken from deployed software instances whose failure status is unknown, as a way of classifying whether the original execution failed. Such information can then be used, for instance, to determine whether the deployed software system is experiencing an already reported bug.

One fundamental assumption these and similar 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 presence or absence of failures. Previous efforts, in fact, appear to support this assumption, successfully applying a variety of program spectra to classify failed program executions [7, 8, 11, 3].

Another less well-understood issue, however, is how best to limit the total costs of implementing these approaches and whether and how tradeoffs can be made between cost and classification accuracy. This issue is important because these approaches have been targeted at deployed software systems, excessive runtime overhead is generally undesirable. Therefore it is important to limit instrumentation overhead as much as possible while still supporting the highest levels of classification accuracy. In general previous efforts have tended to either ignore this problem or have appealed to various sampling strategies (see Section 2) for a solution. One potential drawback of sampling however is that aggressive sampling schemes greatly increase the numbers of observations that must be made in order to have confidence in the data.

While we believe that sampling can be a powerful tool, we also conjecture that large cost reductions may derive from reducing the cost of the measurement instruments themselves. To evaluate this conjecture, we have designed and evaluated an improved approach in which most of the data

FSE '10

collection work is performed by fast hardware performance counters. The data is augmented with further data collected by a minimal amount of software instrumentation that is added to the system's software. We contrast this approach with other approaches implemented purely in hardware or purely in software. Our empirical evaluation, conducted on four open source projects, suggests that for the systems used, our hybrid hardware and software instrumentation approach was as good or better than other approaches in distinguishing failed executions from successful executions; and it did so at a fraction of the cost of using purely software-based instrumentation.

2. RELATED WORK

Classifying Program Executions. Several researchers have studied how to predict program execution behaviors from program spectra. To our knowledge all of these efforts used software spectra (program spectra collected purely by software instrumentation) as their execution data. Podgurski and colleagues [7, 8, 14, 19] present a set of techniques for clustering program executions. For example, in one effort they used cluster analysis to group program executions in deployed instances. They showed that the individual clusters tended to include execution profiles stemming from the same failures. Bowring, Rehg, and Harrold [3] used Markov models to classify program spectra as coming from either successful or failed executions. Their approach uses software instrumentation and requires probes at every branch in a system. Brun and Ernst [4] used two machine learning approaches to identify observed likely program invariants that tended to be present during faulty program runs. This approach computes very detailed abstractions at many program points and is thus, quite expensive. Jones and Harrold and Staska [13] use statement coverage information to identify likely causes of failures. Chen et al. [6] track the software components used in computations and leverage that data for problem determination. We provide a comparison of the proposed approach to similar types of program spectra used by Jones and Chen in the experiments section.

Reducing Data Collection Overhead. Other research has extended earlier work with some consideration for limiting instrumentation overhead. Liblit and colleagues [16, 17] transform the source of software systems to efficiently sample execution data collected from users and use this data to perform fault localization. The rewriting process, however, significantly increases the size of the system code and imposes a significant memory overhead. In addition, the noise created by sampling must be balanced by instrumenting very large numbers of deployed instances, which may not be possible for some applications. Haran et al. [11] developed several techniques for classifying execution data as belonging to one of several classes. Each technique works by instrumenting a set of deployed instances. Each instance in this set, however, captures only a small subset of the program spectra and that subset is different from that captured by all other instances. The goal is to create a combined data set that captures enough information to allow accurate classifications. Yilmaz et al. present a related approach called Time Will Tell (TWT) [24]. In this work they used minimal amounts of software instrumentation to capture time spectra (i.e., traces of method execution times). Their empirical evaluations suggested that coarse-grained execution data could effectively capture patterns in executions.

Hardware-Based Profiling. Another way to cut instrumentation overhead is to do all data collection at the hardware level. Anderson et al. [2] present a lightweight profiling technique done entirely with hardware performance counters. While the program is running, they randomly interrupt it and capture the value of the program counter. Using this information they statistically estimate the percentage of time each instruction is executed. Assuming the program runs long enough, they can generate a reasonably accurate model with overheads of less than 5%. An open question with this work is whether it can be extended to other types of data spectra and analysis purposes. One reason for the uncertainty is that their approach has no way to associate execution data back to program entities.

In general, it is almost always possible to collect quite detailed data from program executions, which is the case with some of the existing approaches discussed above, to distinguish failed executions from successful executions. However, the runtime overhead required to collect such data often makes it impractical. In this work, we aim at finding a sweet spot between the accuracy of classifications and runtime overheads.

3. COMBINING HARDWARE AND SOFTWARE INSTRUMENTATION

In this article we design and evaluate an improved approach to classifying program executions that is both effective and inexpensive. Specifically, we use hardware counters to collect raw data, but we also use lightweight software instrumentation to associate subsets of the hardware-collected data with specific program entities.

Hardware performance counters are hardware-resident counters that record various events occurring on a processor. Today's general-purpose CPUs include a fair number of such counters, which are capable of recording events, such as the number of instructions executed, the number of branches taken, the number of cache hits and misses experienced, etc. To activate these counters, programs issue instructions indicating the type of event to be counted and the physical counter to be used. Once activated, hardware counters count events of interest and store the counts in a set of special purpose registers. These registers can also be read and reset programmatically at runtime.

One challenge we encountered when first using hardware performance counters to collect program execution data was that the counters do not distinguish between the instructions issued by different processes. To deal with this we used a kernel driver, called `perfctr` (`linux.softpedia.com`), which implements virtual hardware counters that can track hardware events on a per-process basis.

A second challenge was that hardware performance counters have limited visibility into the programs being executed, e.g., by themselves they do not know, for example, to which program function the current instruction belongs. As described in more detail in Section 5.1, raw hardware spectra are generally too coarse to be useful in classifying executions. To improve this situation, we chose to associate hardware spectra with function invocations. That is, we use traditional software instrumentation to indicate which function is currently executing so that different subsets of the hardware spectra are properly associated with that function. While we opted to track program execution data at

the function level, the techniques are equally applicable to other granularity levels.

Thus, at a high level our approach in its simplest form works as follows: 1) a hardware counter of interest is activated at the beginning of a program execution, 2) the value of the counter is read before and after each function invocation and the difference is attributed to the invocation, and 3) the counter is deactivated at the end of the execution. Note that hardware counters are always active during executions.

Early experimentation with a prototype implementation uncovered further issues. First, one key cost driver is the use of software instrumentation that causes (very expensive) context switching. In this work we mainly use one instruction, called `rdmpc`, which enables us to read the values of physical counters from user space without causing an OS context switch. For example, to compute the value of a virtual counter, we record two values (`sum` and `start`) at each instrumentation site for later processing, in addition to the current value of the physical counter (`now`). The `sum` variable stores the total number of events observed in the current process from the start up to the last process switch involving the process. The `start` variable records the value of the physical counter at the time of the last process switch. Once these values are known, the value of a virtual counter is computed as $now - start + sum$. The `perfctr` driver used in this work augments the OS-level context switch code to store the `start` and `sum` values at suspends and restore them at resumes. Furthermore, this driver provides a view of these values in user space. Consequently, reading the value of a virtual counter can be done fast without requiring any further expensive OS context switching. To give an example, based on performing a million counter read operations, we observe that it takes 45 clock cycles on average to read the value of a virtual counter on our test platforms.

Even if the cost of reading counters is low, another cost driver is the raw number of times the counter values need to be read. Therefore, to further reduce overhead, we in this work chose not to profile functions that are invoked more than 50 times, nor those that invoke more than 50 other functions. Note that the hardware counters are still active during the execution of unprofiled functions, the hardware-spectra is just associated with function that calls these unprofiled functions. The cutoff value was chosen so that functions that account for the vast majority of invocations were filtered. The information required for this purpose was obtained by analyzing the histograms of function invocation counts observed in our experiments.

Furthermore we chose to ignore functions that handle already detected failures, e.g., `error(...)` and `fatal(...)`. We do this because these function invocations are the effect of failures and thus not useful in understanding failure causes (which is often the scenario for which failure classification is done). In the rest of the paper these filtering steps are referred to as global function filtering.

3.1 A Simple Feasibility Study

Combining hardware and software instrumentation clearly adds some additional context to hardware spectra. Nevertheless, the underlying hardware spectra, which are based on commonly available hardware performance counters, are inherently less flexible than programmable software spectra. As a result, it is unclear whether combined spectra provide data that can be used to accurately classify executions.

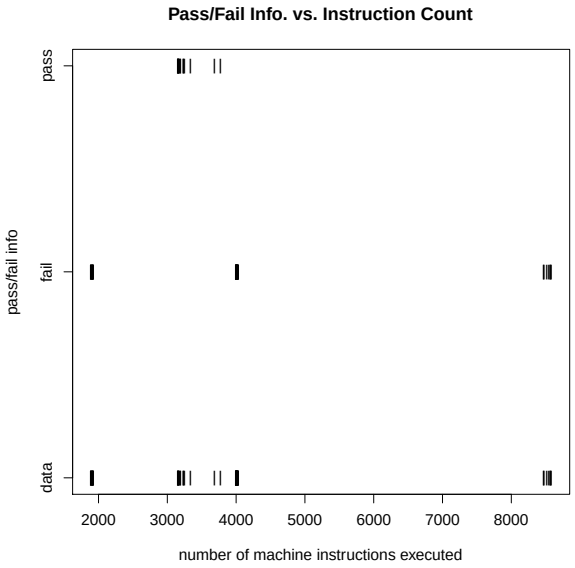


Figure 1: Classification of the number of instructions executed in the `socket` function.

As a simple test, we therefore conducted a rudimentary 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 parameters that this function takes as input: `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.

For this system we created 66 test cases that exhaustively exercised all combinations of the parameters settings. For each input combination, we measured the number of machine instructions executed during the invocation of the function. Since not all input combinations were supported on our test platform, 51 of the test cases failed. Figure 1 depicts a simple hardware spectra we obtained. The horizontal axis depicts the number of machine instructions executed and the vertical axis denotes the entire data as well as the data obtained from the failed and successful executions.

Using this data we asked whether this simple hardware spectra could be used to classify executions in which the requested socket failed to be created? As can be seen from the figure, even using only one simple hardware counter the difference between the successful and failed invocations is immediately apparent; failed invocations execute either many fewer or many more machine instructions compared to those of successful invocations. Training a classification model for our observations (using the Weka's J48 algorithm) and testing it on previously unseen executions provided an F-measure of 0.98 in predicting the failed invocations. With manual analysis we determined that the failed invocations that executed few 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. The failed invocations that executed more instructions, on the other hand, did so because their

subject application	LOC	number of functions	number of versions	number of defects	total tests	passing tests	failing tests
grep	10068	146	4	5	3050	2346	704
flex	10459	162	4	19	11417	8399	3018
sed	14427	255	5	15	5455	4454	1001
gcc	222196	2214	1	unknown	7701	7494	207

Table 1: Subject applications used in the experiments.

failures caused the kernel to immediately release all previously allocated resources. Successful executions, in contrast, hold on to resources until the socket is explicitly closed.

Although this simple demonstration is by no means conclusive, it encouraged our belief that program spectra gathered from hardware performance counters can capture useful data that can be used to reliably detect failures.

4. EXPERIMENTS

To evaluate the accuracy and effectiveness of hardware-based spectra in distinguishing failed executions from successful executions, we conducted a set of experiments using four open source real-life applications as our subject applications. This section reports on our experiments.

4.1 Independent Variables

We manipulated two independent variables in our experiments.

Program spectrum type. This variable has six different levels. The first three program spectra are collected using hardware performance counters:

- **TOT_INS** counts the number of machine instructions executed
- **BRN_TKN** counts the number of branches taken
- **LST_INS** counts the number of load and store memory instructions executed

The three remaining spectra are gathered using traditional software profiling:

- **CALL_SWT** records the functions invoked
- **STMT_FREQ** counts the number of times the source code statements are executed
- **TIME** measures execution times of functions at the level of nanoseconds

Although numerous types of software-based spectra have been proposed in the literature, we chose to use these four types for several reasons. First, each has been successfully used in the literature for related purposes [24, 13, 21, 6]. Second, each is representative of a variety of related spectra in terms of the runtime overhead they impose. For example, the overheads of collecting the CALL_SWT and STMT_FREQ spectra can be viewed as lower bounds on collecting simple function-level, and detailed statement-level execution data, respectively. Similarly, the overhead of collecting the TIME spectra can be viewed as a lower bound on collecting simple execution data that requires OS context switches; we used system calls to compute the execution times in this work.

Dynamic call tree depth to be profiled. We used six different dynamic call tree depths (0, 1, 2, 3, 4, and *all*) as a cutoff point for our instrumentation. For a depth d, only the function calls that happen when the call stack depth is less than or equal to d are profiled. When $d = 0$, only the `main` function is profiled, and when $d = all$, all the function invocations of interests are profiled. Note that hardware counters are always active during an execution. Hardware spectra data for function invocations occurring at depth $> d$ are simply associated with the function invocation at depth d.

4.2 Subject Applications

We used four open source subject applications in our experiments: `grep`, `flex`, `sed`, and `gcc`. These widely-used applications print lines matching a pattern, generate fast lexical analyzers, filter and transform text as a stream editor, and compile C programs, respectively. Table 1 provides some descriptive statistics for the subject applications.

The first three subject applications were taken from the Software-artifact Infrastructure Repository (SIR) [9]. The SIR repository provided us with several versions of these subject applications with known defects. The `gcc` application used in the experiments was the official release of the GNU Compiler Collection version 2.95.2. Each subject application came with its own test suites and oracles, which we used in the experiments.

4.3 Operational Model

To conduct our studies we executed the test suites of our subject applications, collected the spectra from these executions. For each run we measured the overhead incurred by our data collection instrumentation. Next we determined whether each execution was successful or failed using the test oracles that came with the suites. After that, we created classification models for each combination of subject program version, spectrum type, and depth of dynamic call tree being profiled. Once the data was collected, we created and evaluated the classification models by using the Weka's J48 classification algorithm with five-fold stratified cross validation [10]. That is, the evaluation of the models was performed on previously unseen executions.

To evaluate the runtime overhead induced by collecting various types of spectra, we compare the execution times of programs with and without profiling and compute the overhead as follows:

$$overhead = \frac{exec.\,time\,w/\,prof. - exec.\,time\,w/o\,prof.}{exec.\,time\,w/o\,prof.}$$

To evaluate the accuracy of classification models we use several standard metrics. Precision (P) and recall (R) are

two widely used metrics to assess the performance of classification models. For a given testing table T, we define them as follows:

$$recall = \frac{\#\,of\ correctly\ predicted\ failed\ executions\ in\ T}{total\ \#\,of\ failed\ executions\ in\ T}$$

$$precision = \frac{\#\,of\ correctly\ predicted\ failed\ executions\ in\ T}{total\ \#\,of\ predicted\ failed\ executions\ in\ T}$$

Since neither measure predominates our evaluation, we combine these measures using the F - measure. This is defined as:

$$F - measure = \frac{(b^2+1)PR}{b^2P + R}$$

Here b controls the weight of importance to be given to precision and recall: $F = P$ when $b = 0$ and $F = R$ when $b = \infty$. Throughout the paper, we compute F-measure with $b = 1$, which gives precision and recall equal importance, and use it to evaluate the classification models.

4.4 Experimental Setup

The software instrumentation used in associating hardware spectra with function invocations was implemented via binary instrumentation using GNU's C extension framework for instrumentation (i.e., `gcc`'s `-finstrument-functions` option). The same approach was also used to collect the TIME and CALL_SWT spectra. The STMT_FREQ spectra on the other hand were collected using GNU's `gcov` test coverage tool. We modified the implementation of this tool to support selective instrumentation so that coverage information was computed only for the functions of interests.

To compute runtime overheads, we dedicated one CPU of a dual-CPU system to execute the test cases (using `sched_setaffinity` and other related system calls). No other user-level programs were allowed to run on the dedicated CPU. We furthermore measured execution times at the level of nanoseconds in terms of the CPU allocation time using the K-best measurement scheme [5]. All the experiments were performed on a dual Intel processor machine with 2GB of RAM, running the CentOS 5.2 operating system.

All told, our data set is comprised of 19,922 total test cases run across 39 defective versions of our subject applications (Table 1). These 39 defective versions were culled from an set of 166 defective versions using the criteria that the ratio of failed to successful executions was between 0.05 and 1.50. We did this because classification techniques themselves either perform poorly or need special enhancement when one class is much more common than the other. Since our goal is not to evaluate classification techniques themselves we ignore these cases for our analysis. For our evaluations, we created a total of 5,850 classification models (39 versions `x` 6 spectra types `x` 5 call tree depths `x` 5-fold cross validation).

5. DATA AND ANALYSIS

The following sections present and discuss some of our results.

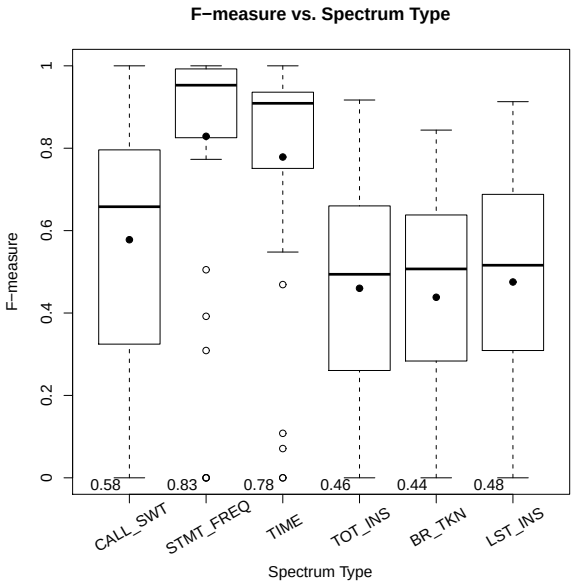


Figure 2: F-measure vs. spectrum type (hardware and software spectra).

5.1 Hardware and Software Spectra

Our first study examined the accuracies and overheads provided by hardware and software spectra. To collect the hardware spectra, we activated one hardware counter of interest, read the value of the counter at the beginning and at the end of an execution, and associated the difference with the execution. Note that hardware counters were read only twice for each program execution in this study. This scenario corresponds to profiling with $d = 0$. To collect the software spectra, we first enabled our global function filters (Section 3) and then profiled all the remaining functions to collect one type of program spectra. Since this process ignores the depth of the call stack, it corresponds to profiling with $d = all$. For these subject programs and test cases, global function filtering prevented 98% of the invocations on average from being profiled.

The last three columns of Table 2 present the runtime overheads incurred by each of the 3 hardware spectra used in our experiments. For this study the rows marked as $d = 0$ are relevant. As the table indicates, the overheads ranged from 0.0031 to about 0.0096. That is, the runtime overheads attributable solely to hardware performance counters were under 1%. The first three columns of Table 2 present the overheads incurred by each of the 3 software spectra used in the experiments. The relevant data are in the rows where $d = all$. The CALL_SWT spectrum, which simply records the list of called functions, incurred overheads of about 1% to 2%. The two remaining software counters incurred higher overheads, ranging from about 7.6% to about 27%.

Figure 2 plots F-measures of classification accuracy we obtained for this study. Each box in this plot illustrates the distribution of F-measures obtained from a spectrum type. The lower and the upper end of a box represents the first and the third quartiles and the horizontal bar inside represents the median value. Numbers given below the boxes indicate the mean values and are visualized as filled circles.

sut	CALL_SWT		STMT_FREQ		TIME		TOT_INS		BR_TKN		LST_INS	
	d	overhead	d	overhead	d	overhead	d	overhead	d	overhead	d	overhead
grep	0	n/a	0	n/a	0	n/a	0	0.0031	0	0.0036	0	0.0037
	1	0.0060	1	0.0656	1	0.0425	1	0.0086	1	0.0089	1	0.0091
	2	0.0072	2	0.0673	2	0.0673	2	0.0098	2	0.0107	2	0.0102
	3	0.0075	3	0.0686	3	0.0650	3	0.0101	3	0.0104	3	0.0109
	4	0.0076	4	0.0742	4	0.0750	4	0.0120	4	0.0114	4	0.0112
	all	0.0112	all	0.0763	all	0.0821	all	0.0125	all	0.0125	all	0.0132
flex	0	n/a	0	n/a	0	n/a	0	0.0060	0	0.0058	0	0.0060
	1	0.0102	1	0.0792	1	0.0847	1	0.0157	1	0.0154	1	0.0161
	2	0.0105	2	0.1184	2	0.1702	2	0.0156	2	0.0166	2	0.0163
	3	0.0130	3	0.1245	3	0.2170	3	0.0165	3	0.0162	3	0.0169
	4	0.0130	4	0.1263	4	0.2199	4	0.0168	4	0.0162	4	0.0168
	all	0.0152	all	0.1303	all	0.2448	all	0.0169	all	0.0168	all	0.0167
sed	0	n/a	0	n/a	0	n/a	0	0.0075	0	0.0096	0	0.0083
	1	0.0182	1	0.2441	1	0.2018	1	0.0142	1	0.0140	1	0.0141
	2	0.0184	2	0.2597	2	0.2268	2	0.0150	2	0.0151	2	0.0151
	3	0.0182	3	0.2551	3	0.2233	3	0.0146	3	0.0143	3	0.0141
	4	0.0188	4	0.2603	4	0.2253	4	0.0144	4	0.0149	4	0.0147
	all	0.0192	all	0.2701	all	0.2278	all	0.0152	all	0.0152	all	0.0161

Table 2: Runtime overheads.

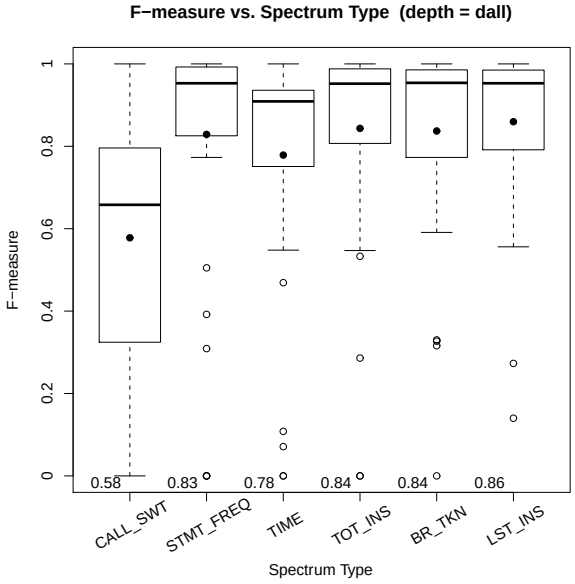


Figure 3: F-measure vs. spectrum type (depth=all).

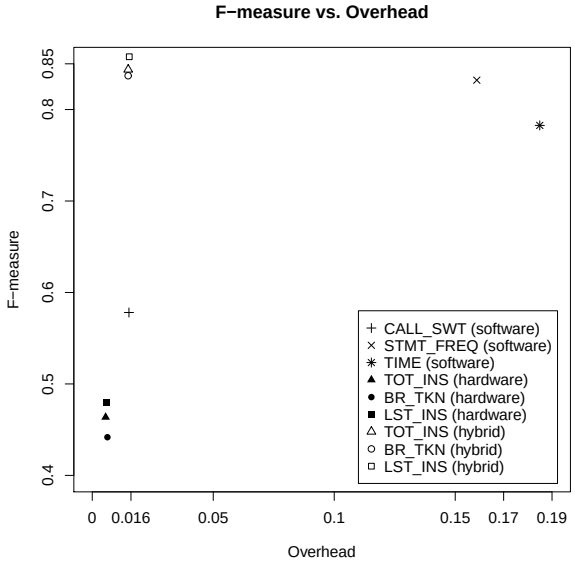


Figure 4: F-measure vs. overhead.

The Figure shows that even though hardware spectra were inexpensive, they were not very effective in classifying executions. In fact, the classification models created using the hardware spectra collected in this study predicted failed executions with an average F-measure of only 0.46. Note: the expected value of the F-measure varies depending on the ration of successful to failing executions, but for our studies random assignment would give F-measures between about 0.5 and about 0.30. The Figure also shows that the F-measures for software spectra were generally higher, ranging from 0.58 for CALL_SWT to 0.83 (STMT_FREQ) and 0.78 (TIME).

Overall we see that the hardware spectra were very inexpensive, but inaccurate. For the software spectra, one, CALL_SWT was also very inexpensive and had moderate accuracy. The remaining 2 software spectra were more accurate, but incurred substantially more overhead.

5.2 Associating Data with Function Invocations

To improve classification accuracy while limiting instrumentation overhead, we decided to combine hardware and software instrumentation. The general idea is to collect most of the data using hardware counters, but to use software instrumentation to assign that data to specific program entities.

To evaluate this idea, we made several changes to our procedures and then repeated the previous study, The first change was that we used software instrumentation to asso-

ciate hardware-collected data with individual function invocations. Specifically, we inserted code into the programs that read the value of the appropriate hardware counter at the beginning and at the end of each invocation. We then attributed the difference to that invocation. Since this process ignores the depth of the call stack, it corresponds to profiling with $d = all$.

Table 2 presents the runtime overhead incurred by each type of performance counter. The data relevant for this study appear in rows marked depth=all. Here we see that the overhead incurred by our combined hardware and software approach ranged from about 1.25% for grep to about 1.5% for sed. These levels of overhead indicate that even minimal software instrumentation increased overhead by a factor of 2–4, but that the total overhead was nonetheless substantially smaller than that incurred with more sophisticated software instrumentation approaches.

Figure 3 depicts the F-measures of classification accuracy we obtained for this study. Our first observation is that associating hardware counters with function invocations significantly boosted classification accuracy over plain hardware spectra. Here the average F-measure increased from 0.46 to 0.85.

Figure 4 provides an integrated view of the costs and benefits of using hardware, software and hybrid spectra. One observation is that for comparable overhead levels, the hybrid spectra were significantly better at detecting the failures than the hardware or software spectra used in the study. For example, compared to the CALL_SWT spectrum, where the average overheads were almost the same (0.015), hybrid spectra provided significantly better F-measures; an average F-measure of 0.85 *vs.* 0.58, respectively.

Alternatively, for comparable accuracy levels, the hybrid spectra induced significantly less overhead (see also the rows where $d = all$ in Table 2). For example, the hardware spectra and the STMT_FREQ spectra have similar F-measures; 0.85 *vs.* 0.83 on average, respectively. However, the hybrid spectra incurred a fraction of the overhead incurred by the STMT_FREQ spectrum. The runtime overhead was 0.015 on average for the hybrid spectra and 0.159 for the STMT_FREQ spectrum. This corresponds to about 10-fold reduction in the runtime overhead.

5.3 Using Structural Sampling

As discussed in Section 2 several previous research efforts have use sampling strategies to further reduce profiling overhead. Many of these strategies are implemented by profiling only a carefully chosen, small subset of all the potential measurement locations. We call this technique structural sampling to distinguish it from statistical sampling, where, logically, all measurement locations are instrumented but the instruments are enabled with certain probabilities. Note however that even statistical approaches are sometimes implemented via structural sampling. Specifically, code sections are duplicated and copied into both instrumented and uninstrumented paths [16]. While sampling certainly reduces overhead it may have a negative effect on classification accuracy, especially when the number of instrumented systems is low and/or when failure frequencies are low.

In this study, we evaluated the overhead and accuracy of classifying executions when execution data is acquired via a structural sampling approach. To carry out the study, we used the following simple sampling approach. We only profile function invocations when they occur at or below depth d in the dynamic call stack, where $1 \leq d \leq 4$. Note however that our hardware counters were always activate during executions. If a function invocation occurs at depth greater than d, the resulting data is not associated with the current invocation, but is associated with the calling function at depth d in the dynamic call stack.

Furthermore, to focus on areas in which the loss of precision might cause a resulting loss of classification accuracy, we considered only those faults that were exercised at a depth greater than d. These defects were identified by manually marking the depths at which the underlying faults were executed and then choosing those that satisfied the condition. All other faults were ignored.

In this study, we did not observe significant overhead reduction by employing our structural sampling approach (Table 2). This was because after enabling our global function filters the number of functions profiled at different depths were close to each others.

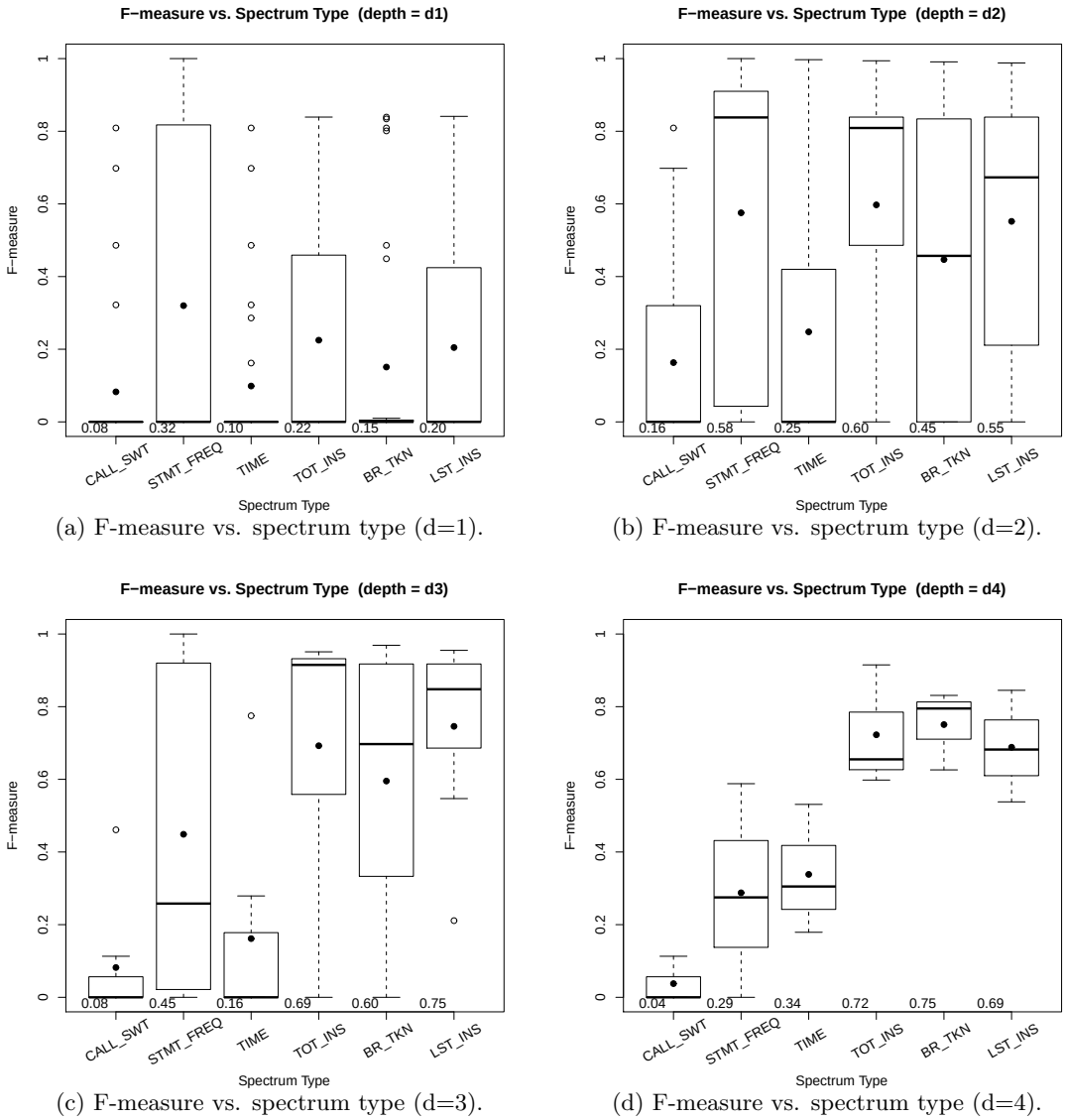
Figure 5 depicts the classification accuracies observed in this study. As the figure indicates, the hybrid spectra were indistinguishable from software spectra when d was low ($d = 1$ and $d = 2$). As depth increased the hybrid spectra generally displayed greater accuracy. Referring back to Figure 3, however, we note that when $d = all$, software spectra were as accurate as hybrid spectra. Referring to Table 2, we also note that hybrid spectra incurred a fraction of the overhead cost compared to that of the software spectra. For example, when $d = 4$, the average F-measure is 0.72 with an overhead of 0.015 for the combined spectra, and 0.29 with an overhead of 0.154 for the STMT_FREQ spectrum.

To better understand these results we manually analyzed the resulting classification models. One reason for which the classification models created from hybrid spectra outperform those created from software spectra is that the hybrid spectra summarize rather than ignore unprofiled functions. That is, when using hardware performance counters we associate the execution data for functions that are called at a level greater than d with the parent function at depth d. In a sense, hardware counters were able to collect some information from the uninstrumented parts of the programs. Visual inspection of the classification models confirmed that the calling functions at level d become the key indicators of failures. On the other hand, the software spectra, such as the STMT_FREQ spectra, suffered greatly, because they completely ignore uninstrumented parts of the code.

5.4 Different Defect Types

While studying the classification models obtained in the previous study, we observed several cases in which software spectra outperformed our hybrid spectra and vice versa. For example, the STMT_FREQ spectra when $d = 1$ performed better than the hybrid spectra. We discovered that our subject applications often perform certain types of error checking at depth one and two, such as checking to see if the return values of function calls indicate an internal failure. Of course, if one of these checks fails, it is a perfect indicator of a failure. In these cases STMT_FREQ spectra were faultless in predicting failures. However, many failures are not internally detected by our subject applications. In these cases software spectra tended to perform poorly (e.g., when $d = 4$).

To further investigate potential relationships between ac-



(a) F-measure vs. spectrum type (d=1).

(b) F-measure vs. spectrum type (d=2).

(c) F-measure vs. spectrum type (d=3).

(d) F-measure vs. spectrum type (d=4).

Figure 5: Failure prediction F-measure vs. spectrum type (depth = {1, 2, 3, 4}).

curacy and failure characteristics, we performed the following additional study. Working with three graduate students at Sabanci University, we asked them to examine each known defect in our subject programs and categorize them into 1 of 3 categories. These categories are laid out by the SIR repository that categorizes each defect into one of the three broad defect types: Type 1, Type 2, and Type 3. Type 1 defects cover the defects that stem from the errors made on variable assignments, such as using erroneous values for variables, and passing incorrect parameters to functions. Type 2 defects represent the errors made on the control flow of programs, such as missing a function call, and using incorrect branching conditions. Type 3 defects on the other hand depict the errors made on memory operations, such as pointers-related errors.

We then used a voting scheme to resolve conflicting categorizations. Further inconsistencies were handled on a case-by-case basis. Out of 39 defects used in the experiments, 19 defects were categorized into Type 1, another 19 into Type 2, and only 1 defect into Type 3. Since we happened to have only one Type 3 defect, we leave it out of discussion in this section.

Spectrum	Type 1	Type 2
CALL_SWT	0.57	0.57
STMT_FREQ	0.69	0.96
TIME	0.64	0.91
TOT_INS	0.72	0.96
BR_TKN	0.71	0.95
LST_INS	0.77	0.95

Table 4: Average F-measures obtained on different types of defects.

depth	CALL_SWT		STMT_FREQ		TIME		TOT_INS	
	F-measure	overhead	F-measure	overhead	F-measure	overhead	F-measure	overhead
1	0	0.0009	0	1.1291	0	0.5304	0	0.0052
2	0	0.0007	0.44	1.1801	0	0.5343	0	0.0049
3	0	0.0076	0.44	1.1891	0	0.5310	0.45	0.0047
4	0.05	0.0131	0.56	1.1901	0.02	0.5561	0.79	0.0115
all	0.59	0.0136	0.82	1.2424	0.59	0.5766	0.81	0.0138

Table 3: Results obtained from the `gcc` experiments.

Table 4 depicts the average F-measures obtained for different types of defects. Note that the F-measures used in this table are adopted from Section 5.2. We observed that the hybrid spectra were generally more accurate than the software spectra across both defect types. That is, different types of defects did not cause any noticeable differences between the relative accuracies of the hybrid and software spectra.

Furthermore, each spectrum type was better at classifying Type 2 failures than Type 1 failures. We observed that Type 2 defects, by directly changing the control flow of programs, often caused pronounced differences in the paths taken by the failed and successful executions. These differences reflected on the STMT_FREQ spectrum, for example, as a set of suspicious source code statements executed and on the combined spectra as a set of suspicious amount of computational activities performed. It turned out that it was much easier for the spectrum types used in the experiments (except for CALL_SWT) to detect such differences compared to the ones caused by Type 1 defects.

5.5 Replication with gcc

The studies discussed so far involved medium-sized applications. In this study, we evaluated our approach on larger GNU Compiler Collection (`gcc`) version 2.95.2. For this study we collected the CALL_SWT, STMT_FREQ, TIME, and TOT_INS spectra at various levels of granularity. We then used this data to classify failing and successful executions. As we do not know the actual causes of the failures for `gcc`, we could not manually verify the specifics of the resulting classification models as we did in our previous studies.

In these experiments, we only profiled the `cc1` component of the compiler. This is the core component of the `gcc` compiler and is responsible for compiling source code into assembly language. We executed a total of 7701 test cases out of which 207 of them failed and 7494 of them passed. As with the rest of the experiments we first applied our global function filters (Section 3). This prevented profiling around 99% of the total function invocations. As with our previous studies, we also did not profile functions that are called only after an internal failure has been detected.

Table 3 depicts the overheads and F-measures observed in this study. In general these results are consistent with those obtained on the smaller-scale applications used in our previous studies. Looking at the overhead data, we observe that while the cost of profiling for the TOT_INS spectrum in this study was similar to the ones obtained in other studies, the STMT_FREQ spectrum imposed significantly higher costs. A manual investigation revealed that one reason for the variation was that the `gcc` application often executed more statements per function invocation compared to the rest of the subject applications.

Compared to the STMT_FREQ spectrum, when $d = all$, the TOT_INS spectrum provided similar F-measures at a fraction of the cost; an average F-measure of 0.81 with an overhead cost of 0.014 for the TOT_INS spectrum and an average F-measure of 0.82 with an overhead cost of 1.24 for the STMT_FREQ spectrum. Alternatively, compared to the CALL_SWT spectrum, where the overhead costs are similar, the TOT_INS spectrum provided significantly better F-measures; an F-measure of 0.81 vs. 0.59. Furthermore, the TOT_INS spectrum was better at predicting the failures at almost all granularity levels of profiling compared to the software spectra.

6. CONCLUDING DISCUSSION

Several research efforts have studied how to infer properties of executing software systems from program spectra. These efforts appear to produce accurate results [24, 13, 23, 22, 12, 18, 15], but the issue of costs is less well understood. In this article we examined a novel approach for instrumenting software systems to support execution classification. This approach combines low overhead, but coarse-grained hardware spectra with minimal amounts of higher overhead software instrumentation to create a hybrid spectra.

We also evaluated this approach and compared it against other representative approaches. In the experiments we used three different types of hybrid spectra and three different types of software spectra. We compared the cost and accuracy of the hybrid spectra to those of the software spectra.

Before discussing the results of these evaluations, we remind the reader that 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 four data points. A related threat concerns the representativeness of the defects used in the experiments. Although, the `grep`, `flex`, and `sed` applications were taken from an independent defect repository which has been leveraged by many related studies in the literature [1, 13, 24, 20, 21] and the `gcc` application was an official release of a frequently-used compiler, they only represent a subset of defects.

Keeping these limitations in mind, we believe that our study supports our basic hypothesis: that our hybrid spectra approach incurs low overhead, but still produces data that generates accurate classification models.

We arrived at this conclusion by applying our approach to several medium-sized subject systems. Based on this we first observed that for our subject systems, test cases and choices of program spectra, hardware-only spectra had low

overhead, but led to inaccurate classification models, while software-only spectra had high overhead, but led to accurate classification models.

Next, we compared the hybrid spectra to hardware-only and to software-only spectra. Compared to hardware-only spectra, our hybrid spectra, however, added little additional overhead, while greatly increasing classification accuracy. Compared to software spectra, our hybrid approach, had equivalent accuracy, but greatly reduced overhead.

We also observed that limiting profiling according to depth of the call stack did not greatly affect the runtime overhead of collecting the various program spectra. This was primarily due to the fact that we had already applied global function filtering, which already reduced the number of function invocations profiled by around 98%. The accuracy of the hybrid spectra appeared to degrade more slowly than did that of the other spectra, but more studies are needed.

After this, we observed that all the program spectra-based approaches we studied were better at classifying failed executions stemming from failures that changed control flow, rather than those that caused by incorrect data usage. From this perspective the hybrid spectra was no different from the hardware-only or software-only spectra.

Finally, we replicated most of our initial studies on a larger system, `gcc`. This study was consistent with our previous findings. This further supports our hypothesis that hybrid spectra can be both lightweight and yield accurate execution classifications.

We believe that this line of research is novel and interesting. We are therefore continuing to investigate how hybrid hardware- and software-based instrumentation can serve as abstraction mechanisms for program executions in various software quality assurance approaches, such as fault localization, failure prediction, security assurance, and in-the-field quality assurance approaches.

7. ACKNOWLEDGMENTS

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

8. 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. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proc. of the Int'l Symp. on Software Testing and Analysis (ISSTA 2004)*, pages 195–205, July 2004.

[4] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of the 26th Int'l Conf. on SW Eng. (ICSE 2004)*, pages 480–490, May 2004.

[5] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2002.

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

[7] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proc. of the 9th ACM SIGSOFT international symposium on Foundations of SW Eng.*, pages 246–255, September 2001.

[8] W. Dickinson, D. Leon, and A. Podgursky. Finding failures by cluster analysis of execution profiles. In *Proc. of the 23rd Int'l Conf. on SW Eng. (ICSE 2001)*, pages 339–348, May 2001.

[9] 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.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1):10–19, 2009.

[11] 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.

[12] G. Hoglund and G. McGraw. *Exploiting software: How to break code*. Addison-Wesley Publishing Company.

[13] 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.

[14] D. Leon, A. Podgurski, and L. J. White. Multivariate visualization in observation-based testing. In *Proc. of the 22nd international conference on SW engineering (ICSE 2000)*, pages 116–125, May 2000.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 141–154, June 2003.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2005)*, June 2005.

[18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *ICSE Conference Proceedings*, pages 465–475, 2003.

[19] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang. Automated support for classifying sw failure reports. In *Proc. of the 25th Int'l Conf. on SW Eng. (ICSE 2003)*, pages 465–474, May 2003.

[20] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ASE Conference Proceedings*.

[21] 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.

[22] S. Singer, K. Gross, J. Herzog, S. Wegerich, and W. King. Model-based nuclear power plant monitoring and fault detection: theoretical foundations. In *Proceedings of the International Conference on Intelligent Systems Applications to Power Systems*, pages 60–65, 1997.

[23] R. Vilalta and S. Ma. Predicting rare events in temporal domains. In *ICDM Proceedings*, pages 474–481, 2002.

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