Seer: A Lightweight Online Failure Prediction Approach

Burcu Ozcelik and Cemal Yilmaz

Abstract—Online failure prediction approaches aim to predict the manifestation of failures at runtime before the failures actually occur. Existing approaches generally refrain themselves from collecting internal execution data, which can further improve the prediction quality. One reason behind this general trend is the runtime overhead incurred by the measurement instruments that collect the data. Since these approaches are targeted at deployed software systems, excessive runtime overhead is generally undesirable. In this work we conjecture that large cost reductions in collecting internal execution data for online failure prediction may derive from pushing the substantial parts of the data collection work onto the hardware. To test this hypothesis, we present a lightweight online failure prediction approach, called *Seer*, in which most of the data collection work is performed by fast hardware performance counters. The hardware-collected data is augmented with further data collected by a minimal amount of software instrumentation that is added to the systems software. In our empirical evaluations conducted on three open source projects, Seer performed significantly better than other related approaches in predicting the manifestation of failures.

Index Terms—Online failure prediction, hardware performance counters, software quality assurance, software reliability

1 Introduction

Software system do fail in the field [1], [2]. By following this pragmatic line of thought, many *online failure prediction* approaches have been developed to predict the manifestation of failures at runtime, i.e., while the system is running and before the failures occur, so that preventive measures, such as system reboots, or protective measures, such as checkpointing, can be proactively taken to improve software reliability [3].

At a high level, online failure prediction approaches operate in a similar manner. The system under observation is augmented with failure prediction models. As the augmented system runs, specific types of execution data, called *system spectra*, are collected and fed to the models. The models then make predictions at runtime about whether the execution will fail or not.

Prediction models are often trained by using historical executions. In particular, these models attempt to capture patterns that are correlated with the expected behavior of the system (e.g., as observed in successful executions) and/or correlated with the manifestation of failures (e.g., as observed in failed executions). A fundamental assumption of 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, strongly support this assumption, successfully applying a variety of system spectra for online failure prediction [4]–[31].

Many online failure prediction approaches treat the system under observation as a black box and collect specific types of execution data that are either directly reported by the system, such as failure and error logs [4]–[16], or directly observable from outside the system, such as CPU and memory utilization of the system [17]–[31]. Although these approaches have been shown to be effective in predicting failures, we believe that the quality of predictions can further be improved by treating the system under observation as a white box and collecting internal execution data, i.e., by collecting data from inside executions. For example, not all failure-inducing errors may leave externally detectable traces, which can reduce the prediction accuracy of black-box approaches. Even if some traces are present, due to the often noisy nature of external measurements, it may take time for these traces to become externally detectable, which can cause black-box approaches to issue late warnings for failures, rather than early ones. Therefore, being close to the sources of failures by collecting and analyzing internal execution data, can improve the quality of failure predictions.

Collecting internal execution data, however, requires to instrument the system under observation; the data is collected every time the instrumentation code is executed. One reason as to why existing approaches generally refrain themselves from collecting internal execution data is the runtime overhead incurred by the collection process, i.e., the runtime overhead of executing the instrumentation code. Since

- *B. Ozcelik was with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey, at the time of the work. E-mail: burcuoz@sabanciuniv.edu*
- *C. Yilmaz is with the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. E-mail: see http://people.sabanciuniv.edu/cyilmaz*

these approaches are targeted at deployed software systems, excessive runtime overhead is generally undesirable. Therefore, if internal execution data is to be collected for online failure prediction, it must be done in a way that limits the runtime overhead as much as possible, while still supporting the highest levels of prediction quality.

In this work we conjecture that large cost reductions in collecting internal execution data for online failure prediction may derive from pushing the substantial parts of the data collection work onto the hardware. To test this hypothesis, we have designed and evaluated a lightweight online failure prediction approach, called Seer, in which most of the data collection work is performed by fast hardware performance counters – CPU resident counters that record various low level events occurring on a CPU. The hardware-collected data is augmented with further data collected by a minimal amount of software instrumentation that is added to the systems software. In the remainder of the paper, the internal execution data collected by combining hardware and software instrumentation, is referred to as *hybrid spectra*, whereas the internal execution data collected by using software instrumentation only, is referred to as *software spectra*.

In earlier work we introduced a number of different types of hybrid spectra for failure detection, which aim to determine whether the hybrid spectra collected from a deployed system comes from a failed or a successful execution [32]. The results of the aforementioned work strongly suggest that hybrid spectra can reliably distinguish failed executions from successful executions at a fraction of the runtime overhead of using software spectra. This work differs from our earlier work in that the earlier work analyzes hybrid spectra in an offline manner to detect failures after they have occurred (i.e., after the executions have terminated), whereas this work analyzes hybrid spectra in an online manner to predict the manifestation of failures before they actually occur. This required us to develop significantly different types of hybrid spectra as well as significantly different types of analysis techniques than the ones used in [32].

We have evaluated Seer by conducting a series of experiments on three widely-used software systems in the presence of both single and multiple defects. At the lowest level of runtime overheads attained in the presence of single defects, Seer predicted the failures about 54% way through the executions[1] with an F-measure of 0.77 (computed by giving equal importance to precision and recall) and a runtime overhead of 1.98%, on average. At the highest level of prediction accuracies attained in the presence of multiple defects, Seer predicted the failures about 56% way through the executions[1] with an F-measure of 0.88 and a runtime overhead of 2.67%, on average. To demonstrate how much additional information the hardware-collected data provided towards predicting the failures over and above the software-collected data in our hybrid spectra, we compared the performance of the hybrid spectra to that of two correlated software spectra. The empirical results show that the data collected by hardware performance counters was a significantly influential factor in successfully predicting the manifestation of failures. We also compared Seer with six different types of *fault screeners* – an alternative state-of-the-art online failure prediction approach that also uses internal execution data. Seer performed significantly better then the fault screeners.

The contributions of this work can be summarized as follows:

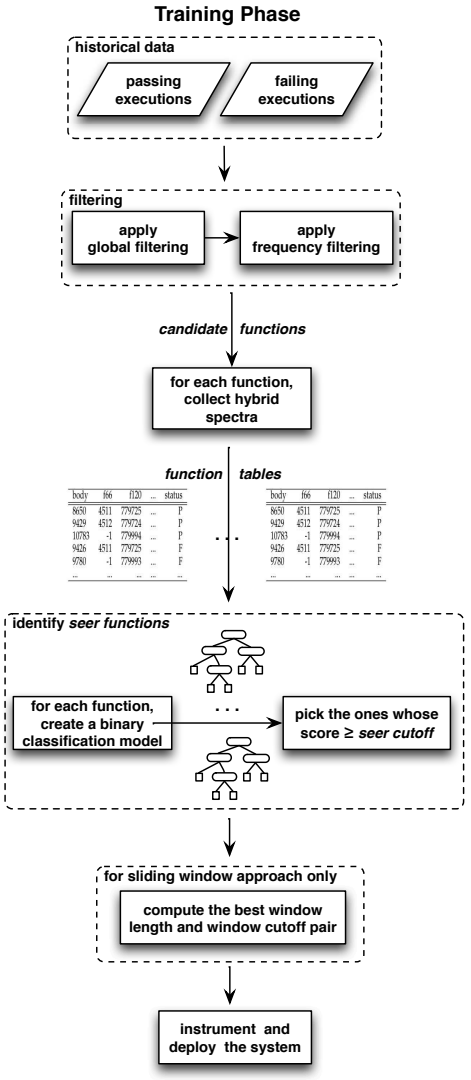
- A novel approach for combining hardware and software instrumentation for online failure prediction and three different types of hybrid spectra produced by using this approach.
- A lightweight online failure prediction approach that uses the proposed hybrid spectra.
- A series of experiments evaluating the performance of the proposed approach and comparing it to that of other related approaches.

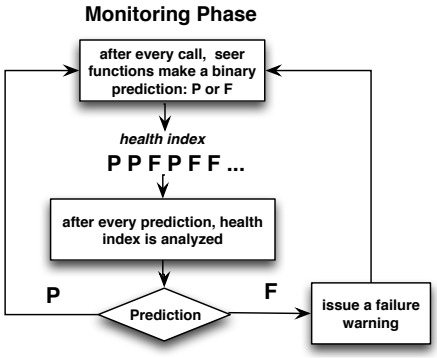
The remainder of the paper is organized as follows: Section 2 presents Seer; Section 3 evaluates the feasibility of Seer; Section 4 evaluates Seer in the presence of single defects; Section 5 compares Seer with six different types of fault screeners; Section 6 evaluates Seer in the presence of multiple defects; Section 7 discusses threats to validity; Section 8 discusses the related work; and Section 9 presents concluding remarks and future work.

2 Approach

Seer tracks internal execution data at the function level and is composed of two phases: an offline *training phase* and an online *monitoring phase*. The training phase takes as input historical data that is comprised of passing and failing executions. First, functions implemented by the system under observation, are filtered out to determine the candidate functions that can potentially be monitored at runtime. Then, hybrid spectra are collected from the historical executions on a per candidate function basis. Next, the spectra collected are used to identify the functions that can predict failures. To this end, for each candidate function, a prediction model in the form of a binary classifier is trained. Then, the best performing functions, i.e., the ones that best distinguish failing executions from passing executions, are determined and marked as *seer functions*. Finally, the seer functions are augmented with their prediction models and the instrumented system is deployed for the monitoring phase.

1. when the duration of an execution is measured as the number of function calls made in the execution, see Section 4 for more information.





Term	Description
global filtering	Filtering out functions that handle already detected failures.
frequency filtering	Filtering out frequently invoked functions.
function table	Hybrid spectra collected for a function using the training set.
prediction model	A binary classification model trained for a function.
function score	F-measure of the prediction model trained for a function.
seer cutoff	Cutoff value used for identifying seer functions.
seer function	A function whose function score ≥ seer cutoff.
seer prediction	A binary prediction (*P* or *F*) made by a seer function.
health index	Sequence of seer predictions made since the beginning of the execution.
Pointwise prediction approach	Issues a failure warning after receiving the first *F* from a seer function.
Sliding window prediction approach	Uses fixed-length sliding windows of seer predictions and issues a failure warning after encountering the first window whose score ≥ window cutoff.
Window length	Length of the windows used in the sliding window prediction approach.
Window score	Sum of the scores of the seer functions that made an F prediction in the window (the score is added for every F prediction made by a seer function).
Window cutoff	Cutoff value used for issuing a failure warning in the sliding window approach.
Monitoring probe	For a seer function, placed at the call sites of the callee functions. Measures the total number of hardware events of interest occurring in a callee.
Prediction probe	Placed for every seer function. Combines the counts collected by monitoring probes and makes predictions about the future of the execution by implementing either the pointwise or the sliding window approach.

Fig. 1: Seer in a nutshell.

In the monitoring phase, after every invocation of a seer function, the hybrid spectra collected during the invocation, is fed to the prediction model of the seer function, such that a binary prediction (i.e., passing or failing) about the future of the execution is made. The sequence of seer predictions made in a fixed-length window, are then analyzed at runtime to predict the manifestation of failures. If the execution is predicted to fail, a warning for a possible impending failure is issued. Once a warning is issued, proactive measures can be taken. However, such measures are beyond the scope of this work. Furthermore, we are mainly concerned with functional failures in this work, rather than non-functional ones.

Figure 1 presents a high level view of Seer together with a list of dedicated terms describing it. Next, we discuss the components of Seer in detail in an order that would make the best sense to the reader, i.e., not necessarily in the order in which these components interact with each other.

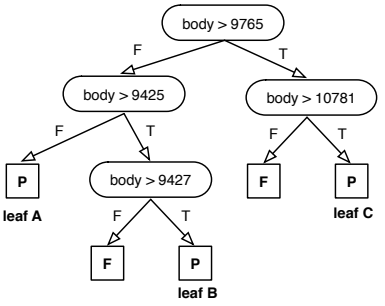
2.1 Collecting Hybrid Spectra

Seer uses hardware performance counters to reduce runtime overheads, while still supporting acceptable levels of prediction quality. In particular, we use fast hardware performance counters to collect raw execution data, but 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

body	f66	f120	...	status
8650	4511	779725	...	P
9429	4512	779724	...	P
10783	-1	779994	...	P
9426	4511	779725	...	F
9780	-1	779993	...	F
...	...	...	...	...

(a)



(b)

```
if (body > 9765){
  if (body > 10781) return P
  else return F
}
else {
  if (body > 9425){
    if (body > 9427) return P
    else return F
  }
  else return P
}
```

(c)

Fig. 2: For function `check_options` ($f19$) implemented in `flex`, a) the function table created, b) the prediction model trained, and c) the prediction code constructed.

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 was that the counters do not distinguish between the instructions issued by different processes [32]. To deal with this, we used a kernel driver in this work, 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. In earlier work, we empirically demonstrated that data collected only from hardware performance counters is generally too coarse to be useful for any software quality assurance activity [32].

To improve this situation, we chose to associate the hardware-collected data with function invocations. That is, we use traditional software instrumentation to indicate which function is currently executing so that different subsets of the hardware-collected data 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, our hybrid spectra in its simplest form is collected as follows: First, a hardware performance counter counting the events of interest is activated at the beginning of a program execution. Next, the value of the counter is read before and after an invocation of a function of interest. The difference between these two readings is the number of events occurred during the execution of the function. We further itemize this event count to reflect the number of events occurred in the body of the function and in each callee function of interest. The event count for the body is computed by subtracting the total number of events occurred in the callee functions of interest from the total number of events occurred in the function. If a callee is invoked multiple times during the execution of the function, the event count for the callee is accumulated over all its invocations. Finally, the hardware performance counter is deactivated at the end of the program execution. In this instrumentation scheme, everything except for counting the events, such as reading the value of a counter, associating event counts with function invocations, and itemizing the event counts as described above, is performed using simple, traditional software instrumentation. The events are counted by hardware performance counters, which are always active during program executions.

Figure 2a depicts an example hybrid spectra we collected in a study conducted on `flex` – a lexical scanner used as a subject application in our experiments. In this spectra we count the machine instructions executed in function `check_options` ($f19$). Each row in the table corresponds to an invocation of $f19$ in an execution. The last column indicates the status of the execution. The rest of the columns present the number of machine instructions executed in the body of $f19$ and in each callee function (e.g., $f66$ and $f120$). Sentinel value -1 indicates that the callee function in the column was not called during the respective invocation of $f19$. For example, the first row corresponds to an invocation of $f19$ that occurred in a successful execution. In this invocation, a total of 8650, 4511, and 779725 machine instructions were executed in the body of $f19$ and in the callee functions $f66$ and $f120$, respectively. In the remainder of the paper these tables are referred to as *function tables*.

The roots of this type of system spectra stem from an earlier work, in which we successfully modeled function execution times in a similar way for fault localization [33]. The rationale can be summarized as follows: Each callee performs a portion of the caller's functionality. Therefore, the way we itemize the event counts in a caller function, reflects how much computation each callee carries out to perform a particular functionality. This provides valuable information, since "suspicious" amount of activities in a function can help predict the manifestation of failures.

2.2 Applying Filtering

To further control the functions that can potentially be monitored at runtime, we have developed two filtering mechanisms: *global filtering* and *frequency filtering*.

Global filtering filters out functions that handle already detected failures, e.g., `error(...)` and `fatal(...)`. Such functions are called only when failures are internally detected by the system under observation. We chose to ignore them because, although such functions are a good indicator of failures, i.e., they appear only in failing executions, they are not a good predictor of failures in the sense that they are called after programs have already failed to gracefully terminate the executions. Thus, such functions provide almost no warning times for failures.

Frequency filtering, on top of global filtering, filters out functions that are invoked more than a predetermined number of times, called *frequency filtering cutoff* (in our case cutoff=50). Note that collecting our hybrid spectra requires to read the value of a hardware performance counter before and after every function invocation. One observation we made in our earlier studies is that even though the cost of reading a hardware performance counter is low (about 45 clock cycles on our experiment platform), the cost is paid every time the counter is read [32]. Therefore, frequency filtering aims to further reduce the runtime overhead by reducing the number of times hardware performance counters are read.

Once a function is filtered out by a filtering mechanism, the function is completely ignored. That is, no software instrumentation is performed for the function in either the training phase or the monitoring phase. All the remaining functions are candidates for becoming a seer function.

2.3 Identifying Seer Functions

Among all the candidate functions, seer functions are the ones that best distinguish failing executions from passing executions. To identify the seer functions, we first create a function table for each candidate function using the historical data, such as the one given in Figure 2a. We then train a prediction model by feeding the function table to a classification tree algorithm (in our case Weka's J48 algorithm [34]). For each candidate function, the result is a binary classification model that attempts to distinguish failing executions from passing ones.

Figure 2b, as an example, presents a prediction model computed for function $f19$ given its function table in Figure 2a. This model tells us that the number of machine instructions executed in the body of $f19$, in short $body$, is strongly correlated with the manifestation of failures. That is, if $9765 < body \leq 10781$ or $9425 < body \leq 9427$, then the execution is likely to fail. Otherwise, the execution is likely to be a successful execution. It turned out that this model was indeed quite effective in predicting failures. An in-depth analysis of what this model actually captured and how it helped predict the failures can be found in Appendix A.

While creating the prediction models, we take several steps to prevent overfitting the data. One standard technique we use is to create the models using n-fold stratified cross-validation (in our case n=5) [34]. This approach builds multiple classification models from different subsets of the input data, and uses the results to identify candidate models that are not overly influenced by a few individual data points.

Once a prediction model is computed for a candidate function, we assign a score to the function, quantifying the success of the function in distinguishing failing executions from passing executions. The score is the F-measure obtained from the cross-validation of the classification model trained for the function. F-measure is a well-known metric, which is computed by giving equal weights to two standard metrics: precision and recall (Section 4.2). It ranges between 0 and 1, inclusive. The higher the F-measure, the better the classification model is.

Not all functions may equally be reliable in distinguishing failures. For this paper, a function is considered to be a seer function, iff, its score is greater than or equal to a predetermined cutoff value, called *seer cutoff* (in our case seer cutoff=0.8, 0.9, or 0.95).

2.4 Predicting Failures

Once seer functions are identified, they are instrumented such that after every call to a seer function, the hybrid spectra collected from the call is fed to the seer's prediction model (see Section 2.5 for details). The model then makes a binary prediction about the future of the execution (P for passing and F for failing). Consequently, the sequence of seer predictions made in an execution can be considered as a string of Ps and Fs. We call this sequence the *health index* of the execution.

Figure 1 presents an example health index. Each literal in a health index represents prediction made by a seer function. If a seer function is not called in an execution, it does not make any predictions. If a seer function is invoked multiple times, one prediction is made after every invocation. Consequently, as seer functions are invoked, the health index grows.

After every prediction made by a seer, we analyze the current health index at runtime to predict the manifestation of failures. To this end, we have developed two approaches: *point-wise prediction approach* and *sliding window prediction approach*.

In the point-wise prediction approach, an execution is predicted to fail after receiving an F prediction from a seer function. In the sliding window prediction approach, on the other hand, failure predictions are based on a sequence of predications made by seer

functions, rather than a single prediction. In particular, this approach uses fixed-length sliding windows of predictions with a slide of one prediction. That is, given window length l, the last l predictions made by seer functions are used for analysis. When a new prediction is received, the window is shifted right by one prediction, such that the newly received prediction becomes the last prediction in the window.

The rationale behind the sliding window approach stems from Zeller's simple failure model [35]. When a defect is exercised in a program execution, it causes an infection, e.g., an erroneous state occurs. The infection propagates over time as the infectious program states are involved in computations, creating even more infectious program states. The infection finally manifests itself as a failure. We conjecture that using a sequence of predictions made over a period of time can improve prediction accuracy, because, in the presence of errors, deviations from normality are likely to increase as time passes.

In the sliding window approach, we dynamically assign a score to every window encountered during an execution. To compute the score, for every F prediction in the window, we add the score of the seer function responsible for the prediction. An execution is predicted to fail after an occurrence of a window whose score is greater than or equal to a pre-determined cutoff value, called *window cutoff*.

Materializing this approach requires to determine the optimal values for the window length and the window cutoff parameters. For that, we have developed a simple approach, in which we take as input the historical data and a range of possible window lengths (in our case the range=[1..10]). For each window length in the given range, we determine and score the windows present in the historical data. We then feed the results to a classification tree algorithm (in our case Weka's J48 algorithm with 10-fold stratified cross-validation) to compute a classification tree of depth 1. For each window length, the result is a window cutoff value that best distinguishes failing executions from passing ones. Finally, among all the window length and window cutoff value pairs trained, we pick the one with the best classification accuracy (i.e., with the best F-measure) and use it in the monitoring phase.

2.5 Instrumenting the System Under Observation

Seer uses two types of probes to instrument the system under observation: *monitoring probes* and *prediction probes*. For a given seer function, monitoring probes are placed at the call sites of the callee functions that are referenced in the prediction model of the seer. The monitoring probe for a callee function, reads the value of the hardware performance counter before and after an invocation of the callee, attributes the difference to the invocation, and accumulates the count over all the invocations of the callee.

Prediction probes, on the other hand, are responsible for making predictions and issuing failure warnings (if necessary). One prediction probe is placed for every seer function. To create the prediction probe for a seer function, we compile the prediction model of the seer into a simple (and possibly nested) if-then-else statement, called *prediction code*. Figure 2c, as an example, presents the prediction code created for the model in Figure 2b.

After every invocation of a seer function, the prediction probe processes the data collected by the monitoring probes and computes a vector of event counts. Each count in this vector represents the number of events observed in the body of the seer function or in a callee function. The vector is then fed to the seer's prediction code. In the point-wise approach, the prediction code simply returns P or F (as shown in Figure 2b). In the sliding window approach, on the other hand, more information is returned in order for the window scores to be computed. In particular, if the prediction is F, the score of the respective seer function is returned. Otherwise, 0 is returned, indicating that the prediction is P. Since the scores of the seer functions are determined in the training phase, they are all hard coded in the prediction codes to reduce the runtime overhead.

After executing the prediction code, the prediction probe applies either the point-wise or the sliding window approach (depending on the configuration). In the point-wise approach, a failure warning is issued after receiving an F prediction from a seer function. In the sliding window approach, the predictions made by seer functions are stored in an array of length l, where l is the window length being used. This array is used in a circular manner to store the last l predictions. If the sum of the scores stored in the array (i.e., window score) is greater than or equal to the window cutoff value determined in the training phase, a failure warning is issued.

3 Evaluating Feasibility

To evaluate Seer, we conducted a series of experiments, which we discuss in the next four sections (including this one). The first study (this section) evaluates the feasibility of Seer. The second study (Section 4) evaluates the performance of Seer. The third study (Section 5) compares Seer with fault screeners. While the first three studies (Sections 3-5) apply the proposed approach to systems with single defects, the fourth study (Section 6) applies it to systems with multiple defects. For each study, we present the research questions, experimental setup, evaluation framework, data and analysis, and a discussion. Furthermore, the data we obtained from these experiments can be found at http://people.sabanciuniv.edu/cyilmaz/seer.

TABLE 1: Statistics about the data set used in the experiments.

subject application	LOC	number of dev. versions	number of faulty versions	total test runs	total passing test runs	total failing test runs
grep	10068	4	6	3660	2886	774
flex	10459	4	24	14458	10435	4023
sed	14427	5	18	6714	5425	1289

An initial question we had is whether seer functions that are capable of reliably distinguishing failing executions from passing executions, exist. If they don't, then the proposed approach will clearly suffer. If they do, then the next question is how much runtime overhead it would impose to use them for failure prediction. In this study we conducted a set of experiments to address these questions at a high level. The results of this study will help us better interpret the detailed results obtained in Sections 4 and 5.

To conduct the study, we used three different types of hybrid spectra: *TOT_INS*, *BRN_TKN*, and *LST_INS*. *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.

These three hardware events were culled from a list of 133 different types of events that could have been monitored on our experiment platform. Clearly, not all these 133 event types may equally be predictive of failures and the predictive power of an event may vary from one type of failure to another. However, it was infeasible for us to evaluate all these events, each of which can indeed be further configured in many different ways, because of the sheer volume of the implied experiment space. Instead, we experimented with only the three events given above. We specifically chose these events because the results of our earlier studies strongly suggest that they can distinguish failing executions from passing executions [32]. One approach to choose the hybrid spectra in practice could be to evaluate different types of hybrid spectra on the historical data and pick the best-performing one.

3.1 Experimental Setup

In the experiments we used three widely used applications as our subject applications: `flex`, `grep`, and `sed`. `Flex` is a lexical scanner that generates fast lexical analyzers. `Grep` is a command line text search utility that prints lines matching a pattern or a regular expression. `Sed` is a stream editor that filters and transforms texts.

All the subject applications were taken from a widely used defect repository, called SIR [36]. SIR provided us with a set of versions for each application, a set of known defects for each version (each of which can separately be activated at will), and a test suite per version together with a test oracle for each test case in the suite. For each version of our subject applications, we first individually activated all the defects known for the version on a one defect at a time basis to create faulty versions with a single defect each (the performance of the proposed approach in the presence of multiple defects is studied in Section 6). We then executed the test suites on the faulty versions of our subject applications and used the test oracles to label the executions as passing (P) or failing (F).

The data set we obtained was comprised of $24,832$ total test runs ($18,746$ passing and $6,086$ failing runs) across 48 faulty versions of our subject applications. By following a similar approach used in [32], these 48 faulty versions were culled from a set of 166 faulty versions using the criteria that the ratio of failing to passing runs was between 0.03 and 1.8. 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 ignored these cases in our analysis. Table 1 provides some descriptive statistics about the data set we used.

To conduct the experiments, we divided this data set into a training set and a test set. For each faulty version of our subject applications, this was done by using stratified sampling (i.e., the proportion of passing and failing test runs were maintained), such that about 70% of all the test runs ended up in the training set and the rest (about 30%) ended up in the test set. Overall, the training set had $17,339$ runs and the test set had the remaining $7,493$ runs. All the training activities in this work were carried out on the training set, whereas all the evaluation activities were carried out on the test set. That is, we evaluated Seer on previously unseen executions.

To instrument the system under observation with monitoring and prediction probes, we used CIL (sourceforge.net/projects/cil) – a source code transformation tool. For this study, we used mocked probes that only measured the number of times they were executed. The actual monitoring and prediction probes are used in Sections 4-6. To program hardware performance counters, we used PAPI (icl.cs.utk.edu/papi) – a platform independent tool to program hardware counters. To identify seer functions, we used Weka's J48 classification tree algorithm with 5-fold stratified cross validation [34] and with seer cutoff value set to 0.8.

Furthermore, frequency filtering cutoff value was set to 50. That is, frequency filtering, on top of global

measure	filtering type	grep hybrid	grep software	flex hybrid	flex software	sed hybrid	sed software
# of candidate functions after filtering	global	61.5	61.5	93.5	93.5	81.4	81.4
	frequency	30.9	30.9	53.1	53.1	27.4	27.4
# of times the candidate functions invoked	global	5087.8	5087.8	7838.9	7838.9	7694.3	7694.3
	frequency	72.4	72.4	255.6	255.6	66.2	66.2
# of seer functions identified	global	16.9	3.1	18.9	6.3	10.7	3.5
	frequency	9.3	0.8	13.4	2.2	5.6	1.2
# of times monitoring probes executed	global	1875.4	945.9	2636.9	1181.2	1324.4	498.5
	frequency	20.9	0.9	77.7	7.8	16.1	3.2
# of times prediction probes executed	global	26.5	4.7	52.8	29.2	18.8	6.3
	frequency	9.2	0.7	16.7	2.8	5.4	1.2

TABLE 2: Some statistics about the seer functions identified.

filtering, filtered out the functions that were invoked more than 50 times. This cutoff value was chosen such that functions that accounted for more than 90% of all invocations were filtered. The information required for this purpose was obtained by analyzing the histograms of invocation counts observed in the experiments.

The experiments were conducted on a Pentium D machine with 1 GB of RAM, running the CentOS 5.2 operating system.

3.2 Evaluation Framework

To evaluate the presence of seer functions, we measured the number of seer functions identified. To evaluate the effect of a filtering mechanism, we applied the filtering and measured the number of candidate functions (i.e., unfiltered functions) as well as the number of times these candidate functions were invoked in executions. To get a rough estimate of the runtime overhead and to study the factors affecting the overhead, we measured the runtime overhead in this study in terms of the percentage of function invocations for which monitoring and prediction probes were executed. In Sections 4-6, on the other hand, we compute the actual runtime overheads.

3.3 Data and Analysis

For each faulty version of our subject applications (a total of 48 faulty versions), hybrid spectra type (3 types of hybrid spectra), and filtering mechanism (global and frequency filtering), we first identified the candidate functions as well as the seer functions. We then measured the number of times the candidate functions were invoked as well as the number of times monitoring and prediction probes were executed.

Table 2 presents the average counts we obtained. Look for the columns titled "hybrid." The columns titled "software" will be discussed in Section 4. All the counts in this table were obtained from only the functions implemented by our subject applications; system functions, such as *printf(...)*, and their invocations were not counted.

We first observed that 53% of all functions, the ones that were filtered out by frequency filtering, accounted for 98% of all function invocations in the executions, on average. That is, using frequency filtering, compared to using global filtering, significantly reduced the number of candidate functions to be possibly monitored from 79 to 37 (by 53%) and function invocations from 6874 to 131 (by 98%), on average.

We then observed that, when global filtering was used, hybrid spectra identified at least one seer function in 99% of the cases (142 out of 144 = 3 hybrid spectra types x 48 faulty versions). For each of the 48 faulty versions of our subject applications, there was at least one hybrid spectra type that identified at least one seer function. An in depth analysis of the results revealed that, the two cases, in which some hybrid spectra types were not able to identify any seer functions, concerned with two different faulty versions of the same subject application. In one case, BR_TKN could not determine any seer functions, but TOT_INS and LST_INS did. In the other case, LST_INS failed to determine any seers, but TOT_INS and BR_TKN succeeded.

Using frequency filtering, however, made the identification of seer functions a slightly more difficult task. When frequency filtering was used, hybrid spectra identified at least one seer function in 94% (135 out of 144) of the cases. An in depth analysis revealed that there was only one faulty version out of 48 faulty versions, for which none of the hybrid spectra types were able to identify any seer functions. For the rest of the faulty versions, there was at least one hybrid spectra identifying at least one seer function.

We believe that the hybrid spectra types used in the experiments failed to identify any seer functions for this one faulty version due to the relatively small number of function invocations present for analysis. For the aforementioned faulty version, after applying frequency filtering, the average number of function invocations present for analysis was 21.4 in passing executions and 26.5 in failing executions, whereas, for the rest of the faulty versions, there were 178.7 invocations in passing executions and 184.7 invocations in failing executions for analysis, on average. Although we believe that such a situation can be improved by increasing the cutoff value used in frequency filtering (i.e., by monitoring more invocations) and/or fine-

tuning the classification algorithm, such that it can work with a small training set, we do not experiment with these potential solution approaches in this work and use the data we obtained as it is in the remainder of the paper.

Regarding the runtime overhead, we first observed that the seer functions were only a small fraction of all functions observed in executions (Table 2). The average number of seer functions was 15.60 (12% of all functions, on average) when global filtering was used and 9.98 (8% of all functions, on average) when frequency filtering was used.

We then observed that, since more seer functions were identified with global filtering than with frequency filtering, using global filtering required to execute significantly more probes, suggesting that frequency filtering can greatly reduce the runtime overhead of the proposed approach. In particular, using global filtering required to execute the probes (either monitoring or prediction) in 31.11% of all function invocations, whereas using frequency filtering required to execute them in 2.32% of all invocations, on average. Furthermore, when frequency filtering was used, the monitoring probes were executed in 1.68% and the prediction probes were executed in 0.64% of all function invocations, on average.

3.4 Discussion

In this feasibility study, for the subject applications, their faulty versions, and the test cases used in the experiments, we observed that 1) seer functions did exist, except for one case, 2) they were only a small fraction of all the functions implemented by the subject applications, and 3) using them for online failure prediction, especially when frequency filtering was used, required to execute monitoring and prediction probes for only a small fraction of all function invocations. These results suggest that the proposed approach can predict the manifestation of failures with low runtime overheads.

4 Evaluating Seer

To test this hypothesis, we conducted a further set of experiments. In these experiments we measured the prediction accuracy and the runtime overhead of the proposed approach as well as the timeliness of the failure warnings issued, i.e., early predictions are better than late predictions.

One specific question we had is how much additional information hardware-collected data in hybrid spectra provides over and above software-collected data towards predicting the manifestation of failures. As discussed in Section 2.1, our hybrid spectra contain both software-collected data, such as caller-callee information, and hardware-collected data, such as the number of machine instructions executed in a function call. Therefore, it is important to factor out the effect of the software-collected data from the results as much as possible to demonstrate the additional information provided by the hardware-collected data.

To this end, we compared the performance of our hybrid spectra to that of two different types of software spectra, which were collected by using software instrumentation only: *Call* and *Visit*. *Call* keeps track of the functions invoked. *Visit* counts the number of times each function is invoked.

As is the case with the hybrid spectra, the software spectra were collected at the function level. For each candidate function, a function table was created. Each entry in these tables, rather than indicating the number of hardware events occurred in a callee function, indicated whether the callee function was invoked (Call) or the number of times the callee function was invoked (Visit). The rest of the analysis was the same for both the hybrid and the software spectra.

We chose to use Call and Visit spectra for three reasons. First, they collect internal execution data, which is either directly collected by our hybrid spectra or correlated with the data collected by them. In particular, both hybrid spectra and Call spectra use caller-callee information. The only difference between them is the presence of event counts collected by hardware performance counters. Moreover, Visit spectra collect internal execution data which is correlated with the data collected by hybrid spectra. For example, the total number of machine instructions executed in a callee function (as is the case in TOT_INS spectra) is generally correlated with the number of times the callee function is invoked. Therefore, the difference between the success of these types of hybrid and software spectra in predicting failures, can safely be attributed to the hardware-collected data. Second, Call and Visit spectra represent two of the most inexpensive types of system spectra that can be collected at the function level. Therefore, they help us better evaluate the runtime overhead of using hybrid spectra by comparing it to some of the lowest overheads that can be achieved in practice. Finally, the same or similar types of software spectra have been used in the past for fault localization [33], [37], [38], suggesting that these types of software spectra can distinguish failing executions from passing executions, which is a prerequisite for predicting failures. From this perspective, we use Call and Visit spectra in a novel way for online failure prediction in this work.

4.1 Experimental Setup

To conduct the experiments, we used the seer functions identified in Section 3 and automatically created the monitoring and prediction probes as described in Section 2.5. The prediction probes, depending on the configuration used in the experiment, implemented

either the point-wise or the sliding window approach. To compute the optimal window length and cutoff value pairs to be used with the sliding window approach, we experimented with the window lengths in $[1..10]$ (see Section 2.4 for more details). We used CIL to instrument the subject applications with the monitoring and the prediction probes and evaluated Seer on the test set (Section 3.1), i.e., on previously unseen executions.

4.2 Evaluation Framework

The evaluations were performed in a multifaceted manner.

4.2.1 Evaluating prediction accuracy

We used several standard metrics to evaluate the prediction accuracy. First, we computed true positives (TP; the number of correctly predicted failures), true negatives (TN; the number of correctly predicted successful executions), false positives (FP; the number of successful executions incorrectly predicted to fail), and false negatives (FN; the number of failed executions incorrectly predicted to be successful). Then, we computed false positive rate $FPR = FP/(FP + TN)$ and false negative rate $FNR = FN/(FN + TP)$. Finally, we computed precision $P = TP/(TP + FP)$, recall $R = TP/(TP + FN)$, and F-measure by giving equal importance to precision (P) and recall (R) as $F\text{-}measure{=}2PR/(P + R)$.

An execution was predicted to fail after receiving the first failure warning, in which case the rest of the predictions were ignored. Otherwise, the execution was predicted to be successful.

4.2.2 Evaluating warning times

In online failure prediction approaches, it is not only about correctly predicting failures, but also about predicting them as early as possible, so that necessary measures against the failures can be taken in time. Therefore, early predictions are better than late predictions.

To evaluate the proposed approach from this perspective, we computed the *warning time* for a failure as the percentage of the function calls made before the prediction. The lower the warning time, the better the proposed approach is. For example, having a warning time of 25% for a failing execution, in which a total of 80 function calls occurred, indicates that the execution was predicted to fail after the 20th function call, i.e., quarter way through the execution when the duration of the execution is measured as the number of function calls made in the execution. We chose to measure the duration of an execution in this way, because function calls provide more information about the execution context than execution times, which in turn can help take better actions against impending failures.

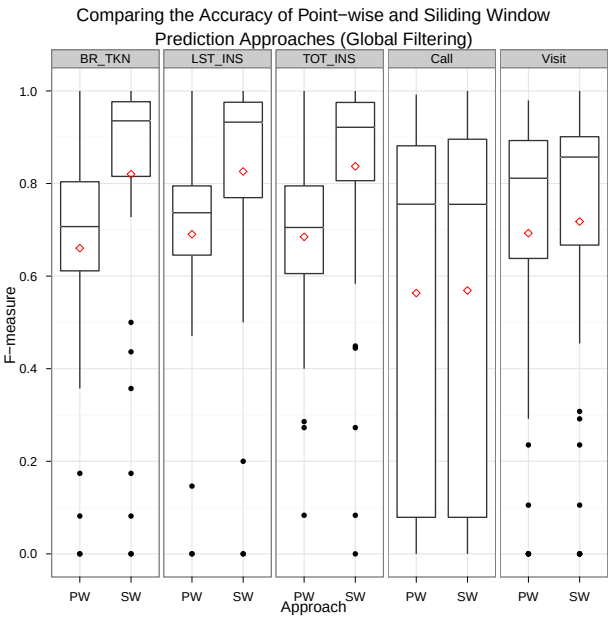


Fig. 3: Comparing the accuracy of point-wise (PW) and sliding window (SW) prediction approaches with global filtering.

Furthermore, when computing warning times, we counted only the calls made to the functions implemented by our subject applications. The rest of the calls, such as the ones made to the functions implemented by the underlying programming language, were not counted. Moreover, each test case used in the experiments was designed to produce an output at the end of the execution. The test oracles that came with the subject applications, determined whether a failure had occurred in an execution or not, only after the output had been produced, i.e., only after the execution was terminated. Therefore, when computing warning times, failures were considered to occur at the end of the executions.

4.2.3 Evaluating runtime overheads

We computed the *runtime overhead* as $((P' - P)/P) * 100$, where P and P' represent the execution time of the original program and that of the instrumented version of the program, respectively, when both programs are executed with the same input.

When measuring the overheads, regardless of the predictions made at runtime, the monitoring and prediction probes were always active during the executions. That is, even if an execution was predicted to fail, all the probes continued to execute. We did this to compute the runtime overheads independently of warning times. Had we deactivated the probes after receiving the first failure warning, then runtime overheads would have depended on warning times.

spectra type	filtering type	average FPR	average FNR	average F-measure	average overhead (%)	average warning time (%)
BR_TKN	global	0.07	0.14	0.82	26.69	38.72
	frequency	0.11	0.23	0.76	2.05	57.15
LST_INS	global	0.08	0.13	0.83	15.55	53.65
	frequency	0.12	0.18	0.78	1.85	55.69
TOT_INS	global	0.07	0.11	0.84	29.21	36.09
	frequency	0.14	0.16	0.78	2.05	47.62
Call	global	0.33	0.38	0.57	1.59	80.42
	frequency	0.47	0.64	0.36	0.66	73.36
Visit	global	0.42	0.54	0.72	12.42	59.13
	frequency	0.16	0.22	0.46	0.77	71.83

TABLE 3: Summary of the results obtained from the sliding window prediction approach.

4.3 Data and Analysis

We first observed that the software spectra identified fewer seer functions than the hybrid spectra (Table 2), suggesting that the hybrid spectra were better at capturing patterns in program executions than the software spectra. The hybrid and software spectra identified an average of 15.60 and 4.83 seer functions, respectively, when global filtering was used, and an average of 9.98 and 1.67 seer functions, respectively, when frequency filtering was used. Furthermore, the hybrid spectra were more resilient to frequency filtering. When frequency filtering was used, compared to using global filtering, the hybrid spectra identified 36% fewer seer functions, whereas the software spectra identified 65% fewer seer functions, on average.

4.3.1 Comparing point-wise and sliding window prediction approaches

We then compared the prediction accuracy of the point-wise (PW) and sliding window (SW) approaches when they were used with global filtering. Figure 3 presents the results we obtained. In this figure, the horizontal axis denotes the prediction approach used and the vertical axis denotes the F-measures obtained. Each box illustrates the distribution of F-measures obtained from a spectra type and prediction approach pair. The lower and the upper ends of boxes represent the first and the third quartiles, respectively. The horizontal bars inside the boxes depict the median F-measures, whereas the diamond shapes depict the average F-measures. The larger the F-measure, the better the approach is.

We first observed that, for the hybrid spectra, the sliding window approach performed significantly better than the point-wise approach in predicting failures. Sliding window approach, compared to point-wise approach, improved the average F-measure from 0.68 to 0.83 (by 22%) for the hybrid spectra. A Kruskal-Wallis test revealed that the difference was statistically significant with a p-value of less than $2.2e-16$. For the software spectra, on the other hand, sliding window approach provided similar F-measures to point-wise approach; an average F-measure of 0.64 for the sliding window approach and 0.63 for the point-wise approach.

We believe that, the sliding window approach improved the prediction accuracy for the hybrid spectra more than it improved the accuracy for the software spectra, because the average window length for the hybrid spectra was 2.6, whereas that for the software spectra was 1.6. As the window length got closer to 1, which is the default window length used in the point-wise approach, the difference between the sliding window and the point-wise approaches diminished. Using the hybrid spectra produced larger window lengths on average, because the hybrid spectra identified more seer functions than the software spectra. Typically, the more seer functions identified, the longer the health indices were, and the longer the health indices, the larger the optimal window lengths were.

Overall, the sliding window approach provided similar or significantly better prediction accuracies than the point-wise approach. Therefore, in the remainder of the paper, we solely focus on the sliding window approach.

4.3.2 Comparing hybrid and software spectra

Table 3 presents the summary of the results obtained from the sliding window approach. When the sliding window approach was used with global filtering, the hybrid spectra provided significantly better prediction accuracy than the software spectra. The average F-measure obtained from the hybrid spectra was 0.83, whereas that obtained from the software spectra was 0.64. The difference was statistically significant with a p-value of less than $7.6e-07$.

Comparing the runtime overhead of using hybrid spectra to that of using software spectra, we, on the other hand, observed that hybrid spectra improved the prediction accuracy at the cost of increased runtime overhead (Figure 4). The runtime overhead of using the hybrid spectra was 24%, whereas that of using the software spectra was 7%, on average. The primary source of this difference was that, since the hybrid spectra identified more seer functions than the software spectra, using the hybrid spectra required

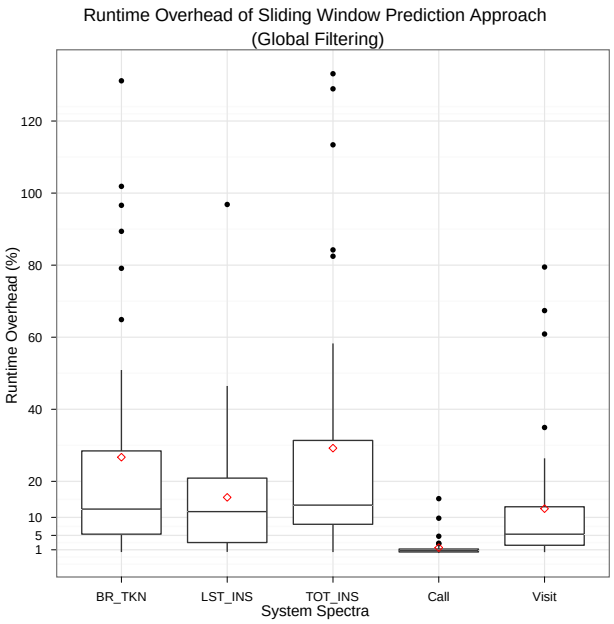


Fig. 4: Runtime overhead of sliding window prediction approach when used with global filtering.

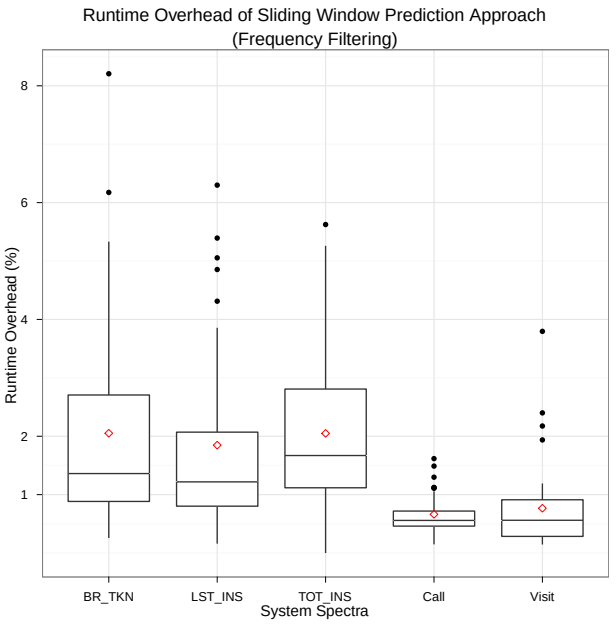


Fig. 5: Runtime overhead of sliding window prediction approach when used with frequency filtering.

to execute significantly more monitoring and prediction probes (Table 2), which in turn incurred more overhead. When global filtering was used, the hybrid spectra required to execute the monitoring probes in 29.54% and the prediction probes in 1.57% of all function invocations, whereas the software spectra required to execute them in 12.87% and 0.60% of all invocations, on average, respectively.

To reduce the number of times monitoring and prediction probes were executed, thus to reduce the runtime overhead, we used the sliding window approach with frequency filtering. Figure 5 presents the runtime overheads we obtained. We observed that frequency filtering significantly reduced the runtime overhead from 24% to 1.98% (by 92%) for the hybrid spectra and from 7% to 0.72% (by 90%) for the software spectra, on average. Overall, monitoring probes accounted for the 44% and prediction probes accounted for the 56% of the total overhead cost, on average (Figure 6).

Comparing the prediction accuracy of the sliding window approach with frequency filtering to that of with global filtering (Figure 7), we first observed that, with frequency filtering, as was the case with global filtering, the hybrid spectra performed significantly better than the software spectra. The average F-measure obtained from the hybrid spectra was 0.77, whereas that obtained from the software spectra was 0.41. The difference was statistically significant with a p-value of less than $6.3e-12$.

We then observed that using frequency filtering made a trade-off between the accuracy and the runtime overhead of the predictions. For the hybrid spectra, frequency filtering, while reducing the average

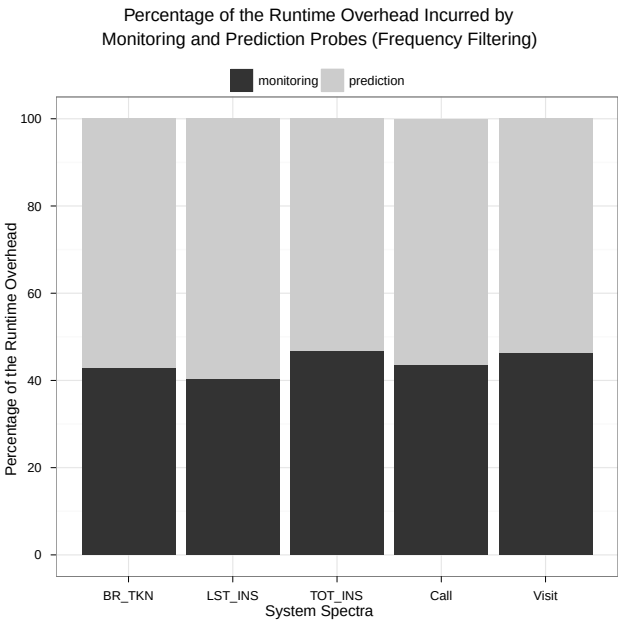


Fig. 6: Percentage of the runtime overhead incurred by the monitoring and prediction probes.

runtime overhead by 92%, reduced the average F-measure by 7.8% (from 0.83 to 0.77). For the software spectra, frequency filtering, while reducing the average runtime overhead by 90%, reduced the average F-measure by 36% (from 0.64 to 0.41).

The hybrid spectra were more resilient to frequency filtering than the software spectra; a drop of 7.8% $vs.$ 36% in the average F-measure. We believe that this was because the hardware counters were still active

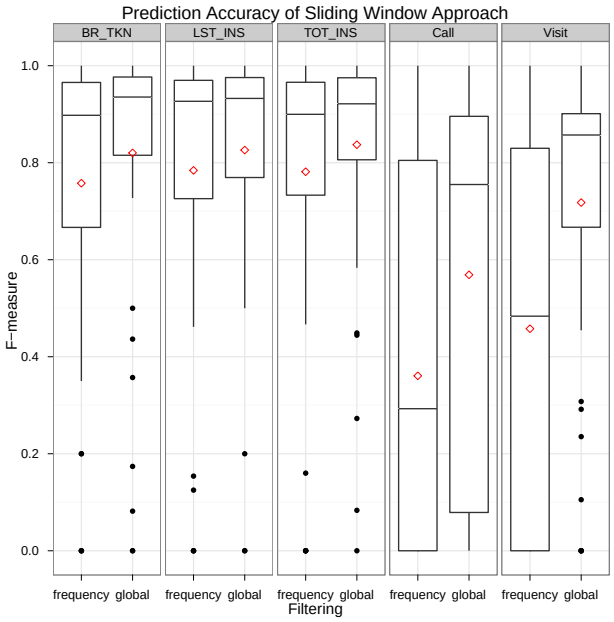


Fig. 7: Prediction accuracy of sliding window approach.

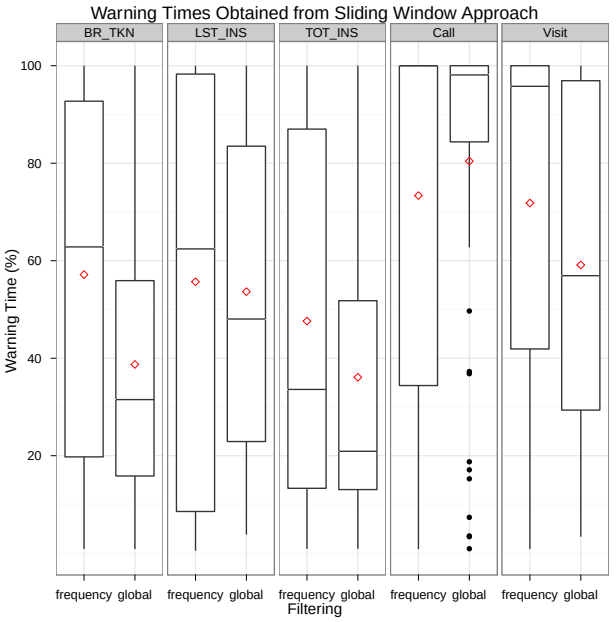


Fig. 8: Warning times obtained from sliding window approach.

during the execution of unmonitored functions (i.e., the ones that were filtered out by frequency filtering); the hardware-collected data was just associated with the functions that called these unmonitored functions. On the other hand, the software spectra were not able to collect any information about the unmonitored functions.

Comparing the warning times obtained from hybrid and software spectra when sliding window approach was used with frequency filtering (Figure 8), we observed that the hybrid spectra significantly improved the warning times by 26%, on average. The difference was statistically significant with a p-value of less than $4.992e-07$. The average warning time for the hybrid spectra was 53.48%, whereas that for the software spectra was 72.60%. That is, when the duration of a program execution is measured as the number of function calls made in the execution (Section 4.2.2), the hybrid spectra predicted the failures about half way through the executions, whereas the software spectra predicted the failures about three-fourth way through the executions, on average.

Comparing the warning times obtained from the sliding window approach with frequency filtering to those obtained with global filtering (Figure 8), we observed that frequency filtering, while reducing the runtime overhead by 92% for the hybrid spectra and by 90% for the software spectra, degraded the average warning time from 42.82% to 53.48% for the hybrid spectra and from 69.77% to 72.60% for the software spectra. Although the hybrid spectra suffered from frequency filtering more than the software spectra, the hybrid spectra still provided significantly earlier warnings for failures, on average.

4.3.3 *An integrated view of prediction quality and runtime overhead*

We now provide an integrated view of the prediction quality and runtime overhead. We observed that using the hybrid spectra with sliding window approach and global filtering provided the best overall prediction quality (Table 3). More specifically, this approach correctly predicted the failures 42.82% way through the executions and with an F-measure of 0.83, on average. However, this was all done at the cost of 24% runtime overhead. Although this approach provided the best prediction quality, we leave it out of the analysis in this section due to its unacceptable runtime overhead.

Figure 9 plots the average runtime overheads against the average F-measures and warning times obtained from the rest of the experiments. One observation is that, for comparable runtime overhead levels, the hybrid spectra provided significantly better prediction accuracies and warning times than the software spectra. For example, the hybrid spectra with frequency filtering and CALL spectra with global filtering have comparable average runtime overheads: 1.98% for the hybrid spectra and 1.59% for Call spectra. However, the hybrid spectra in this setup improved the F-measure by 35% (from 0.57 to 0.77) and the warning time by 34% (from 80.42% to 53.48%), on average, compared to Call spectra.

Alternatively, for comparable accuracy levels, the hybrid spectra incurred significantly less runtime overheads and provided better warning times. For

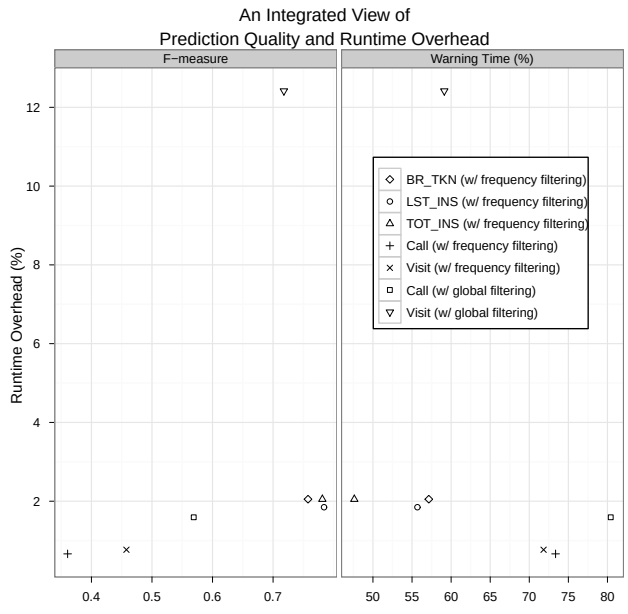


Fig. 9: An integrated view of the prediction quality and runtime overhead of the sliding window approach.

example, the hybrid spectra with frequency filtering and Visit spectra with global filtering have similar F-measures; 0.77 for the hybrid spectra and 0.72 for Visit spectra. However, the hybrid spectra achieved this at a fraction of the overhead compared to Visit spectra. The runtime overhead of the hybrid spectra was 1.98%, whereas that of Visit spectra was 12.42%, on average. This corresponds to a more than 5-fold reduction in the runtime overhead. Furthermore, the hybrid spectra improved the warning times by 10% (from 59.13% to 53.48%) on average, compared to Visit spectra.

Overall, when sliding window prediction approach was used with frequency filtering, the hybrid spectra predicted the failures 53.48% way through the executions with an F-measure of 0.77 and a runtime overhead of 1.98%, on average.

4.4 Discussion

In this study we identified a trade-off between the prediction quality and the runtime overhead of the proposed approach. This trade-off generally is to be expected. Typically, the more internal execution data present for analysis, the better the patterns in program executions can be identified, which can in turn improve the prediction quality. However, typically, the more internal execution data to be collected, the more runtime overhead is incurred. In the experiments we observed that Seer limited the runtime overhead as much as possible while still supporting acceptable levels of prediction quality and that the data collected by hardware performance counters was a significantly influential factor in achieving this.

Seer also offers two tuning parameters to further balance the trade-off between prediction quality and runtime overhead: seer cutoff and frequency filtering cutoff (Section 2). In the presence of excessive number of seer functions, which typically increases the number of times the prediction probes are executed, one can be more selective by increasing the seer cutoff value, so that only the most predictive subset of the candidate functions are selected as seer functions (see Section 6 for a related analysis). In the presence of excessive number of times monitoring probes are executed, one can decrease the frequency filtering cutoff value, so that fewer functions are monitored. In the absence of any seer functions, on the other hand, one can increase the frequency filtering cutoff value to collect more data for analysis and/or reduce the seer cutoff value to be less selective in the seer identification process. Another approach to balance the aforementioned trade-off can be to change the level of monitoring granularity. Although we tracked execution data at the function level in this work, the proposed approach can readily be applicable to finer level monitoring, such as monitoring at the level of code segments, as well as to coarser level monitoring, such as monitoring at the level of components and subsystems.

We also observed that it took about 34 minutes for `flex`, 15 minutes for `grep`, and 8 minutes for `sed` (19 minutes on average) to train Seer after the system spectra for training had been collected. At a high level, the training time grows linearly with the number of functions implemented by the system under observation; a binary classifier is trained for every unfiltered function (Section 2.3). It also increases linearly with the number of historical executions in the training set and with the range of possible window lengths; for each window length and historical execution pair, all windows present in the execution are determined and scored to compute the window length as well as the window cutoff value to be used for predictions (Section 2.4). Therefore, the training time may vary from one system to another. However, one advantage of the Seer's training scheme is that it is embarrassingly parallel in the sense that computing the classification trees for functions as well as performing the operations required for every window length and historical execution pair can all be done in parallel, improving the scalability of the proposed approach. The training times we report above were obtained from a sequential implementation of the Seer's training scheme.

Next, we compare Seer with fault screeners – an alternative online failure prediction approach.

5 Comparing Seer with Fault Screeners

Fault screeners are simple software constructs that predict the manifestation of failures by detecting the presence of "suspicious" values in program executions [39]–[41]. In this approach, values that variables and function arguments take on as well as the values returned from functions are captured to compute likely *value invariants* – a type of invariants that specify the correctness of data values. In the training phase, the invariants are relaxed to accommodate the new values observed. In the monitoring phase, violations of these invariants are flagged as errors.

Fault screeners differ from each other in the types of invariants they use [41]. We in this work distinguish between three types of fault screeners: *single range screeners*, *multiple range screeners*, and *Bloom screeners*.

Single range screeners maintain a tight range that accommodates all the values observed in the training phase [39]. In the monitoring phase, if a value outside this range occurs, then a failure warning is issued.

Multiple range screeners maintain a predetermined number of ranges, rather than a single range. When a new value is observed, the ranges are updated such that the valid range is increased by the least amount [40].

Bloom screeners, on the other hand, use Bloom filters [42] to have a compact representation of the history of variables [41]. A Bloom screener can be considered to be a hash table that is implemented as an array of n bits and that uses k different hash functions. Each hash function maps an element, which is a combination of a value together with the instruction address in the program at which the value is read or written, to one of the n array positions. To add an element in the training phase, the element is fed to each of the k hash functions, k array positions are computed, and the bits at these positions are set to 1. To query for an element in the monitoring phase, the element is fed to the hash functions to get k array positions. If any of the bits at these positions are 0, then it is guaranteed that the element has not been seen before, i.e., the value at the given instruction address was not seen in the training phase. In such a case, a failure warning is issued. On the other hand, if all the bits at the k positions computed by the hash functions, are 1, then either the element was seen in the training phase, or we have a false positive. In either case, no failure warning is issued.

In this work we opted to empirically compare Seer to fault screeners because among all the related works we discuss in Section 8, fault screeners are the most related to our work. This is because they collect internal execution data for online failure prediction and are developed with runtime overhead concerns in mind [39]–[41].

5.1 Experimental Setup

We have implemented single range screeners, multiple range screeners, and Bloom screeners as described in [39]–[41] using LLVM (llvm.org) – a compiler infrastructure providing a collection of modular and reusable compiler and toolchain technologies. In particular, we have implemented a source code transformation pass in LLVM to instrument the source code of our subject applications, such that all accesses to variables (including pointers), function arguments, and return values of primitive types together with the locations of these accesses in the source code, are captured.

In the training phase, all these accesses were recorded to train the screeners. For the single range screeners, we computed a single range for each variable and location pair. For the multiple range screeners, we computed 3 ranges for each variable and location pair, rather than a single range. The number of ranges used in the experiments, i.e., 3, was chosen arbitrarily. For the Bloom screeners, we trained a Bloom filter of size $9,592,955$, which used 7 different hash functions. As is the case in [41], all variable and location pairs shared the same filter. To create the 7 different hash functions, we used the SHA-1 cryptographic hash function with input salting. Furthermore, the size of the filter and the number of hash functions to be used, were determined such that a false positive rate of 0.01 is obtained for a capacity of about one million entries.

In the monitoring phase, the subject applications were augmented with the screeners computed in the training phase. In particular, for every variable and location pair of interest, a software probe was injected. The probe simply captures the value accessed at the location and passes the value and location pair to the respective screener, so that a binary prediction, i.e., P or F, is made at run time. After receiving the first F prediction from a screener, a failure warning is issued. Otherwise, the execution is predicted to be successful.

To train and test the fault screeners, we used the same training and test set that we used in Sections 3-4. We first monitored every possible variable and location pair, which is the default behavior of fault screeners [39]–[41]. Then, to further reduce the runtime overhead incurred by fault screeners, we identified and monitored only those pairs that best distinguished failing executions from passing executions. We call such pairs *able pairs* and the screeners that use them *able screeners*. To find the able pairs, we used the screeners computed in the training phase and measured the individual performance of every variable and location pair on the test set. In particular, we computed an F-measure for every pair and then picked only those pairs whose F-measure is greater than or equal to 0.8. After finding the able pairs, we

screener	FPR	FNR	F-measure	overhead (%)	# of probes injected	# of times probes executed
Regular Single Range	0.09	0.55	0.38	194.40	1534.29	89141.84
Able Single Range	0.01	0.57	0.46	4.07	34.40	2811.26
Regular Multiple Range	0.16	0.46	0.39	221.88	1534.29	89141.84
Able Multiple Range	0.02	0.50	0.52	6.60	43.67	3428.69
Regular Bloom	0.72	0.03	0.45	61499.55	1534.29	89141.84
Able Bloom	0.03	0.41	0.57	14201.67	72.98	3845.39

TABLE 4: Summary of the results obtained from the fault screeners.

trained the able screeners from scratch only for the selected pairs.

Note that the experiments we set up in this section represent an optimal scenario for the fault screeners, in which the test set was made available to the screeners to identify the able pairs. Seer, on the other hand, was evaluated by using previously unseen executions, i.e., by using the previously unknown test set.

5.2 Evaluation Framework

To compare the fault screeners with each other and to reason about their runtime overheads, we measured the number of probes injected into the source code of our subject applications as well as the number of times these probes were executed. To compare the performance of Seer to that of fault screeners, we used the F-measure and the runtime overhead metrics (Section 4.2).

5.3 Data and Analysis

Table 4 summarizes the results we obtained. We first observed that monitoring only the able pairs, compared to monitoring all pairs, increased the average F-measures and significantly reduced the average runtime overheads. The improvements on the runtime overheads were due to the significant reductions in the number of times the probes were executed. Able single, multiple, and Bloom screeners, compared to regular single, multiple, and Bloom screeners, increased the average F-measures by 21% , 33%, and 27%, respectively, while reducing the average runtime overheads by 98%, 97%, and 77%, the average numbers of probes injected by 98%, 97%, and 95%, and the average numbers of times the probes executed by 97%, 96%, and 96%, respectively.

We furthermore observed that, among all the fault screeners, the able Bloom screeners provided the best average F-measure, but they did so at the cost of unacceptable levels of runtime overheads. The average F-measure was 0.57 while the average runtime overhead was about 14202% (Table 4). The excessive overhead was mainly due to cost of computing the hash functions as well as the frequency in which these computations were carried out. Able single range screeners, on the other hand, provided the best runtime overheads among the fault screeners, but they did so at the cost of unacceptable levels of F-measures. The average runtime overhead was 4.07% while the average F-measure was 0.46 (Table 4).

We then compared the performance of fault screeners to that of Seer (Figures 10-11). We observed that, compared to the able Bloom screeners, i.e., the best performing fault screeners in terms of prediction accuracy, Seer statistically significantly (with a p-value of less than 0.004) increased the F-measure by 35%, on average (Figure 10). Compared to the able single range screeners, i.e., best performing fault screeners in terms of runtime overhead, Seer statistically significantly (with a p-value of less than $4.311e{-}10$) reduced the runtime overhead by 51%, on average (Figure 11).

5.4 Discussion

The results obtained from fault screeners in this section as well as the ones obtained from the software spectra used in Section 4, suggest that the failures studied in our experiments were not trivial at all to predict with low runtime overheads. This further emphasizes the importance of the results obtained by Seer.

6 Evaluating Seer with Multiple Defects

We have so far evaluated Seer on systems with single defects. In this study, we apply the proposed approach to systems with multiple defects.

6.1 Experimental Setup

To conduct the study, for each version of our subject systems, we first determined all 2-way and 3-way combinations of the single defects used in Sections 3-4. For each combination, we then created a faulty version by activating the respective defects, resulting in faulty versions with 2 or 3 defects each. After filtering these faulty versions based on their fail/pass ratio as described in Section 3.1, we had a total of 165 faulty versions to work with; 78 versions each with 2 defects and 87 versions each with 3 defects.

We then evaluated Seer on each of these faulty versions. In particular, we used the sliding window prediction approach with frequency filtering, where the frequency cutoff value was set to 50, and experimented with three different seer cutoffs: 0.8, 0.9, and 0.95. The first cutoff value was used to compare

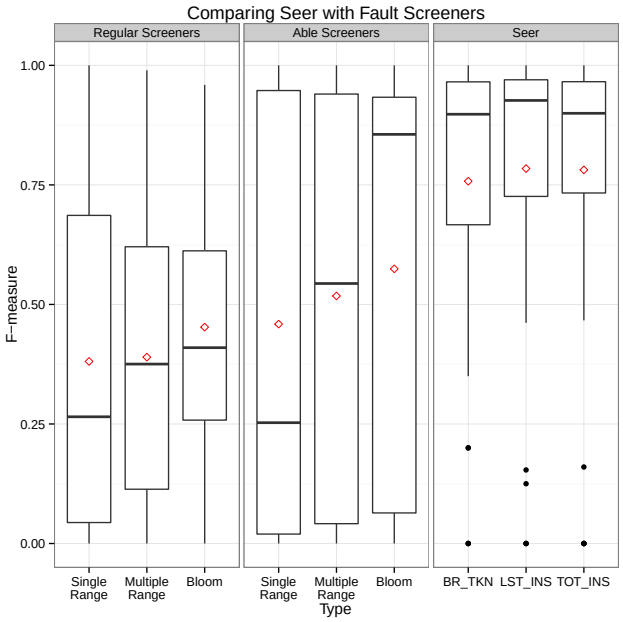


Fig. 10: Comparing the F-measures obtained from Seer to those obtained from fault screeners.

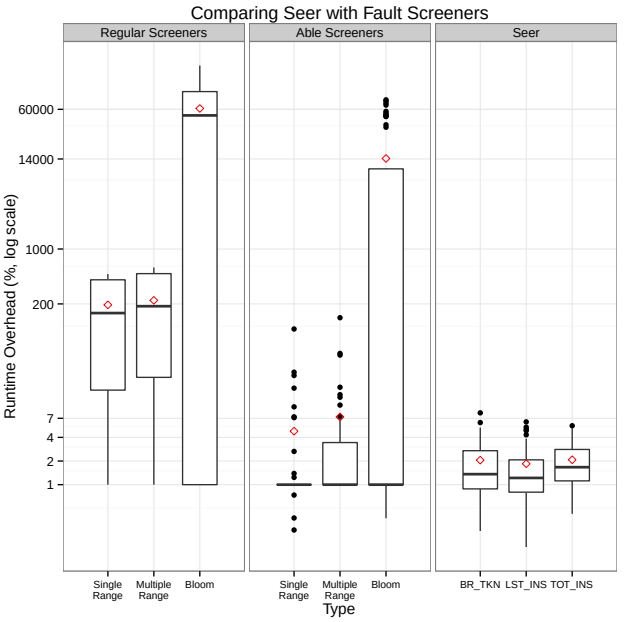


Fig. 11: Comparing the runtime overheads of Seer to those of fault screeners.

the results obtained from multiple defects to those obtained from single defects in Sections 3-4. The rest of the cutoff values, on the other hand, were used to study how Seer was affected by increasing the cutoff.

6.2 Evaluation Framework

For the evaluations, we used the F-measure, runtime overhead, and warning time metrics (Section 4.2). To further reason about the Seer's performance, we used the number of seer functions identified, the window length computed, and the percentage of the function invocations, for which the monitoring and prediction probes were executed (Section 3.2).

6.3 Data and Analysis

We first compared the performance of Seer on multiple defects to that on single defects when the seer cutoff value was set to 0.8. Figures 12-14 summarize the data we obtained in the presence of multiple defects (look for seer cutoff=0.8). We observed that Seer achieved a similar performance in the presence of multiple defects to that in the presence of single defects. The hybrid spectra correctly predicted the failures 56% way through the executions and with an F-measure of 0.88 and a runtime overhead of 2.67%, on average. Compared to the software spectra, the hybrid spectra, improved the F-measure by 66% (the difference was statistically significant with a p-value $< 2.2e-16$) and the warning time by 19% (the difference was statistically significant with a p-value $< 4.215e-14$), on average. In particular, the software spectra correctly predicted the failures 69.07% way through the executions and with an F-measure of 0.53 and a runtime overhead of 1.73%, on average.

We then examined how the performance of hybrid spectra was affected by the number of defects in the systems under observation. Table 5 presents the results we obtained (look for seer cutoff=0.8). We observed that, as the number of defects increased, the prediction accuracy increased as well. The average F-measures obtained from the hybrid spectra were 0.77, 0.86, and 0.90 in the presence of 1, 2, and 3 defects, respectively. We believe that this was because, as the number of defects increased, the number of seer functions identified tended to increase, suggesting that it was easier to distinguish failing executions from passing ones in the presence of multiple defects. The average numbers of seer functions were 9.98, 14.89, and 16.57, respectively. We then observed that the warning time tended to also increase as the number of defects increased. We believe that this was because the window length tended to increase, following the rise in the number of seer functions identified. The average window lengths were 1.95, 2.18, and 2.23 in the presence of 1, 2, and 3 defects, respectively. The average warning times, however, were still between 50 and 60%. Interestingly enough, in the presence of multiple defects, as the number of defects increased from 2 to 3, although the number of seer functions as well as the window length increased, the runtime overhead tended to decrease (Table 5). The average runtime overheads were 2.75 and 2.61, respectively. An in-depth analysis revealed that this was because both the monitoring and the prediction probe invocation percentages tended to decrease, which can be

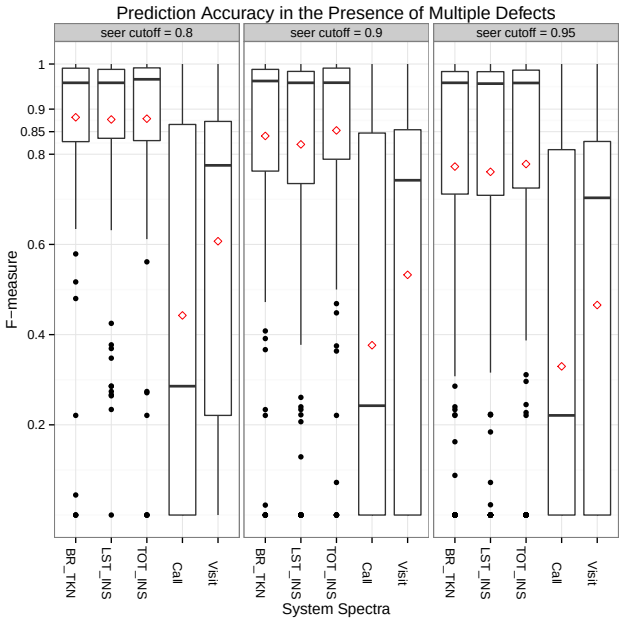


Fig. 12: Prediction accuracies obtained from the sliding window approach with frequency filtering in the presence of multiple defects.

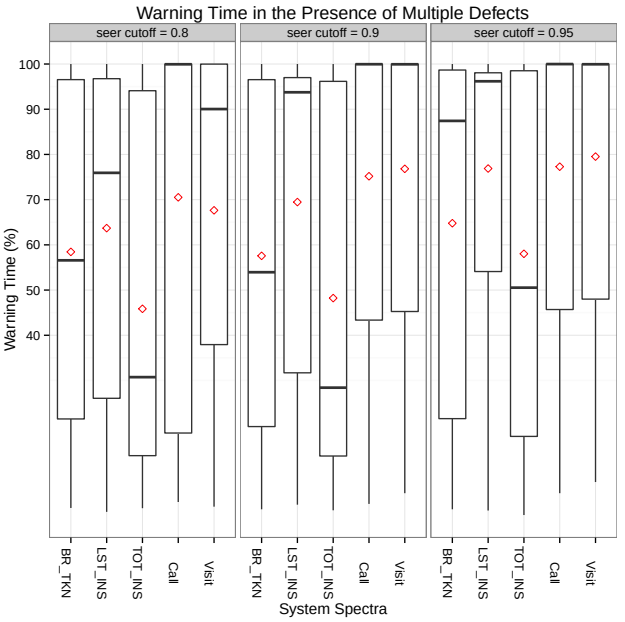


Fig. 13: Warning times obtained from the sliding window approach with frequency filtering in the presence of multiple defects.

attributed to the faulty versions of our subject systems as well as the test cases used in the experiments. The percentages of function invocations, for which the monitoring and prediction probes were executed, were 2.27 and 1.98 in the presence of 2 and 3 defects, respectively.

Comparing the runtime overhead of the proposed approach for single and multiple defects, we observed that the overhead was slightly larger in the presence of multiple defects. The average overhead was 1.98% for single defects and 2.67% for multiple defects. We attribute this to the increase in the percentage of function invocations for which the monitoring and prediction probes were executed, which in turn was caused by an increase in the number of seer functions identified. In the case of single defects, the average number of seer functions was 9.98 and the probes were executed in 2.32% of all function invocations on average. In the case of multiple defects, on the other, the average number of seer functions was 15.78 and the probes were executed in 2.11% of all function invocations on average.

To further reduce the runtime overhead and to further study the tradeoff between the prediction quality and the seer cutoff value, we repeated our multiple-defects experiments by increasing the cutoff value from 0.8 to 0.9 and then to 0.95. The results can be found in Table 5 and Figures 12-14 (look for seer cutoff $>$ 0.8). We observed that as the cutoff value increased from 0.8 to 0.95 the average number of seer functions and the average window length decreased. The average numbers of seer functions were 15.78,

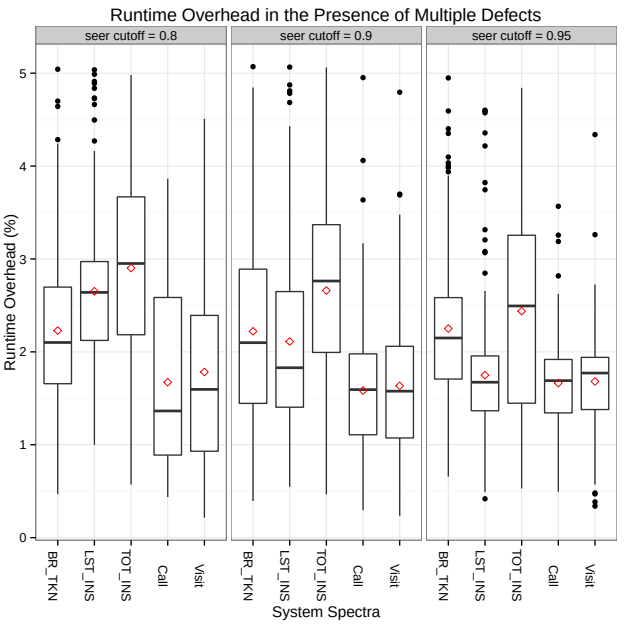


Fig. 14: Runtime overheads obtained from the sliding window approach with frequency filtering in the presence of multiple defects.

11.30, and 8.62, and the average window lengths were 2.21, 1.33, and 1.07, for the seer cutoff value of 0.8, 0.9, and 0.95, respectively. We further observed that, while the average runtime overhead decreased, the average F-measure decreased and the average warning time increased. The average runtime overheads were 2.67%, 2.41%, and 2.22%, the average F-measures

measure	seer cutoff = 0.8 # of defects 1	2	3	seer cutoff = 0.9 # of defects 2	3	seer cutoff = 0.95 # of defects 2	3
F-measure	0.77	0.86	0.90	0.81	0.86	0.73	0.81
Warning Time (%)	53.48	53.36	58.36	56.55	60.09	66.27	66.83
Runtime Overhead (%)	1.98	2.75	2.61	2.47	2.36	2.25	2.19
# of seer functions identified	9.98	14.89	16.57	10.71	11.84	8.16	9.04
Window length	1.95	2.18	2.23	1.29	1.38	≈ 1	1.17
Monitoring probe invocation percentage	1.68	1.65	1.47	1.08	0.92	0.69	0.63
Prediction probe invocation percentage	0.64	0.62	0.51	0.43	0.35	0.29	0.25

TABLE 5: Summary of the results obtained from the hybrid spectra when used with the sliding window approach and frequency filtering for different number of defects as well as different seer cutoff values.

were 0.88, 0.84, and 0.77, and the average warning times were 56%, 58.42%, and 66.56%, respectively.

An interesting observation is that when seer cutoff$=0.9$ in the presence of multiple defects, more seer functions (an average number of 11.30 vs. 9.98) were identified than in the presence of single defects with the cutoff$=0.8$, as a result of which better prediction accuracies were obtained (an average F-measure of 0.84 vs. 0.77). When the cutoff$=0.95$ in the presence of multiple defects, although slightly fewer seer functions (an average number of 8.62 vs. 9.98) were identified than in the presence of single defects with the cutoff$=0.8$, similar prediction accuracies were obtained (the average F-measure was 0.77 in both cases).

6.4 Discussion

All these results suggest that, although some quality assurance activities, such as fault localization, may suffer in the presence of multiple defects, online failure prediction approaches in general and Seer in particular may not get affected as much (or even perform better), because in the presence of multiple defects, executions are likely to deviate more from normality.

7 Threats to Validity

All empirical studies suffer from threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our studies to industrial practice.

One threat concerns the representativeness of the subject applications used in the experiments, namely `grep`, `flex`, and `sed`. Although they are all widely used applications, they only represent three data points. A related threat concerns the representativeness of the defects used in the experiments. Although all the faulty versions of our subject applications were taken from SIR [36] – a defect repository that has been used by many related studies in the literature, they represent only a subset of all defects. To address this threat, we have evaluated the proposed approach in the presence of both single and multiple defects.

The representativeness of the test cases used in the experiments is also of concern. Although all the test cases were taken from SIR, they often had a short life span. With regard to the accuracy of the proposed approach, long running test cases pose no theoretical problems. Typically, the longer the execution, the more internal execution data is collected, which in turn can help identify the patterns better. With regard to the runtime overhead, long running test cases pose no theoretical problems either. The runtime overhead incurred by Seer primarily depends on the ratio of function invocations for which monitoring and prediction probes are executed. Therefore, as long as this ratio is kept under control (and different ways to achieve this have already been discussed in Section 4.4) the overhead can be kept under control.

Another potential threat concerns the representativeness of the types of hybrid spectra we used and the approaches against which we benchmarked the proposed approach. The justification for the choice of the three types of hybrid spectra we used in this work, can be found in Section 3. The reasons for choosing the two types of software spectra and the fault screeners we used for benchmarking, are discussed in Section 4 and Section 5, respectively.

According to the taxonomy proposed by Salfner et al. [3], Seer is a symptom monitoring-based online failure prediction approach that uses classifiers (Section 8). The performance of all classifier-based approaches [21]–[23] is bounded by the representativeness of the passing and failing executions used for training the classifiers. Seer is no exception.

We, however, strongly believe that classifier-based approaches in general and Seer in particular are of great practical importance. One reason is that online failure prediction is not concerned with isolating the defects causing failures (i.e., root cause analysis). Rather, it aims to detect failure inducing program states. As different defects can lead to the same or similar failure inducing states, a classifier trained to detect a particular failure inducing state can be used to predict the failures caused by different defects leading to the same or similar failure inducing state.

For example, consider a program that processes a sequence of transactions. After receiving a transaction,

the program carries out some complex computations and stores the result in a buffer for later processing. Once the end of the sequence has been reached, the results stored in the buffer are processed to compute the final output. Now consider a scenario in which the result buffer does not get populated after processing a transaction. The execution reaches to a failure inducing state, in which the result of the transaction is lost. Therefore, the execution is destined to fail. If this failure inducing state can be detected at runtime, the failure can be predicted as soon as the first transaction producing this failure inducing state is observed, without even waiting for the subsequent transactions. Note that there could be many different defects scattered through the program that prevent the result buffer from being populated properly. However, to predict the manifestation of the failures, all we need is to detect a single failure inducing program state, in which a transaction is processed, but the result buffer is not populated. Therefore, a training set that helps us identify this failure inducing state is sufficient to predict the failures caused by all known as well as unknown defects leading to the same or similar states, demonstrating that not all defects must be known a priori in order to train classifier-based failure prediction models.

Obtaining a representative sample to identify failure inducing program states can still be a difficult task, though. In such cases, Seer can be geared towards predicting the manifestation of particular types of failures rather than attempting to predict all failures. For example, with automated error reporting tools, such as Windows Error Reporting (msdn.microsoft.com), and public bug repositories, such as Bugzilla@Mozilla (bugzilla.mozilla.org), becoming mainstream, developers are not only informed about their system's failures faster but also able to collect more data about the failures than it is used to be. The data collected, besides being used as a debugging aid, is often used to prioritize the failures according to their impact on end users [43]. Even with all the data that developers have about the failures, it may still take an undesirable amount of time for them to pinpoint and fix all the defects. In such cases, Seer can be trained with the failure data collected from the field to predict the manifestation of specific failures that affect the end users the most. With the proactive measures for the impending failures in place, this approach can greatly improve the reliability of the system by making the end users suffer less until the defects are pinpointed and fixed.

8 Related Work

Online failure prediction differs from reliability prediction [44] in that reliability predictions, such as predicting the failure rates of software components, are long-term predictions made in an offline manner with no regard to the runtime state of the system, whereas online failure predictions are short-term predictions made at runtime on the basis of information obtained by monitoring a running system.

Online failure prediction approaches. Salfner et al. [3] classify the existing online failure prediction approaches into three main categories: failure tracking-based [4]–[6], error reporting-based [7]–[16], and symptom monitoring-based approaches [17]–[31].

Failure tracking-based approaches analyze the occurrences of previous failures to reason about future failures. Pfefferman et al. analyze the time-instants at which earlier failures occurred to estimate the probability distribution of the time to next failure [4]. Liang et al. use both temporal and spatial correlations between previously reported failures for online failure prediction [5]. Fu et al. present a method for quantifying temporal and spatial correlations between failure events [6].

While failure tracking-based approaches analyze failure logs, error reporting-based approaches analyze error logs to predict the manifestation of failures. These approaches differ from each other in that errors, such as unexpected system states, may not always cause failures, which are externally observable unexpected system behaviors. Hatonen et al. [9], Vilalta et al. [12], and Weiss [11], mine error logs to identify simple, yet predictive rules, such as "if errors A and B occur within 5 seconds of each other, then the system is likely to crash." The rules are then checked at runtime and failure warnings are issued accordingly. Nassar et al. [7], Lin et al. [8], Lal et al. [10], and Levy et al. [14] analyze temporal and spatial correlations between errors to predict failures. Vilalta et al.[13], Salfner et al. [15], and Salfner and Malek [16] use pattern recognition techniques to identify the patterns in error logs that are predictive of failures.

Symptom monitoring-based approaches, on the other hand, attempt to identify error symptoms leading to failures. Vaidyanathan et al. [17], Li et al.[45], Andrzejak et al. [18], Kapadia et al. [20], and Hoffmann [19] use function approximation techniques to compute a function that maps the values of monitored system variables to values of target variables that are capable of predicting failures. Although the current values of target variables can be measured at runtime, function approximation is used to extrapolate the target values into the future to predict failures. Hamerly et al. [21], Murray et al. [22], and Bodik et al. [23] train classifiers to identify a boundary between failure-prone and non-failure-prone system states. To make a prediction, the current system state is checked against the boundary to determine which side of the boundary the state belongs to. Hughes et al. [24], Murray et al. [22], Gross et al. [46], and Cassidy et al. [25], on the other hand, compute observed behavior models that capture the behavior of the system as observed in successful executions. Then,

in previously unseen executions, the deviations from these models are quantified to predict failures. Garg et al. [26], Cheng et al. [27], Shereshevsky et al. [28], Hellerstein et al. [29], and Meng et al. [30] use time series analysis techniques for online failure prediction. Fault screeners we have studied in Section 5, are also a type of symptom monitoring-based online failure prediction approach [39]–[41].

Architecture-based self-adaptation, which falls into the category of symptom monitoring-based approaches, has also been studied [47], [48]. These approaches typically take as input an abstract architectural model of the system, use this model to monitor the system's runtime properties, evaluate the model for violations of predetermined correctness constraints, and perform adaptions at the level of architectural elements (if needed). Casanova et al. introduce an online architecture-based fault localization approach, which is generally geared towards quality-of-service-related non-functional failures [49], [50]. Given an architectural model of the system and a set of manually-developed oracles that detect system errors, this approach produces a ranked list of suspicious architectural elements that might have caused the errors.

According to the taxonomy given above, Seer is a symptom monitoring-based online failure prediction approach that uses classifiers. It differs from the existing symptom monitoring-based approaches discussed above in that these approaches (except for architecture-based self-adaptation approaches and fault screeners) treat the system under observation as a black-box and use only the data which is directly observable from outside executions, such as CPU and memory utilization, latency and throughput of the system, and the number of active threads and processes, etc. Seer, on the other hand, treats the system under observation as a white box and collects data from inside executions. Seer is also different from architecture-based self-adaptation approaches and fault screeners, which collect internal execution data. The differences between fault screeners and Seer have been extensively studied in Section 5. Seer differs from architecture-based approaches discussed above in that it predicts the manifestation of functional failures, rather than non-functional failures, and does so without requiring an abstract model of the system or a set of predetermined oracles. Finally, Seer differs from failure tracking- and error reporting-based approaches in that these approaches operate only on the data collected by the system under observation and require the system to internally detect its own failures or errors. Seer, on the other hand, decides what needs to be collected, which may not spontaneously be collected by the system under observation, and does not expect the system to internally detect its own failures or errors.

Hardware performance counters. Hardware performance counters have been used for performance debugging [51], offline failure detection [32], failure classification [52], and fault localization [53]. Seer, on the other hand, uses them for online failure prediction. Racunas et al. implement special-purpose hardware components to identify "suspicious" changes in the values computed by static machine instructions to detect and recover from hardware-caused failures [54]. Seer is different in that we use general-purpose hardware (i.e., hardware performance counters) to predict software-caused failures.

Data-driven approaches for improving software quality. In recent years, numerous researchers have proposed data-driven approaches based on collecting and analyzing internal execution data to improve the quality of software systems. At a high level, these techniques typically follow the general approach of instrumenting the system under test, collecting data every time the instrumentation code is executed, analyzing the resulting data, and then acting on the analysis results. Some example applications of this general approach include failure detection, failure classification, and fault localization [33], [43], [55]–[61]. We believe that many types of software spectra used in these approaches, can also be used for online failure prediction. Elbaum et al. partially evaluate this hypothesis [31]. However, their work is primarily concerned with prediction accuracy and do not address the issues regarding runtime overhead. Therefore, the tradeoff between the prediction quality and the runtime overhead of using existing software spectra to predict the manifestation of failures, is still of great practical concern and yet to be evaluated. In this work we, on the other hand, combine hardware and software instrumentation for online failure prediction.

9 Concluding Remarks and Future Work

In this work we developed a lightweight online failure prediction approach, in which most of the data collection work is performed by fast hardware performance counters. The hardware-collected data is augmented with further data collected by a minimal amount of software instrumentation that is added to the systems software. In particular, we introduced three different types of hybrid spectra for online failure prediction.

We then conducted a series of experiments to evaluate the proposed approach. In these experiments, we used three widely used open source applications as our subject applications. At the lowest level of runtime overheads attained in the presence of single defects, Seer predicted the failures about 54% way through the executions with an F-measure of 0.77 and a runtime overhead of 1.98%, on average. At the highest level of prediction accuracies attained in the presence of multiple defects, Seer predicted the

failures about 56% way through the executions with an F-measure of 0.88 and a runtime overhead of 2.67%, on average.

To demonstrate how much additional information the hardware-collected data provided towards predicting the failures over and above the software-collected data in our hybrid spectra, we compared the performance of our hybrid spectra to that of two correlated software spectra that are collected by using software instrumentation only. For comparable runtime overhead levels, hybrid spectra provided significantly better prediction accuracies and warning times. Alternatively, for comparable prediction accuracy levels, hybrid spectra incurred significantly less runtime overheads and provided better warning times. These results strongly suggest that the data collected by hardware performance counters was a significantly influential factor in predicting the manifestation of failures.

We also compared the proposed approach with six different types of fault screeners. Compared to the best performing fault screener in terms of prediction accuracy, the proposed approach significantly improved the prediction accuracy. Compared to the best performing fault screener in terms of runtime overhead, the proposed approach significantly reduced the runtime overhead.

All these results support our basic hypothesis that large cost reductions in collecting internal execution data for online failure prediction may derive from pushing the substantial parts of the data collection work onto the hardware, while still supporting acceptable levels of prediction quality.

We believe that this line of research is novel and interesting. We are therefore continuing to investigate how combining hardware and software instrumentation can improve software quality. One possible avenue for future research is to use multiple hardware performance counters simultaneously for online failure prediction, rather than using a single counter in isolation. Another avenue is to enhance the existing online failure prediction approaches with the data collected by hardware performance counters. Yet another possible direction is to design special-purpose hardware components for online failure prediction.

10 Acknowledgments

This research was supported by a Marie Curie International Reintegration Grant within the 7th European Community Framework Programme (FP7-PEOPLE-IRG-2008), and by the Scientific and Technological Research Council of Turkey (109E182).

References

[1] M. Malek, F. Salfner, and G. A. Hoffmann, "Self-rejuvenation: An effective way to high availability," in *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, (Bertinoro, Italy), June 2004.

[2] F. Salfner, G. A. Hoffmann, M. Malek, O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, van A., and M. Steen, "Prediction-based software availability enhancement," *Lecture Notes in Computer Science*, vol. 3460, pp. 143–157, 2005.

[3] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, pp. 10:1–10:42, Mar. 2010.

[4] J. D. Pfefferman and B. Cernuschi-Frías, "A nonparametric nonstationary procedure for failure prediction," *IEEE Trans. Rel.*, vol. 51, no. 4, pp. 434–442, 2002.

[5] Y. Liang, Y. Zhang, M. Jette, A. Sivasubramaniam, and R. Sahoo, "BlueGene/L failure analysis and prediction models," in *Proc. 2006 International Conference on Dependable Systems and Networks (DSN '06)*, pp. 425–434, 2006.

[6] S. Fu and C.-Z. Xu, "Quantifying temporal and spatial correlation of failure events for proactive management," in *Proc. 26th IEEE International Symposium on Reliable Distributed Systems*, pp. 175–184, 2007.

[7] F. A. Nassar and D. M. Andrews, "A methodology for analysis of failure prediction data," in *Proc. 1985 IEEE Real-Time Systems Symposium*, pp. 160–166, 1985.

[8] T. Lin and D. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Trans. on Rel.*, vol. 39, no. 4, pp. 419–432, Oct.

[9] K. Hatonen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen, "Tasa: Telecommunication alarm sequence analyzer or how to enjoy faults in your network," in *Proc. 1996 IEEE Network Operations and Management Symposium*, vol. 2, pp. 520–529, 1996.

[10] R. Lal and G. Choi, "Error and failure analysis of a unix server," in *Proc. 3rd IEEE High-Assurance Systems Engineering Symposium*, pp. 232–239, 1998.

[11] G. M. Weiss, "Timeweaver: A genetic algorithm for identifying predictive patterns in sequences of events," in *the Proc. of Genetic and Evolutionary Computation Conf.*, pp. 718–725, 1999.

[12] R. Vilalta and S. Ma, "Predicting rare events in temporal domains," in *Proc. 2002 IEEE International Conference on Data Mining*, pp. 474–481, 2002.

[13] R. Vilalta, C. Apte, J. Hellerstein, S. Ma, and S. Weiss, "Predictive algorithms in the management of computer systems," *IBM Systems Journal*, vol. 41, no. 3, pp. 461–474, 2002.

[14] D. Levy and R. Chillarege, "Early warning of failures through alarm analysis a case study in telecom voice mail systems," in *Proc. 14th International Symposium on Software Reliability Engineering*, pp. 271–280, 2003.

[15] F. Salfner, M. Schieschke, and M. Malek, "Predicting failures of computer systems: A case study for a telecommunication system," in *Proc. 20th International Parallel and Distributed Processing Symposium*, pp. 8 pp.–, 2006.

[16] F. Salfner and M. Malek, "Using hidden semi-Markov models for effective online failure prediction," in *IEEE International Symposium on Reliable Distributed Systems*, pp. 161–174, 2007.

[17] K. Vaidyanathan and K. Trivedi, "A measurement-based model for estimation of resource exhaustion in operational software systems," in *International Symposium on Software Reliability Engineering*, pp. 84–93, 1999.

[18] A. Andrzejak and L. Silva, "Deterministic models of software aging and optimal rejuvenation schedules," in *Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 159–168, 2007.

[19] G. A. Hoffmann, *Failure prediction in complex computer systems: A probabilistic approach.* PhD thesis, 2006.

[20] N. Kapadia, J. A. B. Fortes, and C. Brodley, "Predictive application-performance modeling in a computational grid environment," in *Proc. 8th International Symposium on High Performance Distributed Computing*, pp. 47–54, 1999.

[21] G. Hamerly and C. Elkan, "Bayesian approaches to failure prediction for disk drives," in *Proc. 18th International Conference on Machine Learning*, (San Francisco, CA, USA), pp. 202–209, Morgan Kaufmann Publishers Inc., 2001.

[22] J. Murray, G. Hughes, and K. Kreutz-Delgado, "Hard drive failure prediction using non-parametric statistical methods," in *Int'l Conf. on Artificial Neural Networks and Int'l Conf. on Neural Information Processing*, 2003.

[23] P. Bodik, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. Jordan, and D. Patterson, "Combining visualization and statistical analysis to improve

operator confidence and efficiency for failure detection and localization," in *Proc. 2nd International Conference on Autonomic Computing*, pp. 89–100, 2005.

[24] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan, "Improved disk-drive failure warnings," *IEEE Trans. Rel.*, vol. 51, no. 3, pp. 350–357, 2002.

[25] K. Cassidy, K. Gross, and A. Malekpour, "Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers," in *Int'l Conf. on Dependable Systems and Networks*, pp. 478–482, 2002.

[26] S. Garg, A. Van Moorsel, K. Vaidyanathan, and K. Trivedi, "A methodology for detection and estimation of software aging," in *Proc. 9th International Symposium on Software Reliability Engineering*, pp. 283–292, 1998.

[27] F.-T. Cheng, S.-L. Wu, P.-Y. Tsai, Y.-T. Chung, and H. C. Yang, "Application cluster service scheme for near-zero-downtime services," in *Proc. 2005 ICRA*, pp. 4062–4067, 2005.

[28] M. Shereshevsky, J. Crowell, B. Cukic, V. Gandikota, and Y. Liu, "Software aging and multifractality of memory resources," in *Proc. 2003 International Conference on Dependable Systems and Networks*, pp. 721–730, 2003.

[29] J. Hellerstein, F. Zhang, and P. Shahabuddin, "An approach to predictive detection for service management," in *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management*, pp. 309–322, 1999.

[30] H.-N. Meng, Y. Qi, D. Hou, and Y. Chen, "A rough wavelet network model with genetic algorithm and its application to aging forecasting of application server," in *Int'l Conf. on Machine Learning and Cybernetics*, vol. 5, pp. 3034–3039, 2007.

[31] S. Elbaum, S. Kanduri, and A. Amschler, "Anomalies as precursors of field failures," in *Proc. 14th International Symposium on Software Reliability Engineering*, pp. 108–118, 2003.

[32] C. Yilmaz and A. Porter, "Combining hardware and software instrumentation to classify program executions," in *Int'l Symposium on Foundations of Software Engineering*, pp. 67–76, 2010.

[33] C. Yilmaz, A. Paradkar, and C. Williams, "Time will tell: fault localization using time spectra," in *Proc. 30th International Conference on Software Engineering (ICSE '08)*, pp. 81–90, 2008.

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

[35] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005.

[36] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Soft. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[37] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pp. 595–604.

[38] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *ECOOP 2005 - Object-Oriented Programming* (A. Black, ed.), vol. 3586 of *Lecture Notes in Computer Science*, pp. 733–733, Springer Berlin / Heidelberg, 2005.

[39] R. Abreu, A. González, P. Zoeteweij, and A. J. C. van Gemund, "Automatic software fault localization using generic program invariants," in *Proceedings of the 2008 ACM Symposium on Applied Computing*, pp. 712–717, 2008.

[40] J. Santos and R. Abreu, "Lightweight automatic error detection by monitoring collar variables," in *Testing Software and Systems*, vol. 7641 of *Lecture Notes in Comp. Sci.*, pp. 215–230, 2012.

[41] R. Abreu, A. González, P. Zoeteweij, and A. J. Van Gemund, "On the performance of fault screeners in software development and deployment," *Proceedings of the 3rd International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'08)*, pp. 123–130, 2008.

[42] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

[43] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. 25th International Conference on Software Engineering (ICSE '03)*, pp. 465–475, 2003.

[44] M. R. Lyu, ed., *Handbook of software reliability engineering*. Hightstown, NJ, USA: McGraw-Hill, Inc., 1996.

[45] L. Li, K. Vaidyanathan, and K. Trivedi, "An approach for estimation of software aging in a web server," in *Int'l Symposium on Empirical Software Engineering*, pp. 91–100, 2002.

[46] K. Gross, V. Bhardwaj, and R. Bickford, "Proactive detection of software aging mechanisms in performance critical computers," in *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, pp. 17–23, 2002.

[47] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, pp. 46–54, Oct. 2004.

[48] J. Cmara, R. de Lemos, M. Vieira, R. Almeida, and R. Ventura, "Architecture-based resilience evaluation for self-adaptive systems," *Computing*, vol. 95, no. 8, pp. 689–722, 2013.

[49] P. Casanova, B. Schmerl, D. Garlan, and R. Abreu, "Architecture-based run-time fault diagnosis," in *Software Architecture*, vol. 6903 of *Lecture Notes in Computer Science*, pp. 261–277, 2011.

[50] P. Casanova, D. Garlan, B. Schmerl, and R. Abreu, "Diagnosing architectural run-time failures," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 103–112, 2013.

[51] N. Smeds, "OpenMP application tuning using hardware performance counters," in *Proc. 2003 International Conference on OpenMP Shared Memory Parallel Programming*, (Berlin, Heidelberg), pp. 260–270, Springer-Verlag, 2003.

[52] B. Ozcelik, K. Kalkan, and C. Yilmaz, "An approach for classifying program failures," in *Int'l Conf. on the Advances in System Testing and Validation Lifecycle*, pp. 93–98, 2010.

[53] C. Yilmaz, "Using hardware performance counters for fault localization," in *Proc. 2010 International Conference on the Advances in System Testing and Validation Lifecycle*, pp. 87–92, 2010.

[54] P. Racunas, K. Constantinides, S. Manne, and S. Mukherjee, "Perturbation-based fault screening," in *Int'l Symposium on High Performance Computer Architecture*, pp. 169–180, Feb 2007.

[55] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Int'l Conf. on Software Engineering*, pp. 467–477, ACM, 2002.

[56] G. Hoglund and G. McGraw, *Exploiting software: How to break code*. Addison-Wesley Publishing Company, 2004.

[57] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *SIGPLAN Not.*, vol. 38, no. 5, pp. 141–154, 2003.

[58] W. Dickinson, D. Leon, and A. Podgurski, "Pursuing failure: the distribution of program failures in a profile space," in *Proc. 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 246–255, September 2001.

[59] W. Dickinson, D. Leon, and A. Podgursky, "Finding failures by cluster analysis of execution profiles," in *Int'l Conf. on Software Engineering*, pp. 339–348, May 2001.

[60] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil, "Applying classification techniques to remotely-collected program execution data," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 146–155, 2005.

[61] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *Int'l Symp. on Software Testing and Analysis*, pp. 195–205, July 2004.

Burcu Ozcelik received the BS and MS degrees in computer science and engineering from Sabanci University, Istanbul, Turkey, in 2009 and 2012, respectively. She is currently working as a freelance software developer.

Cemal Yilmaz received the BS and MS degrees in computer engineering and information science from Bilkent University, Ankara, Turkey, in 1997 and 1999, respectively. In 2005, he received the PhD degree in computer science from the University of Maryland at College Park. Between 2005 and 2008, he worked as a post-doctoral researcher at IBM Thomas J. Watson Research Center, Hawthorne, New York. He is currently an assistant professor of computer science on the Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul, Turkey. His current research interests include software engineering and software quality assurance.