Lightweight Runtime Failure Prediction

by Burcu Özçelik

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabanci University

February, 2012

Lightweight Runtime Failure Prediction

APPROVED BY:

Assist. Prof. Dr. Cemal Yılmaz (Thesis Advisor)	[illegible]
Assoc. Prof. Dr. Albert Levi	[illegible]
Assoc. Prof. Dr. Erkay Savaş	[illegible]
Assoc. Prof. Dr. A. Berrin Yanıkoğlu	[illegible]
Assist. Prof. Dr. Hakan Erdoğan	[illegible]

DATE OF APPROVAL: February 1, 2012

Lightweight Runtime Failure Prediction

Burcu Özçelik
CS, Master's Thesis, 2012
Thesis Supervisor: Cemal Yilmaz

Keywords: Software Quality Assurance; Dynamic Program Analysis; Runtime Failure Prediction.

Abstract

Software systems are getting increasingly complex and bigger in size. When these general trends are coupled with the shortcomings of software quality assurance techniques and time-to-market pressures, development houses are forced to release their software with many known and unknown defects, which inevitably cause failures in the field.

Many approaches have been proposed in the literature to predict the manifestation of software failures at runtime and proactively take preventive measures, such as preventing the failures or decreasing their harmful consequences. Runtime prediction of failures is an integral part of such proactive-preventive frameworks.

One downside of the existing approaches is that they treat software systems as a black-box and leverage only the profiling data which are directly observable from outside the programs, such as, CPU, memory, and network utilizations. Internal execution data is typically not leveraged. This is solely due to the potential runtime overhead cost that can be imposed by collecting internal execution data while the programs are running. As the failure prediction approaches target software systems operating in the field, high overhead costs are generally not acceptable. Consequently, the existing approaches mainly target at predicting failures caused by software aging.

In this thesis, we present a lightweight runtime failure prediction approach that leverages internal execution data. We, furthermore, evaluate the approach by conducting a series of large-scale experiments, in which three widely-used software applications were used as subject applications. The results of our experiments strongly suggest that the proposed approach can reliably predict software failures at an affordable cost.

Hafif Yüklü Çevirimiçi Hata Tahmini

Burcu Özçelik

CS, Yüksek Lisans Tezi, 2012

Tez Danışmanı: Cemal Yilmaz

Anahtar Kelimeler: Yazılım Kalite Güvencesi; Dinamik Program Analizi; Çevirimiçi Hata Tahmini.

Özet

Yazılım sistemleri gün geçtikçe büyümekte ve karmaşıklaşmaktadır. Bu genel eğilimlerin üzerine, yazılım kalite güvencesi tekniklerinin yetersizlikleri ve günümüzdeki market baskıları eklenince, yazılımlar içlerinde bilinen ve bilinmeyen birçok yanlışlarla sahaya sürülmekte ve bu yanlışlar kaçınılmaz olarak hatalara sebebiyet vermektedir.

Literatürde, yazılımların sahadaki güvenilirliklerini artırmak için, meydana gelebilecek hataları önceden tahmin etmeyi ve bu hataların oluşumlarını engellemeyi veya hataların verebilecekleri zararları en aza indirgemeyi amaçlayan birçok yöntem yer almıştır. Bu yöntemlerin en önemli işlevsel parçası, hataların önceden tahmin edilmesidir.

Literatürde önerilen hata tahmin yöntemlerindeki olumsuz yan, bu yöntemlerin yazılım sistemlerine kara kutu muamelesi yapması ve işlemci, hafıza ve ağ kullanımı gibi sadece dışarıdan gözlemlenen özellikleri kullanarak telemetri verisi toplamasıdır. İçsel telemetri verisi genel olarak kullanılmamıştır. Bunun tek nedeni çalışmakta olan programdan veri toplanması sırasındaki olası çevirimiçi ek yüktür. Hata tahmin yöntemleri sahadaki yazılımları hedef aldığından dolayı, yüksek ek yük harcamaları genelde kabul edilebilir değildir. Bunun sonucu olarak, var olan hata tahmin yöntemleri ana olarak yazılım yaşlanmasından kaynaklana hataları hedef almaktadır.

Bu tezde, içsel telemetri verisi kullanarak hafif yüklü çevirimiçi hata tahmini yapan bir yöntem sunuyoruz. Buna ek olarak, bir seri geniş kapsamlı deneyle bu yöntemin değerlendirmesini sunuyoruz. Bu deneylerde üç adet yaygın kullanımlı yazılım kobay uygulama olarak kullanılmıştır. Deneylerden elde edilen sonuçlar önerilen yöntemin yazılım hatalarını makul maliyetlerde ve güvenilir bir şekilde tahmin edebileceğini belirgin bir şekilde göstermiştir.

to my beloved mother

Acknowledgements

I wish to express my sincere gratitude to my supervisor Cemal Yılmaz for his invaluable guidance, support and patience all through my work. Also I am grateful to my thesis committee members Albert Levi, Erkay Savaş, A. Berrin Yanıkoğlu, and Hakan Erdoğan for their valuable review and useful remarks. It was a privilige to work with such valuable professors.

I am indebted to thank all my friends from FENS 2001; Duygu Karaoğlan, Erman Pattuk, Cengiz Örencik, Barış Altop, Leyli Javid Khayati, Yarkın Doröz, Kübra Kalkan, İsmail Fatih Yıldırım, Ahmet Onur Durahim, Emre Kaplan and İsmet Özalp. I also thank to my dear friends, Berfin Dinler, Elif Damla Akbulut, Tuğçe Yazıcıgil, Oya Çitçi, Gülçin Coşkun, İncinur Temizer, İbrahim Boylu and with his last minute entrance İsmail Kuru. With all my heart I thank to Erdal Mutlu for his endless support, patience and efforts to keeping my spirits up whenever I am downhearted.

I owe my most sincere thanks to my family, especially my mother, for their unlimited support and love.

I also thankfully acknowledge the financial support provided by the Scientific and Technological Research Council of Turkey (TÜBİTAK).

Contents

List of Figures

List of Tables

1 Introduction

Software systems are getting increasingly complex and bigger in size. When these general trends are coupled with shortcomings of software quality assurance techniques and time-to-market pressures, development houses are forced to release their software with many known and unknown defects, which inevitably cause failures in the field. Software failures are proved to be costly in both quantitative and qualitative terms.

One way to prevent software failures in the field, thus to increase software reliability, is to never release software with any defects. This can be achieved by identifying and fixing all defects before the system is released. Software testing and formal verification methods are the widely-used means that can be used towards achieving this goal. However, limitations of these techniques are well-known and sometimes severe.

We believe that, to further improve software reliability, one should accept that software systems do fail in the field. By following the same realistic line of thought, many approaches have been proposed in the literature to predict the manifestation of software failures at runtime and proactively take preventive measures, such as preventing the failures or decreasing the harmful consequences of failures.

Predicting failures at runtime is an integral part of such proactive-preventive frameworks. Existing runtime failure prediction approaches operate in a similar manner: Behavioral models that abstract program executions are created by leveraging historical data and then failures are predicted by identifying and scoring similarities to these models and/or deviations from them.

One downside of the existing approaches is that they treat software systems as a black-box and leverage only the profiling data which are directly observable from outside the programs, such as, CPU, memory, and network utilizations and the number of active processes in the system. No internal execution data is collected. Therefore, these approaches mainly target at predicting failures caused by software aging.

The reason behind why the execution data that could be collected from inside the programs has not been leveraged in predicting failures is, since the failure predictions are performed at runtime (while the programs are running) in production environments, obtrusive nature of collecting internal execution data has believed to be unaffordable.

In this thesis, we, however, empirically demonstrate that certain types of internal execution data can be collected from inside program executions at an affordable cost and these data can help predict failures at runtime.

1.1 Organization of the Thesis

The rest of this thesis is organized as follows; in Section 2, we give background information for thesis, Section 3 describes the related studies in the literature, our motivation is given in Section 4, Proposed approach is given in detail on Section 5 and empirical studies conducted on this approach are explained with their results in Section 6. Section 7 describes threats to validity of this thesis. In section 8 we give a conclusion and discuss the probable future studies in Section 9.

2 Background Information

In this section, some background information is given on tools and methods used in this study. For some experiments, we leveraged hardware performance counters for a way of collecting inner execution data. Then, these execution data is modeled by using classification trees. Our subject applications are gathered from SIR Repository and we used `cilly` tool for instrumentation purposes.

2.1 Hardware Performance Counters

Originally, hardware performance counters were used by CPU producers for performance analysis of their product. They were used for hot spot analysis. Later on, these counters are released to common usage, but they are traditionally used in hardware performance analysis. We on the other hand, use them for software based purposes.

Hardware performance counters can be used to record various events that are occurring on a processor. These counters are part of the hardware architecture and general purpose computers that are used nowadays involve such counters. As said before, hardware performance counters can record several events occurred in a processor, such as the number of instructions executed, the number of branches taken, the number of cache hits and misses experienced, etc.

By default, these counters are inactive and activation methods can differ from processor to processor, but in principle procedure to be followed is quite the same. To activate hardware performance counters, the event of interest has to be linked with the physical counter to be read.

Generally there are a fair amount of hardware performance counters resident in a CPU, utilizing this property, several events can be counted at once by assigning each event to a different counter. In addition to that, by using some additional software, with multiplexing method, more events than counters can be read at once.

When a counter is activated, it starts to count the assigned event and stores the count into a set of special purpose registers. These values can then be read and reset on demand.

2.1.1 Programming Hardware Performance Counters

Hardware performance counters are activated and read by processor commands. There are two instructions are available for reading them;

rdpmc reads the performance counters value.

rmtsc reads the time stamp counter value from a special time counter.

One challenge hardware performance counters introduce is that these counters are low level and cannot distinguish the issuers of the instructions. To monitor programs we have to be able to count the events they issued, so we need this integration.

Also, the registers that store the count values reside in kernel space of the CPU, and the applications users run and need to be monitored will reside in the user space of the CPU. Which means, to access the values stored in the counters, context switches will be required. As known, context switches create additional time overhead, since the reason we leveraged hardware performance counters at the first place is to reduce the time cost of monitoring, this introduced a step-back.

To overcome these problems; we used `perfctr` -a kernel level driver- which maps the counter registers in the kernel level to virtual counters in the user space so that the counter value can be read without the context switch. Also, these virtual counters can be integrated with processes to count events from a single process.

Furthermore, hardware performance counters are independent from the program context, which means they just count the events issued. Collecting inner execution data from executions require integration of counter values with program contexts. To handle this, we used software instrumentation.

Following the programing hardware performance counters, a method is needed to create models from the collected execution data, for this purposes classification trees were used.

2.2 Classification Trees

Classification is an important data mining problem which is in general the act of deciding the class of an observed event. In a classification problem, a dataset called the training set is taken as input. Training set includes several examples each having a number of attributes. These attributes can be continuous or categorical [2]. One example of categorical attribute is class label attribute which is assigning a label to a particular data for deciding which class it will fall into. Deciding if a test will fail or pass is also such a classification problem.

When a classification algorithm inputs a training set, the aim is to build a model of the class label using the other attributes of the training data set. Then, this created model can be used to classify previously unseen data points. In general, the input set is divided into two subsets beforehand; training set and test set. Training set is used to train a classification model, and test set is used to evaluate and fine-tune the model.

When dividing the input data set into train and test sets,the ratio on the number of class labels is preserved. Preserving the ratio of the number of labels while dividing the sets is called stratified sampling [3].

When a training set includes noise or random error, there is a chance that the model trained from this data describe the noise in the model, too. This situation is called overfitting and causes models to fail predicting the classes of unseen data because the model is adjusted the rare random or noisy situations [4]. To avoid overfitting n-fold cross-validation technique can be used. In this method, input data set is randomly -but stratified-divided into n subsets. In each round of cross-validation; one of the subsets is selected as test set, and other n-1 subsets are used to create a model. In the same manner n rounds are performed where each subset is used once as test set. After k rounds are finished, either the best model is used -this is the general approach- or average of the models is used as the resulting model. In either way, overfitting to any training set will be prevented [4].

In classification tree algorithms, the resulting model is a decision tree whose nodes consist of logical conditions on attributes. In this tree each leaf has a class label, thus each route from the root node to a leaf node corresponds to a rule for that class label which consists of only `and gates` (a conjuction). Therefore this tree model is disjunctions of each rule from root to leaves, which is a disjunction of conjunctions.

To illustrate the tree model, an example data and resulting decision tree is given in Table 1 and Figure 1. In the Table 1, there are 4 attributes and 2 classes. This data is simply plotting if the weather conditions are suitable for a kid to play outside. As seen, attributes can be numerical or binary valued. This training data can lead to several decision trees, one of the trees is shown in Figure 1. In this tree, nodes indicate the test to be apllied on the data point, and arrows indicate the possible results of the test of the node. For instance, at the root node, outcast attribute is tested, according to possible outcomes of the attribute ("sunny", "overcast", "rain"), there are three arrows leading to three nodes in the tree. When an unseen data point needed to be classified, this tree is used to give a label to the data point. To illustrate, an example data point {"sunny","70","70","true"} can be traced in the tree in Figure 1. In the node, overcast attribute is used for testing, our data point has "sunny" as the attribute value, so it goes to the leftmost child of the node. At this node, humidity is used, if humidity is smaller or equal to 75 the data point is labeled as "Play", otherwise it is labeled as "Don't Play". Since example data point's humidity value is "70", left child is selected and data point is labeled as "Play". As in this illustration, some attributes may not be checked while deciding the label of a data point. Even sometimes some attributes may not be included in the decision tree.

Since methods are found for collecting the execution data and creating behavioral models from them, subject applications need to be found for experiments.

Outlook	Temp(F)	Humidity(%)	Windy?	Class
sunny	75	70	true	Play
sunny	80	90	true	Dont't Play
sunny	85	85	false	Dont't Play
sunny	72	95	false	Dont't Play
sunny	69	70	false	Play
overcast	72	90	true	Play
overcast	83	78	false	Play
overcast	64	65	true	Play
overcast	81	75	false	Play
rain	71	80	true	Don't Play
rain	65	70	true	Don't Play
rain	75	80	false	Play
rain	68	80	false	Play
rain	70	96	false	Play

Table 1: Sample Training Set [1]

2.3 SIR Repository

An experimental study's success lies on the reliability of the results, and results' reliability highly depends on the subject applications used in the experiments. Being used in similar studies and representativeness of the subject application is important in the sense of generalizing the results of the experiment.

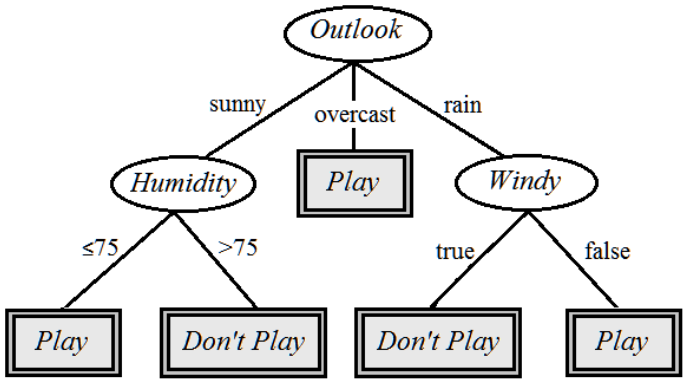


Figure 1: Decision Tree of Table 1

In addition, this study has other expectations from subject applications. First of all, these subject applications should include some defects which will cause failures. These defects has to be known and easily locatable for examining the results of failure prediction method. Also; this subject applications should have test suites that can include both failing and passing tests in it.

In the search of subject applications, under the light of these expectations we decided to leverage SIR repository. SIR (Software-artifact Infrastructure Repository) is a bug repository that provides users several software systems, in multiple versions with their known defects and test suites. Each one of the set defects that SIR repository provided for applications is identified with a unique defect identifier, and these defects could be activated individually as needed.

2.4 Cilly

Since automating the instrumentation phase is quite important in terms of usability of the method, several options are visited for this purpose such as `gcc` function handlers or wrappers. However the most beneficial tool for our needs was CIL.

CIL (C Intermediate Language) is a high-level C analysis and transformation tool [5]. CIL, creates an intermediate source code from C source codes by removing ambiguities and redundant constructs while maintaining types and a close relationship with the source program [6]. CIL makes a source-to-source transformation and create a representation of the original code which is easy to manipulate and analyze [6].

To create CIL output, `cilly` script is used. `cilly` is a perl script which works like a C compiler, can be used with existing Makefiles and compiler options. In addition, cilly has its own options, so that users can turn on CIL's options. We utilized CIL with several options, each of these options are explained in brief detail below.

`--dooneRet:` This option, provides every function has at most one return statement. In the functions with several return statements, uses jumps to one return point that it added to the end of the function.

`--dologcalls:` This option, inserts print commands to the code, so that it will print the name of the functions when they are called.

`--save-temps:` This option is used to preserve the temporary files created.

Integration of monitoring tools and program context done by leveraging cilly tool to insert monitoring and prediction codes into the software.

3 Related Works

In this work, we combined two different research areas for purpose of failure prediction. As a result of this interdisciplinary structure of the work, the related works are divided in to two sub sections; related works on hardware performance counters and related work on failure prediction.

3.1 Related Works on Hardware Performance Counters

Hardware performance counters are mostly used for performance analysis and finding the performance problems of software systems. These kind of studies are categorized as “hot spot” analysis. Primary purpose of hot spot analysis is to find components causing bottleneck in the system. These information is than used to improve performance of these components in order to improve overall performance.

Hardware performance counters are used in different hot spot analysis like; performance analysis of standalone programs [7, 8], performance analysis of distributed and parallel software systems [9, 10, 11], and dynamic improvement of system performance [12, 13, 14]. It is easy to increase the number of such studies.

As hardware performance counters are proved to be successful for performance analysis, to improve the usability of such counters different tools and libraries, both open source [15, 16].

3.2 Related Works on Failure Prediction

Today, encountering failures of software system in the field is very common [17, 18]. There are two main methods for minimizing the cost caused by the software failures and for increasing software reliability by eliminating these failures. These methods are classified as proactive and preventive methods. Proactive methods aim to recover from failures in software systems effectively and quickly by simply returning to the state before the failure [19, 20]. We can give operating systems' automatic restart mechanism as an example to proactive methods. The important thing that shouldn't be missed is that these methods become active after failure. On the other hand preventive methods aims to predict and prevent failures before they occur. In this spectrum, preventive methods have better potential on increasing software reliablity [21]. In this work, we designed a preventive method for failure prediction.

All the previous studies conducted in this area uses black box approach for collecting data from software. Which means these studies collect the data only can be observed from outside of the executions, such as memory usage, network usage, number of processes working, error logs and response time.

In [21], Lin used creation times of the failure log files for showing that failure prediction beforehand is possible and developed a method called "Dispersion Frame" for this purpose. The method uses the statistical difference between error distribution reports that are created before failure occurs and doesn't occur.

Likewise in [22, 23], Vilalta again used failure log files but rather than look for the creation times, he used fault types for failure prediction with

"Event Set" technique. Vilalta's studies showed us that there are statistical differences not only on the number of the files created but also in the types of the failure log files. He also studied algorithms to be used for failure prediction and their success on doing so [24].

In [25], Salfner used both failure log file creation time and fault types for developing "Similar Events Prediction" method. Later, he proposed another method, which uses "Hidden Markov Models" in order to lower the cost of behavioral model creation, for the same purpose [26]. These studies show that using both fault type information and fault occurance time gives better results than using them separately.

4 Motivation

In today's world software systems are not only a subject of the computer engineering, but also it resides in almost every aspect of daily life. It is almost inevitable to have defects in software systems since user behavior may not be known and tested completely. Thus, software systems are released to users with bugs and they lead to failures. These failures are costly in both quantitative and qualitative terms. Under these circumstances to increase the reliability of the software in the field; proactive-preventive methods are proposed and used in the literature. These methods aim to predict the failures beforehand and try to minimize the harmful consequences of failures.

To predict failures in programs, profiling data is collected from the executions of the software and environmental elements. Then, this execution data is used to create behavioral models of the software. Thereafter, when software is being used by end users, dynamically collected execution data is compared to this created model and similarities and/or differences from the model is scored. In case of any of these similarities or differences are considered as ‘suspicious’, prediction is made about an upcoming failure and preventive mechanisms step in.

In this method, the important part to focus on is the type and the quality of the execution data, because the accuracy of the behavioral model depends on the execution data. Typically the more detailed and inclusive the execution data is, the more reliable the models are. However, collecting more data means more overhead both in term of time and space. Since this approach is targeted at software systems running in the field, keeping the overhead low is an important factor. At this point, a trade off problem comes into the picture. If collected data is too extensive, overhead is expected to be too high; otherwise overhead will be kept at a low level but this time created behavioral model may not be reliable. So, one of the main limitations in this area is finding a low cost monitoring method. The main drawback of the previously proposed methods is this trade-off between data efficiency versus overhead efficiency.

First of all, most of the proposed methods in the literature apply failure prediction on the level of systems rather than the level of programs. I.e. failure prediction mechanism on a server that provides hosting for an online application, uses the data that are related to the server machine such as memory, network and disc usage, etc. These approaches have no awareness of the software that is running on the server machine. Since failures generally originated from software, it is expected that being able to look into the software will produce a higher success rate on failure prediction.

Second, the techniques which are collecting execution data from the software, generally collect data that can be observed from outside of the software such as latency and throughput. These methods do not gather any information about the inner state of the application. Hence, these methods can make predictions on the level of software systems but the models created and predictions made are limited to the data that can only be observed from outside of the software. This limitation makes these approach perform poorly in failure prediction.

Collecting inner execution data is not a topic that has not been visited before, but the cost of collecting the data would be too high to make the method feasible. The most important motivation of this study is the hypothesis that inner execution data can be collected with a minimal cost and this data collected can be used to create reliable models of the executions.

5 Approach

In this study, our aim is to dynamically predict failures at runtime while the program is running. The proposed method requires dynamic data collection in the field, so monitoring cost has to be as low as possible. To overcome this, monitoring phase is pushed onto the hardware as much as possible. For that purpose, we leverage hardware performance counters to collect hardware level monitoring data. Since hardware performance counters do not have information about the issuer process of the event they count, we used virtual counters to make hardware performance counters collect data from a single process.

To conduct the study, we created a two step approach. First step consists of using execution data to create behavioral models of executions and creating a prediction model from them. Second step covers instrumenting software to be deployed in the field for failure prediction.

In Figure 2 overall structure of the proposed method can be viewed. The rest of this section will explain each step of this flow in detail in two main subsections; Training Phase (5.1) and Deployment Phase (5.2).

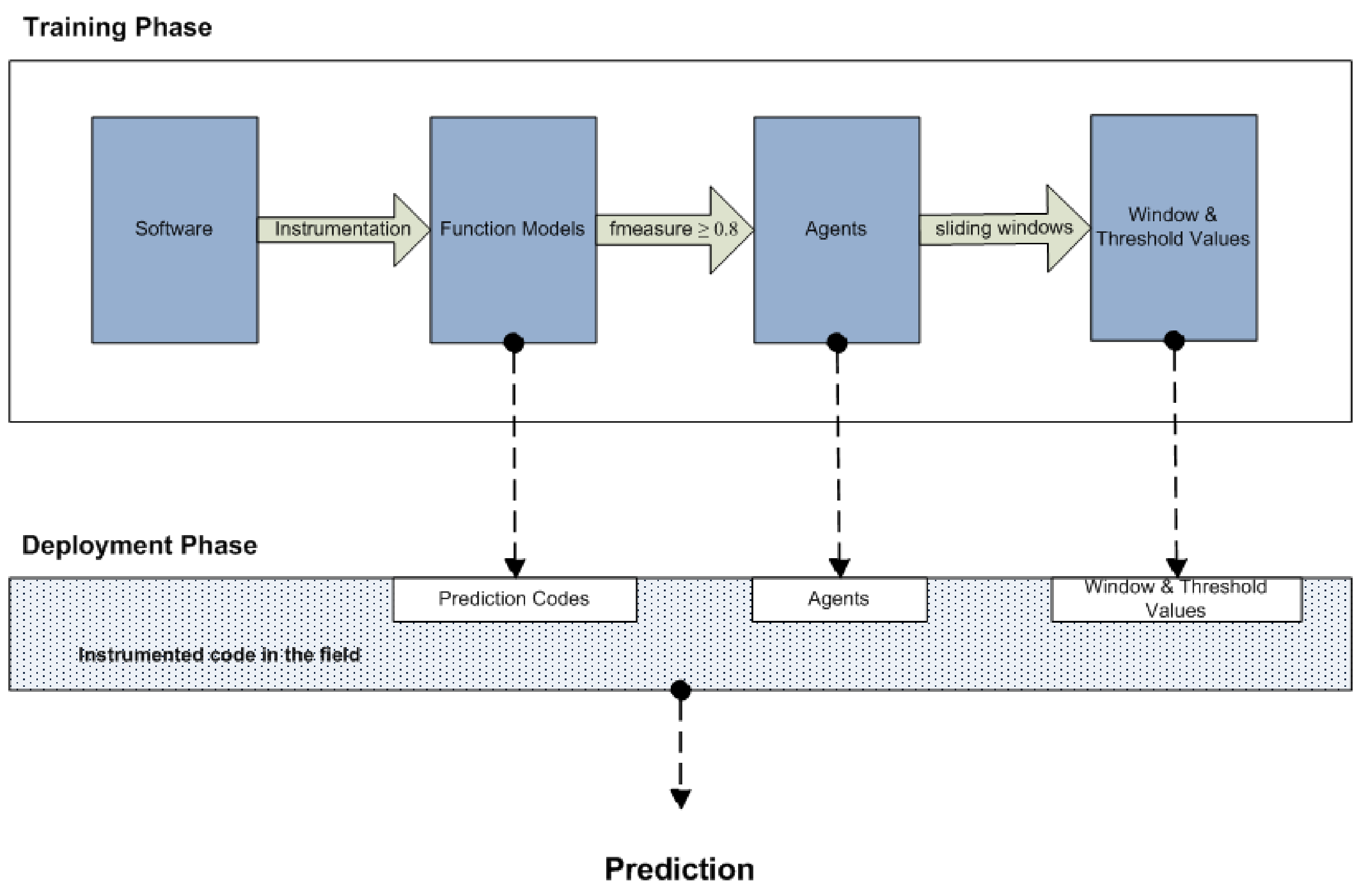


Figure 2: Flowchart of the Proposed Method

5.1 Training Phase

This section describes the steps of training phase of the approach in detail.

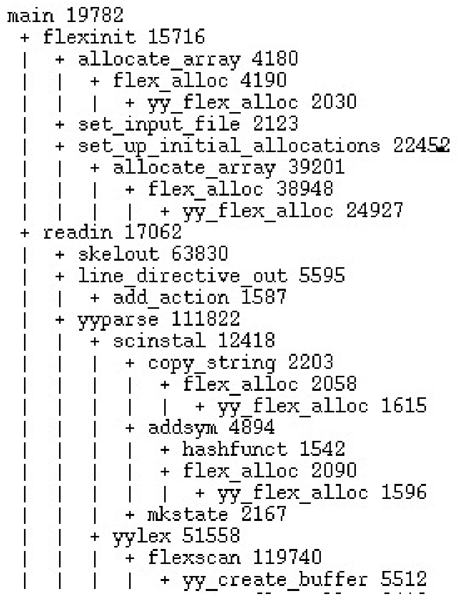
5.1.1 Instrumentation

For percieving the relationship between monitoring event and the source code or its context, we need instrumentation of the source code to collect execution data. When connecting monitoring tools to software contexts, it can be done in several levels. The instrumentation can be on the level of systems, sub-systems, components, functions, and code blocks.

Each level of integration to software results in a different program profiling data. In this study, as monitoring level, we selected function level monitoring. Each function in the execution carry out a subtask in the execution. Therefore function level monitoring will allow us to monitor the tasks carried out in the execution on the basis of subtasks and they have well defined boundaries in the program execution.

To create the integration between hardware performance counters and functions, a simple procedure is followed.

1. At the beginning of the execution, hardware performance counter is activated with the desired event to count.
2. Counter value is read at the beginning and at the end of each function. The difference between these two values are associated with the invocation.
3. At the end of the execution, hardware performance counter is deactivated.



```
main 19782
 + flexinit 15716
 |  + allocate_array 4180
 |  |  + flex_alloc 4190
 |  |  |  + yy_flex_alloc 2030
 |  + set_input_file 2123
 |  + set_up_initial_allocations 22452
 |  |  + allocate_array 39201
 |  |  |  + flex_alloc 38948
 |  |  |  |  + yy_flex_alloc 24927
 + readin 17062
 |  + skelout 63830
 |  + line_directive_out 5595
 |  |  + add_action 1587
 |  + yyparse 111822
 |  |  + scinstal 12418
 |  |  |  + copy_string 2203
 |  |  |  |  + flex_alloc 2058
 |  |  |  |  |  + yy_flex_alloc 1615
 |  |  |  + addsym 4894
 |  |  |  |  + hashfunct 1542
 |  |  |  |  + flex_alloc 2090
 |  |  |  |  |  + yy_flex_alloc 1596
 |  |  |  + mkstate 2167
 |  |  + yylex 51558
 |  |  |  + flexscan 119740
 |  |  |  |  + yy_create_buffer 5512
```

Figure 3: Calltree Example

Once the execution data is collected from an execution, it is used to create an annotated calltree. In this calltree, each function invocation associated with the number of hardware events monitored during the invocation. A calltree example can be seen in Figure 3. In this calltree, each line corresponds to a function invocation and indentation on each line depicts the depth of the invocation. At each line, a function invocation is proceded with the number of events counted. Additionally, using this tree, caller - callee relations can be seen easily.

In our study, functions are modeled on the basis of their callee functions. Each sub-function carry out a functionality of its callee function, therefore, to itemize the counter values by sub-functions gives us information about the effort made for each functionality. The suspicious changes in the effort made to complete a functionality could point out that something went wrong in the execution [27].

5.1.2 Modeling Function Behaviors

After creating annotated call trees, as a preliminary step to creating behavior models, call tables created for each function by using all call trees of a program. A function table, stores information of all the invocations of that function within all tests. Each table stores the number of events monitored, on the basis of the body and the callee functions of that function. Each function invocation is represented with a separate line in the table.

test	depth	body	f10	f24	f41	f109	f128	pass/fail
71	1	2401	2600	7632	-1	63831	85284	P
443	1	1846	2605	7631	-1	35000	74528	F
206	1	1876	2600	7632	-1	63830	40000	P
206	3	1849	2610	7623	-1	63830	77928	F

Table 2: Function Table Example

Table 2 presents an example function table for function `foo`. In the table, first column gives the test and second column gives the depth at the call tree where the data of the row is taken. Third column gives the number of monitored events inside the body of the function except the events monitored inside callee functions. The columns after third one, shows the callee functions and the number of events monitored inside these functions. Therefore the columns after third column may vary for every function table.

The last column of the table depicts if the test was passed or failed; P stands for passing and F stands for failing. If a value is '-1' in a cell, it means that the callee function was not called within that invocation.

Our aim is to create models for functions so that these models can be used for classifying unseen runs as failing or passing. For this purpose, we leverage classification trees.

After the function tables are created, for each function we run a classification tree algorithm. Tables of each function are fed to the algorithm, to produce a decision tree for each function. When tables are given as input, test name and depth columns are eliminated. 10 fold cross validation was used for fine-tuning the results.

By creating a decision tree for each function, we obtained a behavioral model for each function. The resulting tree can predict if an execution is going to fail by using the monitoring events inside the callee and body of the function. A possible classification tree created from Table 2 is shown in Figure 4.

Once models are created, they can easily be converted to a `C` code, so that it can be used in instrumentation. Each node in the tree corresponds to an `if` statement, each child of a node is either corresponds to its `then` or `else` part. Therefore, a decision tree can be represented as an `if - then - else` statement. From now on this code is referred as *prediction code.* An example prediction code of Figure 4 can be viewed in Example 1.

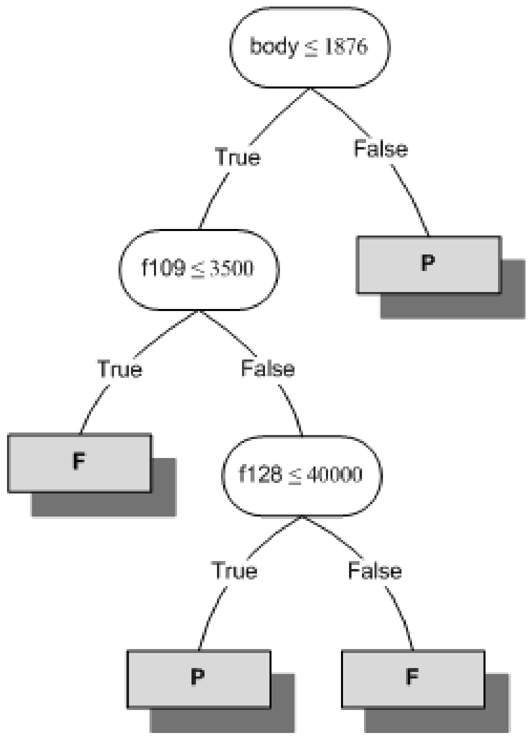


Figure 4: Decision Tree Obtained From Table 2

Example 1 Example prediction code

```
if (body <= 1876)

{

  if ( f109 <= 3500)

  {

    //Failing;

  }

  else

  {

    if(f128 <= 40000)

    {

      //Passing;

    }

    else

    {

      //Failing;

    }

  }

}

else

{

  //Passing;

}
```

5.1.3 Identifying Agents

When behavioral models of the functions are evaluated with test sets, we saw that not all models are reliable, i.e., some functions are more successful in distinguishing failing executions from successful executions, and some functions are in no way related to the failures in the execution, so they could not distinguish between failing and successful executions.

In the light of this observation and similar observations made in previous works, instead of monitoring every function in the execution, we monitored only the ones that are successful in distinguishing failed runs from successful runs. We conjecture that this approach can increase the success rate on failure prediction and decrease the overhead cost of the method. After this point, these functions are going to be referred as *agent functions*.

To find the functions that are successful in predicting failing executions, we calculated the success of each function model. For this calculation, we used F-measure parameter. F-measure parameter combines precision and recall to give a score to the performance of the model. Precision calculates the fraction of instances that are failing in the instances that are labeled as failing by the model and recall calculates the fraction of instances that are failing and correctly labeled. Using a threshold value of 0.8, agent functions are determined. Figure 5 illustrates the agent selection process.

After filtering the functions according to their accuracy in failure prediction, some statistics about these agent functions are given in Table 3. As explained before, each software under test (*sut*) in this study have several versions and for each version there are several defects identified. Each time a defect activated, it will become a different program and we find agents for

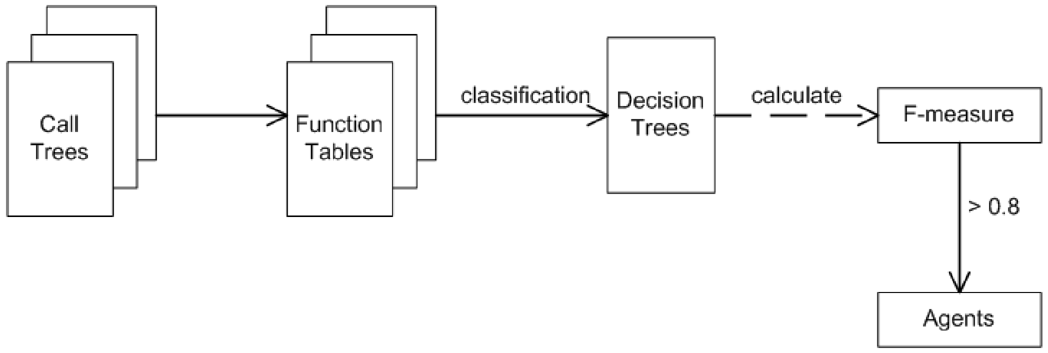


Figure 5: Flowchart of Agent Selection Process

them separately. In the table, the statistical values are averaged over each version.

Two important observations can be done at this point. First one is, there are functions that are capable of predicting failures with a 0.8 or higher accuracy. Second, these functions are generally a small percentage of the all functions in the program. These observations are important and encouraging for the future steps of the study.

sut	version	# of functions	# of agents	agent percentage(%)
`flex`	v1	603	22,2	4
	v2	480	9,8	2
	v3	461	27,8	6
	v4	1033	9	0,9
`grep`	v1	48	34	70
	v2	132	13,5	10
	v3	145	15	10
	v4	72	27	38
`sed`	v2	295	18,5	6
	v3	221	8,75	4
	v5	67	6,3	9
	v6	460	18,75	4
	v7	405	10,6	3

Table 3: Statistics on Agent Functions

5.1.4 Creating Prediction Models

We have so far identified reliable agents. Now, we need to develop a prediction mechanism. Once agents are instrumented with their prediction codes, they will make predictions about the future failures while execution is proceeding. Hence, when an agent is called in the execution it will create a prediction 'F' (failing) or 'P' (passing) according to its prediction code. Consequently we will have a string consists of 'P's and 'F's, and this string is going to grow longer as the execution proceeds since each time an agent is invoked, it will add a prediction to the string. This evolving sequence of prediction is called *health index*. Using this health sequence, two approaches proposed.

Point Prediction (PP): First approach that comes to mind is to determine that there will be a failure, whenever a 'F' is seen in the sequence. We will call this method *PP* method.

However this method only considers one agents opinion, and just looks for the decision of the agent. Also in general, there is a pattern for fault formation in systems. First a problem occurs in a state, then the current program state is defected. This defected state affects next state and makes next state defected, too. These successive events pile up over time to create a failure. Which means, failure occurs over a time window. Therefore, it is coherent with this fault model to make the prediction of the failure, using the decisions of the agents over a window.

Moreover, PP method does not take into account the accuracy scores of the agents. So another method that can address these needs is developed.

Sliding Windows Prediction (SWP): This method processes the health sequence over a window. At each window it looks for the 'F' values, if there are any, it checks for the accuracy score of the agent that gave the 'F' decision. All the accuracy scores of the 'F' values within the window is added up.

These windows are found with sliding window method, i.e. at every step window is shifted one character right. This process is represented in Figure 6. In this figure, time is passing from left to right, and as new agents are run, the sequence grows in time. For the example, window size is selected 3.

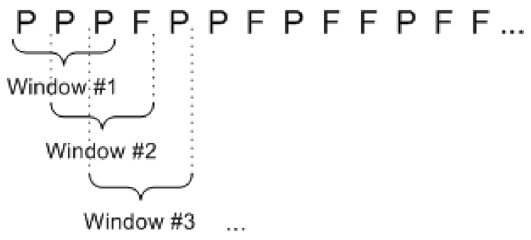


Figure 6: Sliding Window Method

At each step of this approach a sum is calculated for the window, so, this process outputs a string of numbers. For each subject application, these score strings are gathered from each test. Afterwards, these sequences are fed to classification algorithm which will output a threshold value for each tuple. This threshold value indicates that whenever a window's score passed this threshold value, the execution will be faulty.

The size of the sliding window plays an important role on this methods success. To find optimal values for window sizes, for each subject application we tried window sizes changing from 1 - 10. Then F-measure parameters (see Section 6.2 for details) are compared to find the window that generates most accurate results.

At the end of this step, for each software - version - defect tuple

- agents,
- prediction codes for agents,
- window sizes and
- window thresholds

are known. In other words, everything needed for instrumentation are known. Next step is using this information, instrumenting the software that is going to be deployed in field.

5.2 Deployment Phase

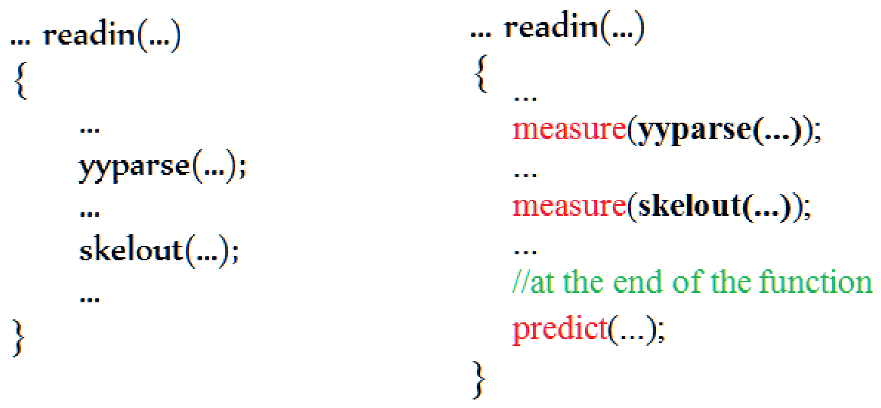
When instrumenting the systems, only agent functions are instrumented. Other functions remain untouched. Within the agent functions, only the sub-functions that are used needed in prediction code are instrumented.

Two types of instrumentation code is inserted to software, monitoring and prediction codes. Monitoring codes count the events inside the body of the agent functions and their callee as needed. Prediction code on the other hand uses the data collected by monitoring codes and gives a prediction about the execution. An example prediction code was given in Example 1.

When inserting the monitoring codes, they are placed at the entrance and exit points of the agent function. Also to count the events occurred in the sub-functions codes inserted into the agent function's body just before and just after the calling of the relevant sub-function. After that, prediction codes inserted at the end of the agent function.

An example of instrumentation over a very simplified function is given in Figure 7. Figure 7(a) and 7(b) present the function before and after instrumentation, respectively.

Thanks to CIL's `dooneret` and `dologcalls` options, each function's entrance and return points became apparent and each function call is underlined with a print statement in a slightly changed source file with extension of `"cil.c"`. Our instrumentation tool takes this cil.c file as input, and uses



```
... readin(...)
{
      ...
      yyparse(...);
      ...
      skelout(...);
      ...
}
```

```
... readin(...)
{
   ...
   measure(yyparse(...));
   ...
   measure(skelout(...));
   ...
   //at the end of the function
   predict(...);
}
```

(a) Original function (b) Instrumented function

Figure 7: Instrumentation Example

this print and return points as the spots to insert the instrumentation codes in the functions needed. Before the instrumentation ends, all of the print statements that CIL inserted are removed since they are unnecessary.

6 Empirical Studies

6.1 Hypothesis

This study depends on two main hypotheses. First one is that that there are repeatable and identifiable patterns in program executions and deviations from these patterns and/or similarities to them are highly correlated with the manifestation of failures. Second one is that program execution data can be collected from inside executions at an affordable cost and the collected data can be used for runtime failure prediction.

6.2 Independent Variables

In the experiments conducted to test these hypotheses several independent variables were present. These are the variables that are manipulated during experiments.

First, type of the execution data (*metric type*) is an important variable for these experiments. This variable determines the event to be monitored and used in the modeling.

Our approach design is generically, for it works for any event that can be counted, but in this study we used six events to collect execution data. These are;

Visits This execution data covers the number of each function visited within the execution.

Path This type of execution data includes the path of the execution on the basis of functions.

Time This execution data simply consists of the time measurements of the executions, functions, etc.

TOT_INS Records the number of machine instructions executed.

BR_TKN Counts the number of branches taken.

LST_INS Records the number of load and store memory instructions executed.

The last four metrics described can be collected using hardware performance counters.

Another variable was the level of integration between performance counters and the source codes of the program. It could be done on the level of functions, program components, subs-systems or systems. In this study we selected function level integration and build our experiments on this basis.

Also, to enhance the prediction or overhead performances, some filters were applied on our training sets. There are three filter options;

no filter: As the name suggests, in this option no filters applied on the data.

global filter: This filter removes the functions that are globally indicates functions. Such as functions that are only visited when a failure occurs. This filter is applied to increase reliability of our results. I.e. selecting these functions as failure prediction agents would not be useful since these functions mean that failure is already occurred.

50 filter: This filter removes the functions that are invoked more than 50 times during an execution. The purpose of this filter is to lower the expected overheads by prevent them to be selected as agents.

6.3 Dependent Variables

In this study, several events were monitored in the basis of functions and calltrees created from each test's execution. With this data, several criteria were calculated.

F-measure We used this criterion for agent selection process. It gives a balanced value between precision and recall values.

False Acceptance Rate (FAR) This criterion is used for performance calculation of decision trees. It gives the probability of falsely accepting instances for a class [28].

False Rejection Rate (FRR) This criterion is again used for performance calculation of decision trees. It gives the probability of falsely rejected instances of a class [28].

Half Error Rate (HER) This criterion used to combine false acceptance rate and false rejection rate in the same calculation. It gives the arithmetic mean of FAR and FRR.

These explained criteria are used to evaluate the performance of the failure prediction mechanism. These measurements are widely used in the literature for similar purposes. For evaluating the method in performance analysis, following criteria are used;

Runtime Overhead of Monitoring The effect of collecting execution data on the program performance is calculated by timing the program without any data collection and with data collection. Then the difference is divided to original timing, which gives the time overhead percentage of the collection.

Runtime Overhead of Prediction In a similar manner, this cost is calculated by timing the program with and without the prediction codes. The ratio of the difference of these two timings to the original timing gives the prediction cost in percentage.

Warning Point This parameter gives the ratio of time it took to make the prediction to the time of the failure. It is calculated in the basis of function invocations.

6.4 Evaluation Framework

When evaluating the experiment results, criteria explained in previous section was used. For evaluating the success of the behavioral function models, half error rate is used. Half error rate is calculated over failure class. Which means failure false acceptance rate and failure false rejection rate is used when calculating the half error rate. From now on, failure half error rate will be referred as half error rate for simplicity. In a similar manner, FAR and FRR are calculated from failure point of view, since our aim is to predict failing executions.

For FAR, FRR and HER, low values are desirable. They get a value between 0 to 1, where 0 means perfect accuracy in prediction. Warning point also takes a value between 0 and 1, 0 indicates that no part of the execution is seen and 1 indicates that all of the execution is seen. So, low values are better for this parameter, since it will provide more time to switch on preventive and protective mechanisms.

For time overhead measurements on the other hand, lower overhead is the better. When gathering the time measurements multiple time measurement techniques visited. Since our subject applications are short running algorithms, we placed an extra emphasis on resolution of time measurement. All the test cases used in the experiments were short-living test cases. Therefore, we paid special attention to the way we compute the overheads. There are four different time measurement techniques we considered;

- Wall Clock Time Measurement (WT): This method is simply measures the real time passed from the start to the end of the execution.
- Virtual Clock Time Measurement (VT): This method calculates the time only spent for the tasks of the process of interest.
- Wall Clock Cycle Counter (WC): This method counts the clock cycles passes from the start to the end of the execution.
- Virtual Clock Cycle Counter (VC): This method counts the clock cycles only when process of interest is working in CPU.

To find the best timer for our experiments, a comparative experiment was conducted. Using a point from our experiment space (`flex v1` with the defect `F_HD_1` activated), time overhead calculations are made with original code in both sides of the comparison. Which means this application's time overhead over itself is calculated. In theory, an application's overhead to itself is 0, so any difference from 0 in the calculations will show us the error of the timing tool.

For the subject application, `flex v1` with defect `F_HD_1`, all tests are run for 50 times each and their average is used. Then same process is repeated, and compared to first time's results. In Figure 8, results of this comparison is given. It is clear from the figure that wall clock timers introduce too much noise into the equation, sometimes as much as 30% time overhead is calculated from noise, so wall clock timers are ruled out.

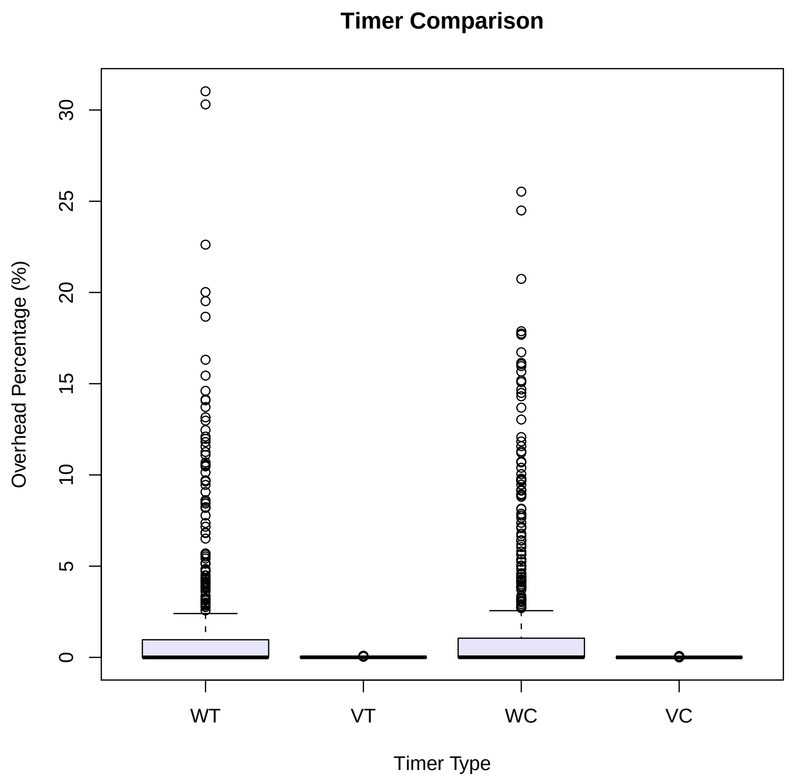


Figure 8: Comparison of Timing Methods

Taking a closer look to just virtual counters, Figure 9, shows that even though slightly, virtual clock cycle counter performs better than virtual timer. Therefore virtual clock cycle counter was selected for our time measurements. All timing and overhead analyses in this study were made using virtual clock cycle counters.

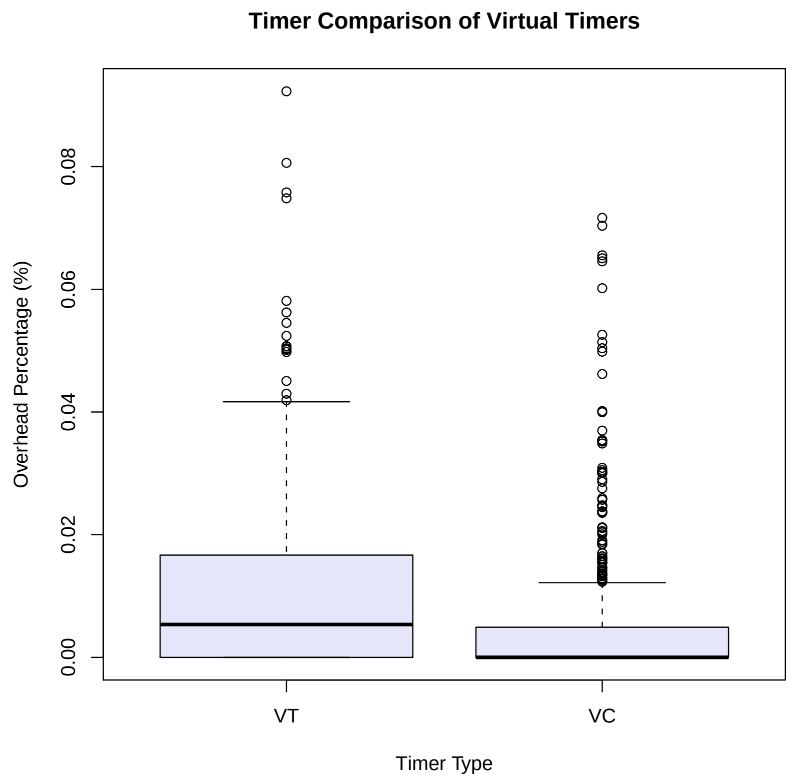


Figure 9: Comparison of Virtual Timing Methods

6.5 Subject Applications

For our research; three open source applications selected from SIR repository as our subject applications; flex, grep, and sed. These applications are all widely used UNIX/Linux based applications.

`flex`: This UNIX based application is a lexical scanner which used to generate fast lexical analyzers.

grep: This UNIX based application is a command line text search utility which prints lines matching a pattern or regular expression.

sed: This UNIX based application is a stream editor which filters and transforms texts.

As explained before, SIR provides defects and test suites for each version of each application. Each subject application selected has its own test suite and test oracles. Table 4 includes some statistics about the subject applications used in this study. In the table, number of defects and number of tests columns give the aggregated numbers over all versions.

software	lines of code	# of versions	# of defects	# of tests
`flex`	10459	5	52	3037
`grep`	10068	4	20	2440
`sed`	14427	6	26	2367

Table 4: Subject Applications' Statistics

6.6 Operational Model

Each application in SIR, comes with a base version and defined faults. Base version is the fault-free original version and faults can be inserted into base versions to create new versions. In this study, versions created from subject applications by inserting one fault at a time.

In Section 5 steps of the method was explained. For finding all values that needed for instrumentation, two phases of experiments conducted. In first phase, reliable functions in failure prediction (agents) are found, in second phase reliable prediction models are found for agents (sliding window size and threshold). Since the experiments are two-fold, experiment space is divided into two phases, too. Furthermore, each phase is divided into training and testing sets for evaluations. Figure 10 shows this division and the number of test cases fall into each phase set.

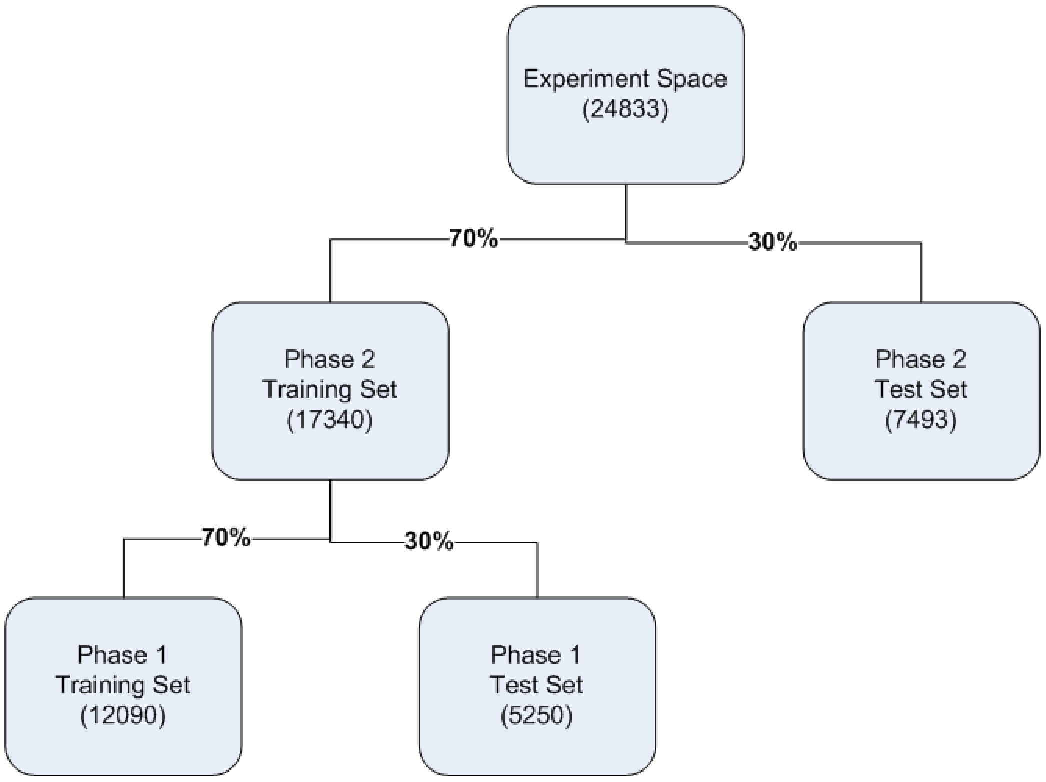


Figure 10: Experiment Space Divided into Two Phases

Function models are created using Phase 1 training set and evaluated on Phase 1 test set. The results of the evaluation is used to select the agent functions. Afterwards, for agent functions, PP (Point Prediction) and SWP

(Sliding Window Prediction) were applied on Phase 2 training set. Then evaluation of these methods are done by using Phase 2 test set.

Furthermore, all these steps are realized automatically by the framework developed. This framework inputs the subject application and values for independent variables and carry out every step from creating behavioral models to instrumenting the software to be deployed and calculating the evaluation metrics. By doing so, human factors in the experiments are minimized so that they will be more reliable and fast.

This study was carried out using CIL's 1.3.7 version, and papi library's 3.6.2 version. All experiments conducted on a Pentium D machine with 1 GB of RAM, running on the CentOS 5.2 operating system.

6.7 Data and Analysis

In this section, we present the results of the experiments. To conduct the experiments, all tests in the test suites for subject applications run, collected execution data used for modeling function behaviors, agent functions selected and prediction codes created. Prediction codes generates predictions while execution is continuing. These generated predictions form health index. Using the health index failure prediction is made.

Using the health index, two methods of failure prediction was proposed (see section 5.1.4). These methods' accuracy results and their comparative analysis is made mainly on HER and failure warning time.

The results are represented in box&whisker plots for all six profiling metrics (`BR_TKN, LST_INS, TOT_INS`, path, time, visits) in four separate plots (`flex, grep, sed`, All Suts) for each method.

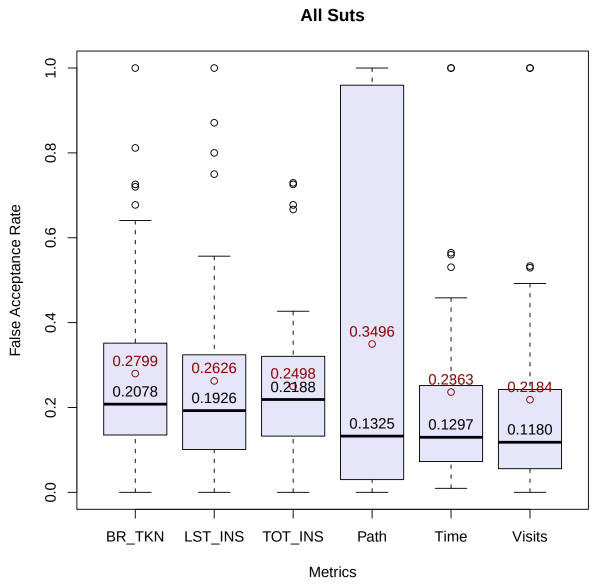
Initially, PP method's accuracy is tested on all subject applications without any filtering and results are presented in Figures 11, 12, and 13. If we look at Figure 13(a) to evaluate the method over all subject applications, we can see that this approach has a half error rate around 12%, which is promising.

To compare the two methods we proposed, SWP method's accuracy on all subject applications without any filtering is shown in Figure 14, 15, and 16. The results in this figure implies that SWP performs better or equal than PP method. Also hardware performance counter based metrics benefit more from this method. The results show that, the error rate of methods `TOT_INS, LST_INS, and BR_TKN` are down to 4%'s.

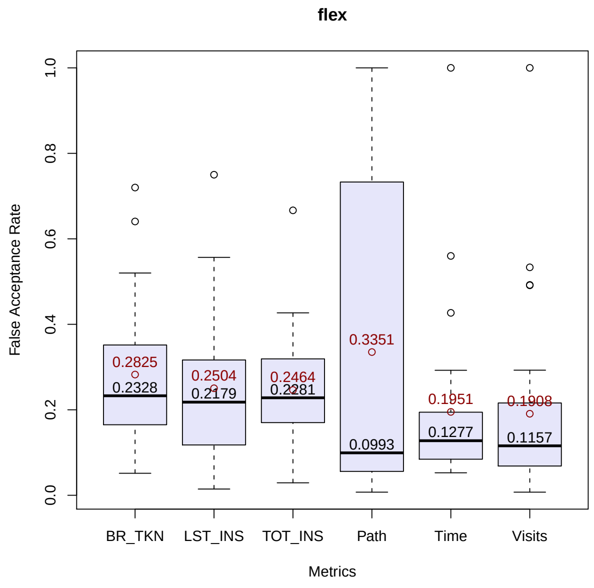
Therefore, we moved on analyzing overhead cost of SWP method. Figure 17 shows the overhead costs of each metric performed on SWP method. As promising the failure predictions as, the overhead costs we observe were up to 28%, this overhead is unaffordable for an online application.

As mentioned in Section 6.2, for satisfactory results and in the purpose of having a lower threshold, filter are applied to the data (global filter and 50 filter). In 50 filter the functions that are visited more than 50 times, excluded from the experiments since they may introduce high overhead results. Although this exclusion may lead to lower prediction success, it is expected to create lower overhead costs, and this trade-off may be preferable.

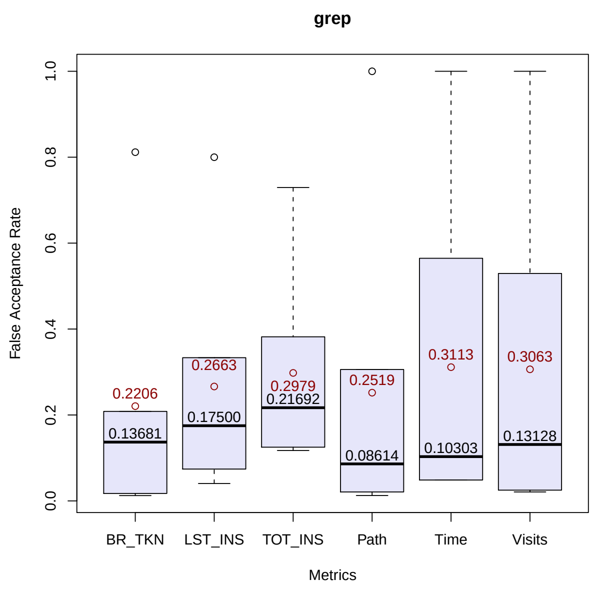
Figures 18(a) and 18(b) illustrates the HER values of the methods under both filters. When we compare the results in Figure 13 and 16 to the results in Figure 18, especially in PP results, we see that some metrics performed better and some performed worse under filtering. We conclude from that the


All Suts
False Acceptance Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.2799
0.2078
0.2626
0.1926
0.2498
0.2188
0.3496
0.1325
0.2363
0.1297
0.2184
0.1180
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

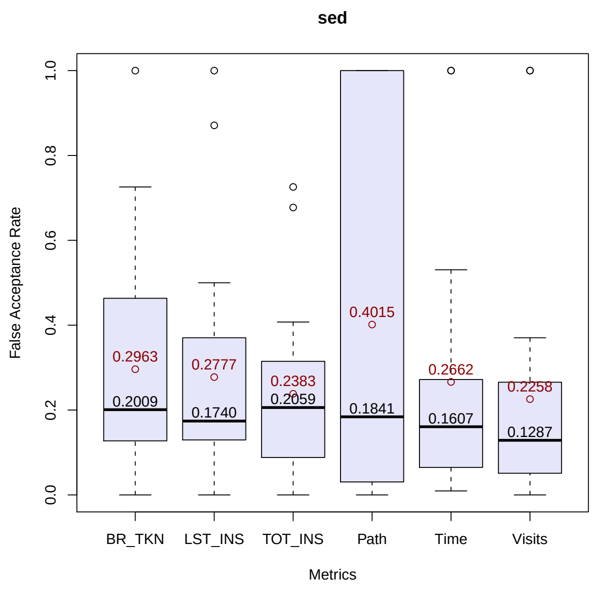
(a)


flex
False Acceptance Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.2825
0.2328
0.2504
0.2179
0.2464
0.2281
0.3351
0.0993
0.1951
0.1277
0.1908
0.1157
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

(b)

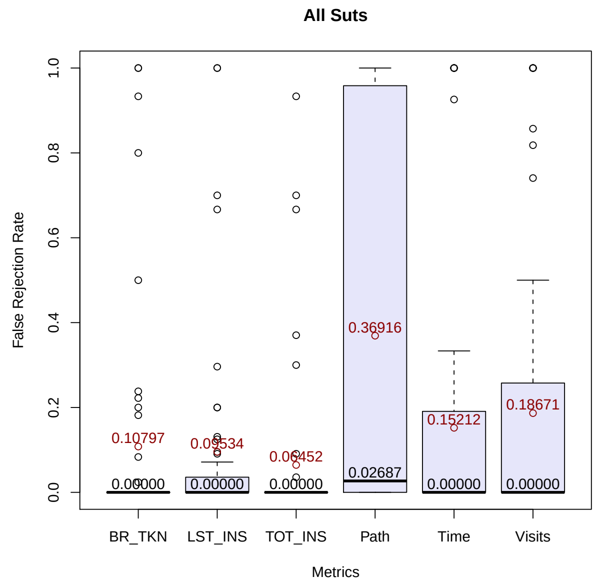


(c)

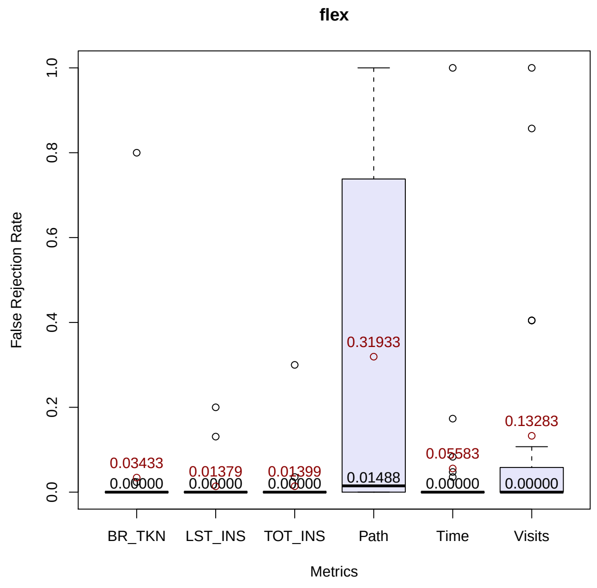


(d)

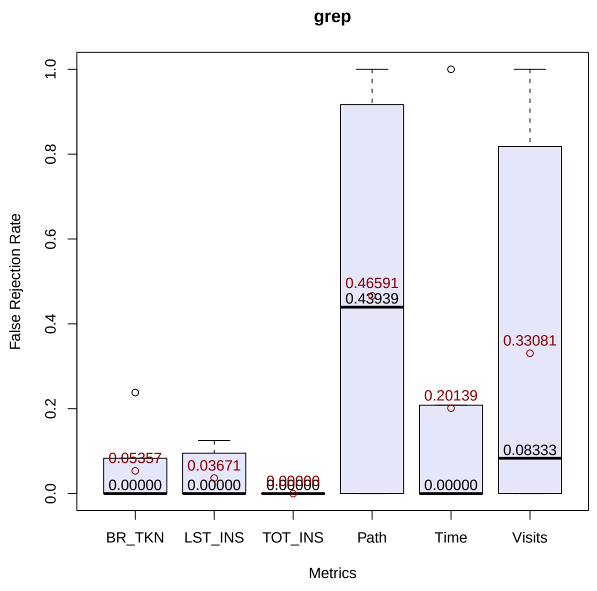
Figure 11: FAR Analysis of PP Method


All Suts
False Rejection Rate
Metrics
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
0.10797
0.09534
0.06452
0.36916
0.15212
0.18671
0.00000
0.00000
0.00000
0.02687
0.00000
0.00000

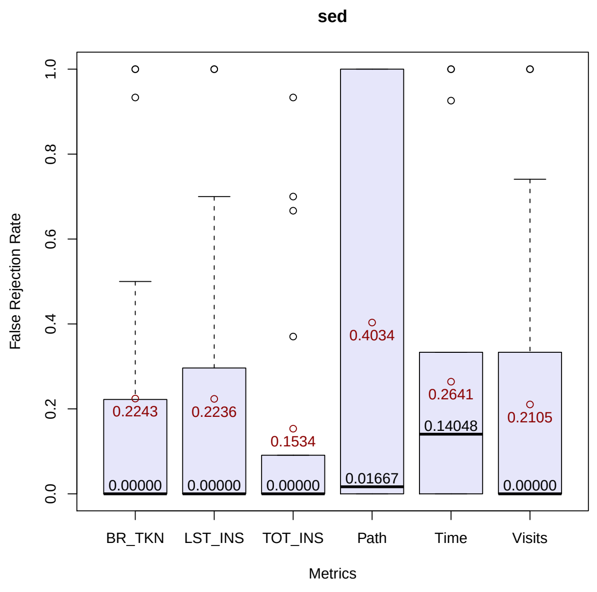
(a)


flex
False Rejection Rate
Metrics
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
0.03433
0.01379
0.01399
0.31933
0.05583
0.13283
0.00000
0.00000
0.00000
0.01488
0.00000
0.00000

(b)

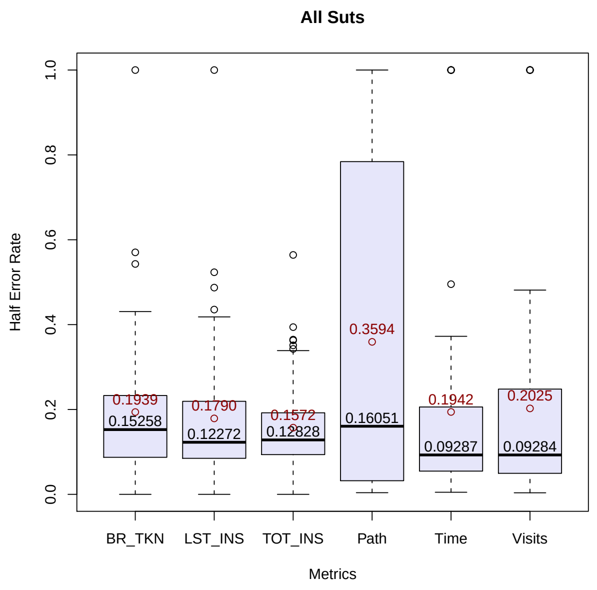


(c)

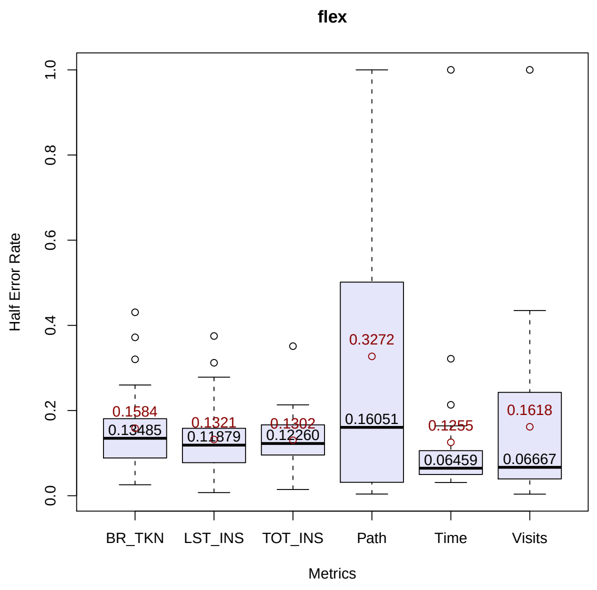


(d)

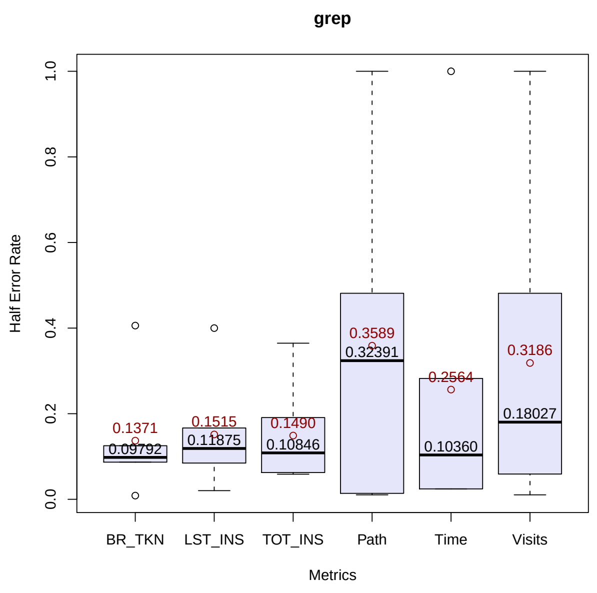
Figure 12: FRR Analysis of PP Method


All Suts
Half Error Rate
Metrics
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
0.0
0.2
0.4
0.6
0.8
1.0
0.1939
0.15258
0.1790
0.12272
0.1572
0.12828
0.3594
0.16051
0.1942
0.09287
0.2025
0.09284

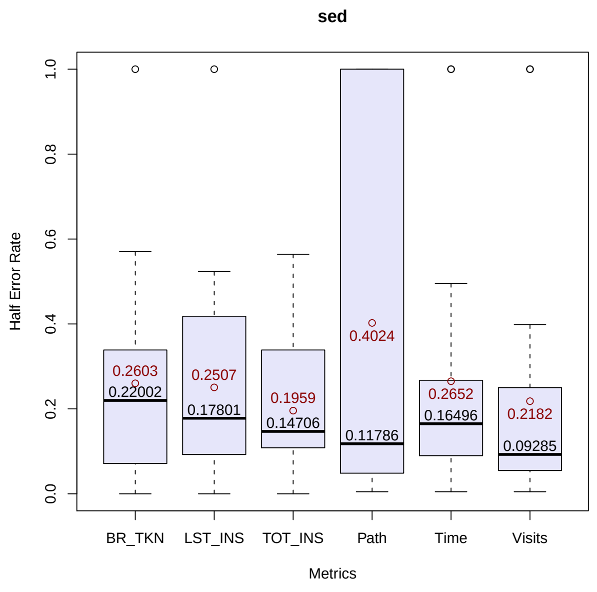
(a)


flex
Half Error Rate
Metrics
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
0.0
0.2
0.4
0.6
0.8
1.0
0.1584
0.13485
0.1321
0.11879
0.1302
0.12260
0.3272
0.16051
0.1255
0.06459
0.1618
0.06667

(b)

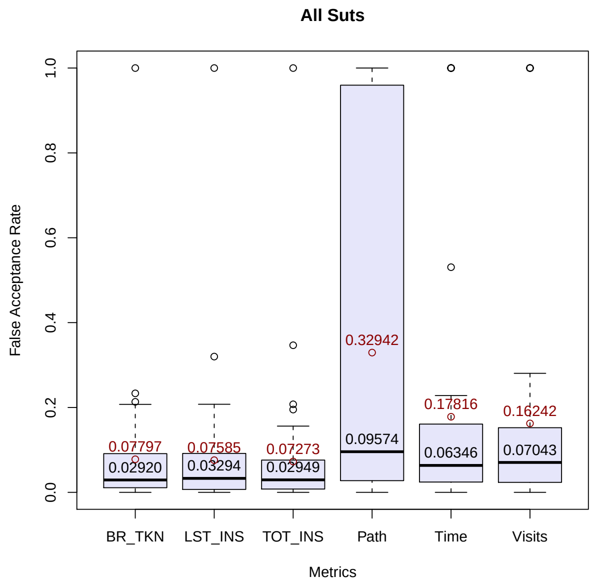


(c)

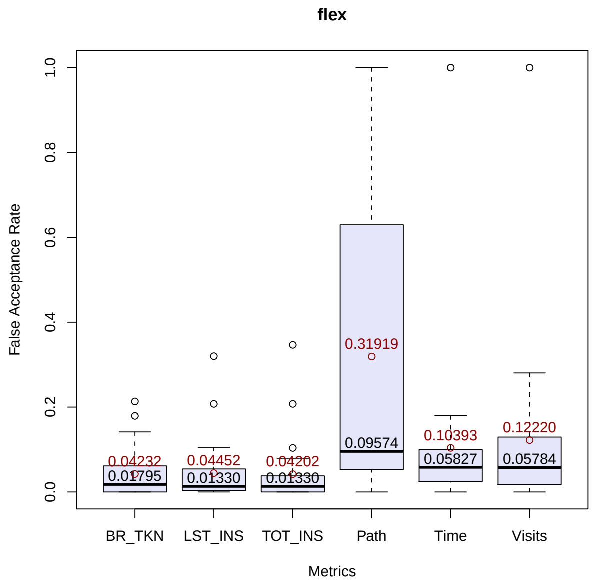


(d)

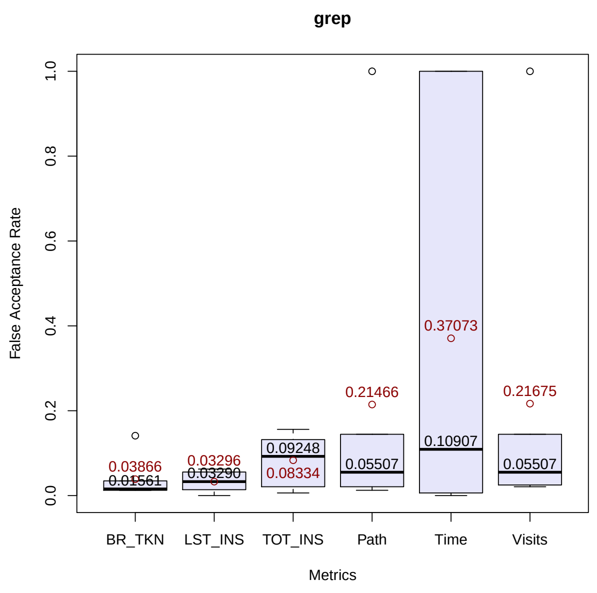
Figure 13: HER Analysis of PP Method


All Suts
False Acceptance Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.07797
0.02920
0.07585
0.03294
0.07273
0.02949
0.32942
0.09574
0.17816
0.06346
0.16242
0.07043
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

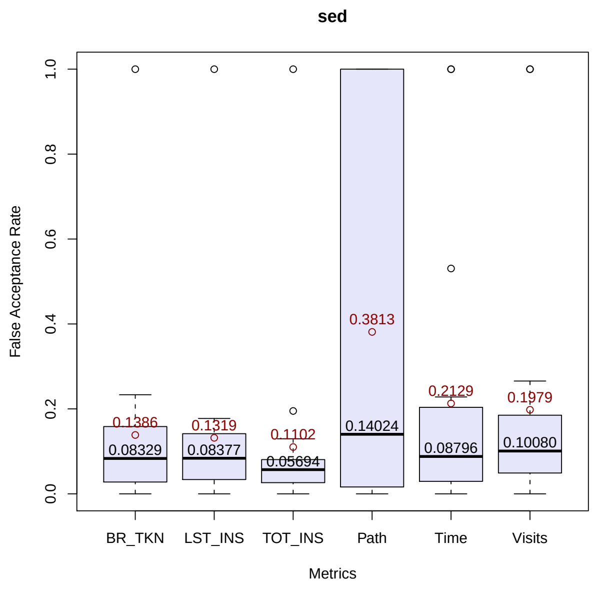
(a)


flex
False Acceptance Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.04232
0.01795
0.04452
0.01330
0.04202
0.01330
0.31919
0.09574
0.10393
0.05827
0.12220
0.05784
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

(b)

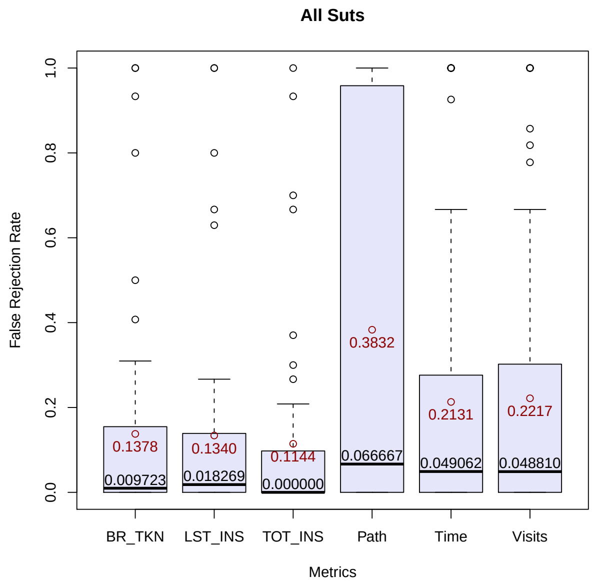


(c)

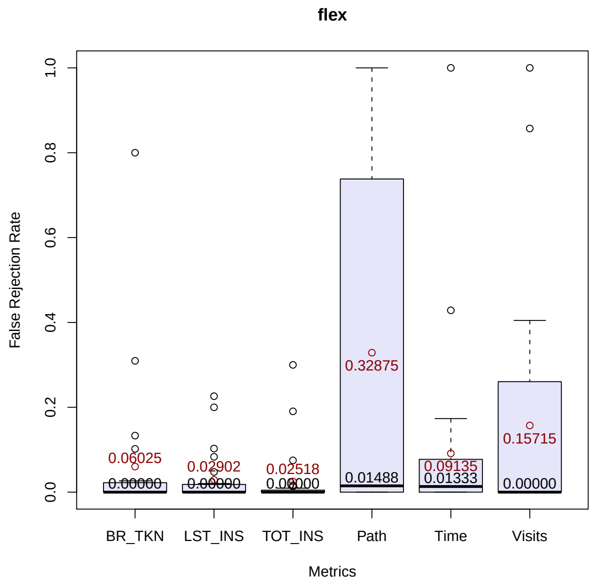


(d)

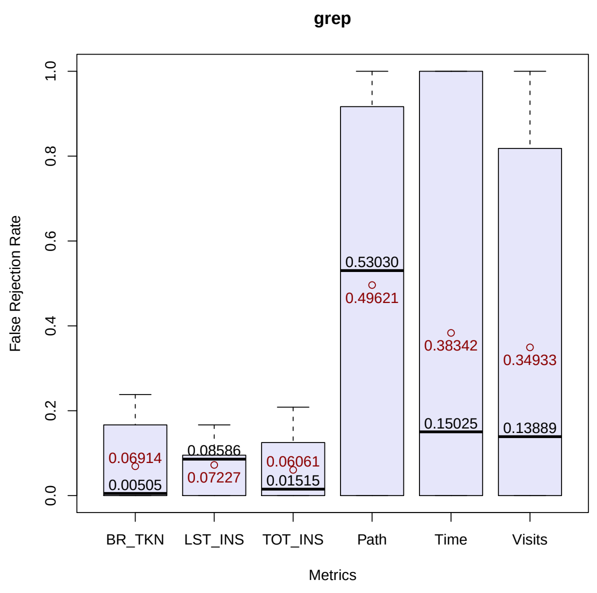
Figure 14: FAR Analysis of SWP Method


All Suts
False Rejection Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.1378
0.009723
0.1340
0.018269
0.1144
0.000000
0.3832
0.066667
0.2131
0.049062
0.2217
0.048810
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

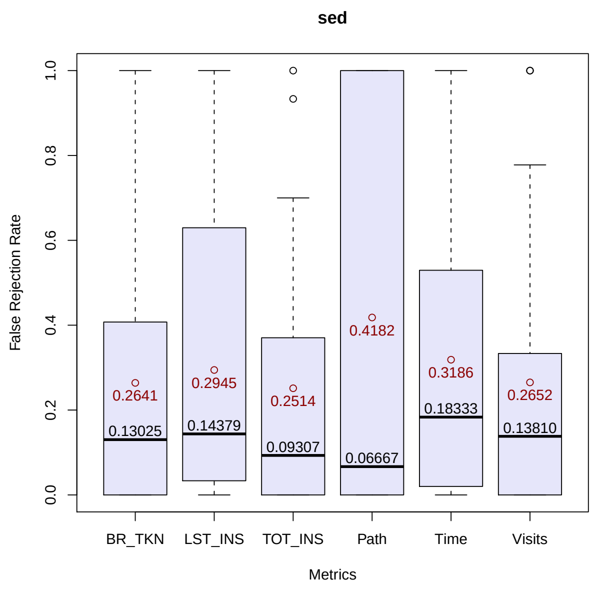
(a)


flex
False Rejection Rate
0.0
0.2
0.4
0.6
0.8
1.0
0.06025
0.00000
0.02902
0.00000
0.02518
0.00000
0.32875
0.01488
0.09135
0.01333
0.15715
0.00000
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

(b)

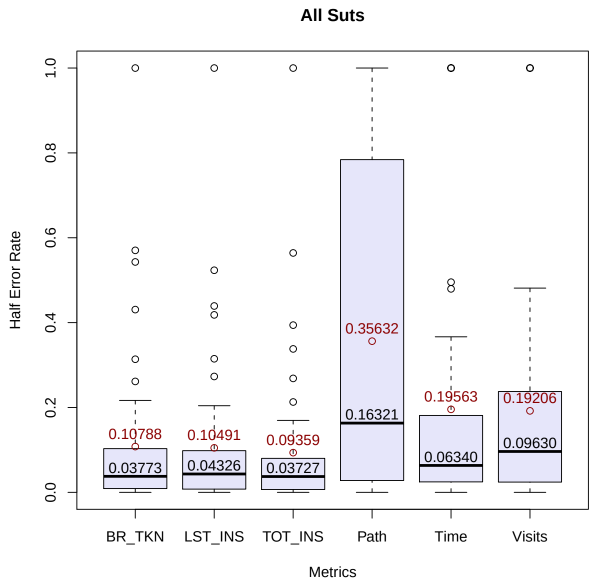


(c)

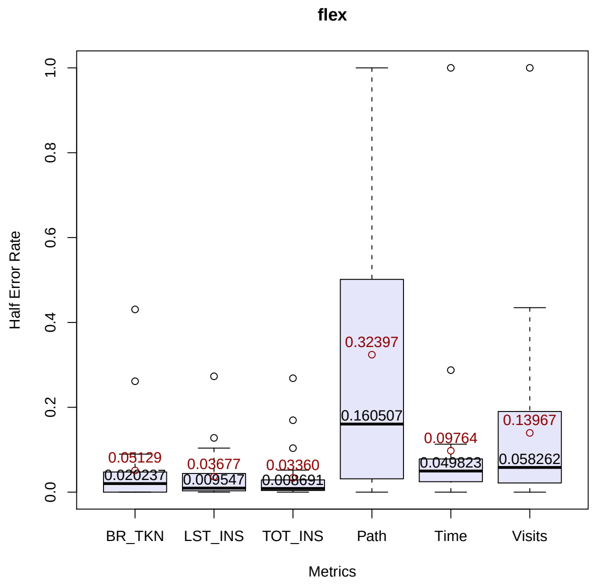


(d)

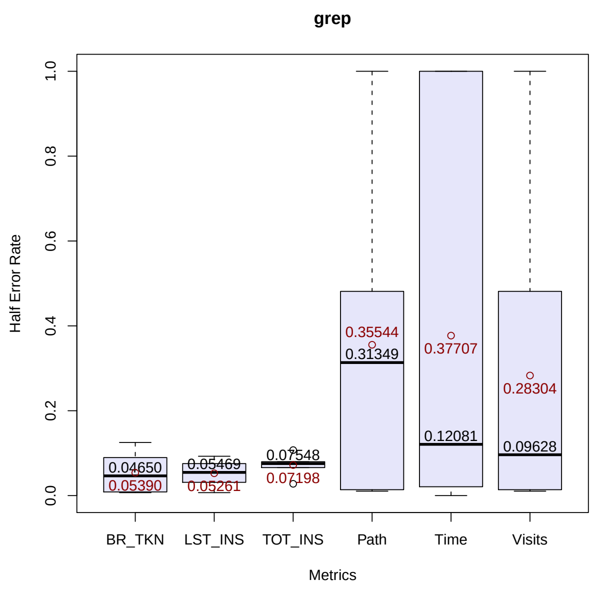
Figure 15: FRR Analysis of SWP Method


All Suts
Half Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.10788
0.03773
0.10491
0.04326
0.09359
0.03727
0.35632
0.16321
0.19563
0.06340
0.19206
0.09630
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

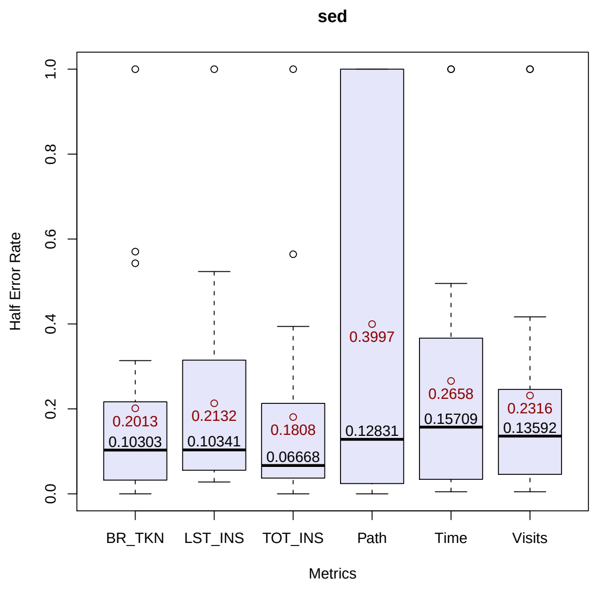
(a)


flex
Half Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
0.05129
0.020237
0.03677
0.009547
0.03360
0.008691
0.32397
0.160507
0.09764
0.049823
0.13967
0.058262
BR_TKN
LST_INS
TOT_INS
Path
Time
Visits
Metrics

(b)



(c)



(d)

Figure 16: HER Analysis of SWP Method

functions filtered may be related to the failures (hence the decline in success) or may be unrelated to them and causing a noise in the prediction (explains the increase in the success).

Then, overhead analysis of the filtered methods are analyzed. Figure 19 shows these results. One thing can be deduced form this plot is that, even though we observed some decline in failure prediction performance, overhead cost is really lowered. Also, we can see that under the filters, all six of the metrics produce a manageable overhead, which was the point of filtering.

As an addition for overhead analyses, we plotted the relation between prediction and monitoring times of the method. Figure 20(b) and 20(a) shows these relations for non-filtered and filtered methods respectively.

Another important factor of failure prediction is warning time, so, at this point warning times of the two methods are analyzed. Figure 21(a) and 21(b) depicts the warning times of these methods. These plots tell us that even though they all performed successful predictions, hardware performance counters based models were able to predict failures much more earlier than normal execution data based models.

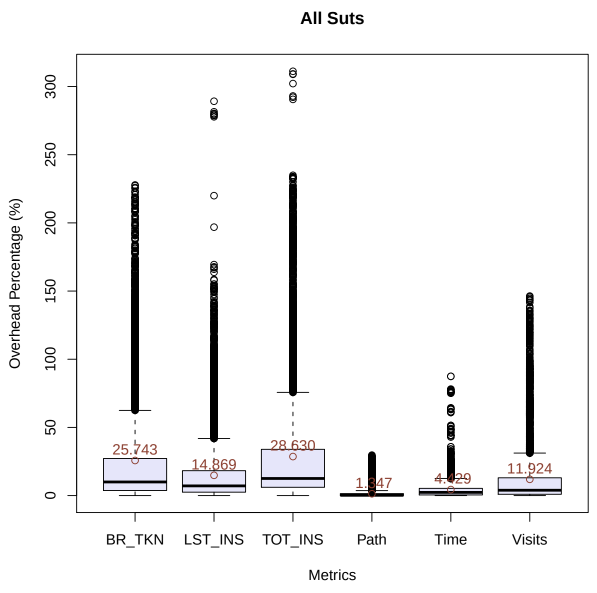
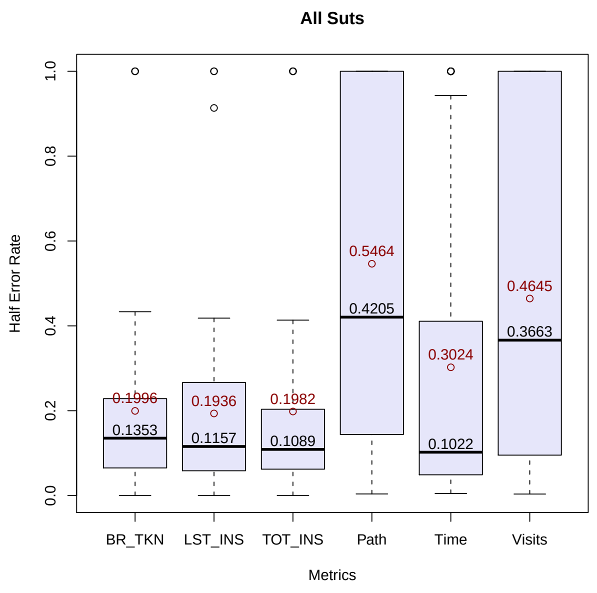
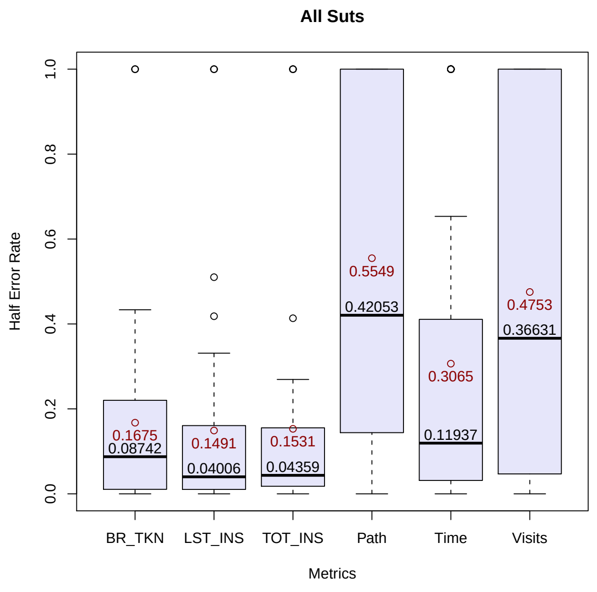


Figure 17: Time Overhead of SWP Method



(a) PP method



(b) SWP method

Figure 18: HER Analysis of PP and SWP Methods After Filtering

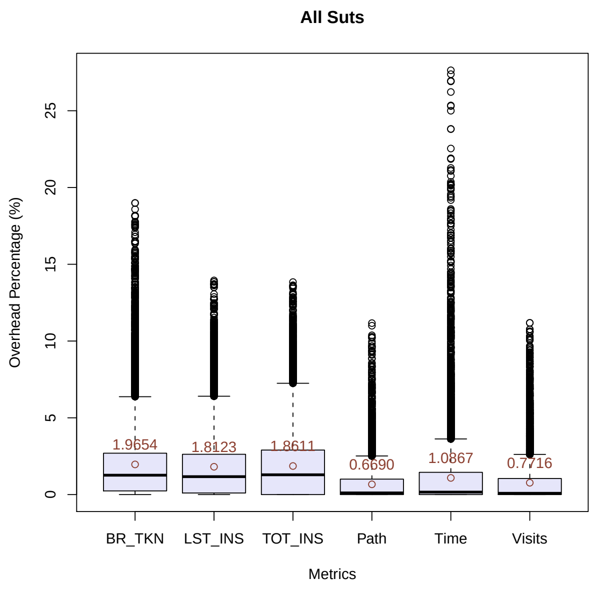
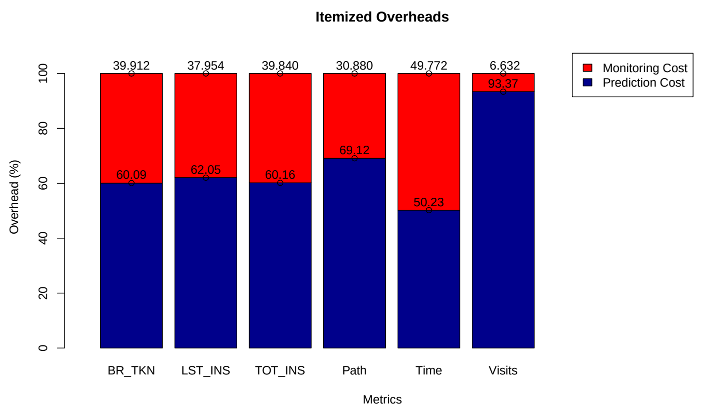
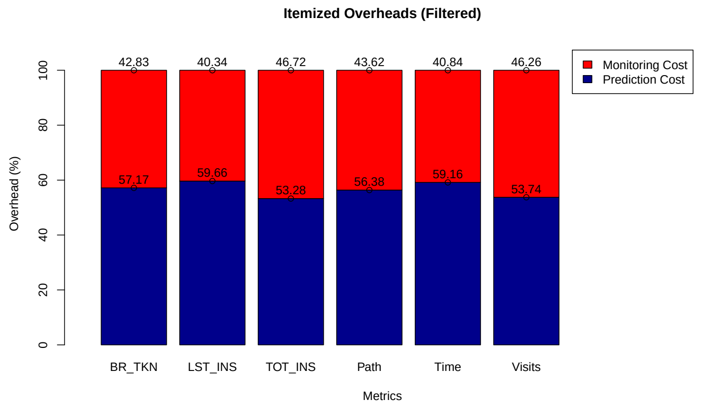


Figure 19: Time Overhead of SWP Method After Filtering

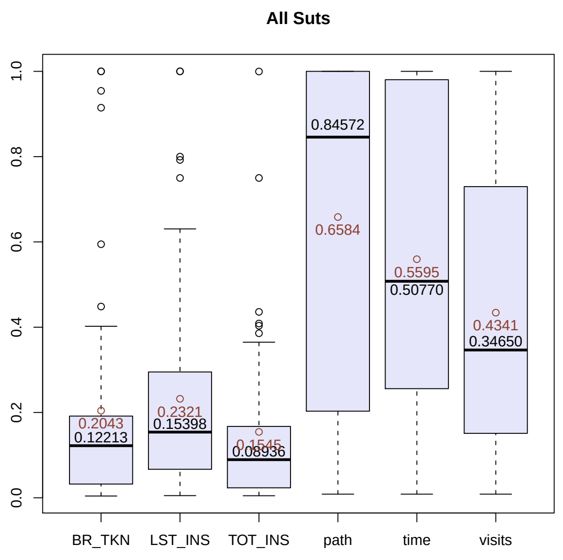


(a) Without Filtering

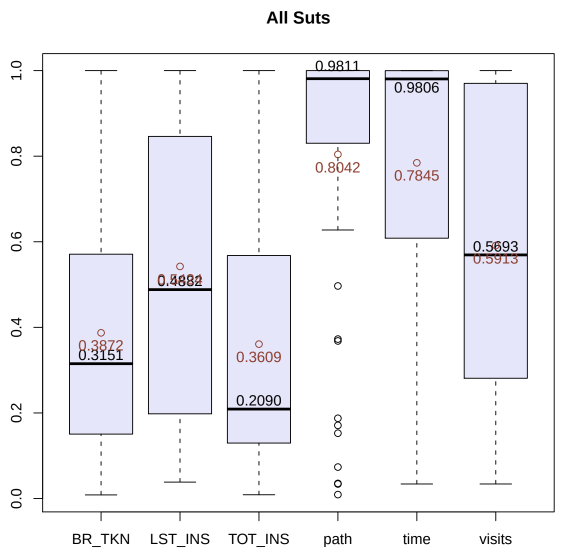


(b) With Filtering

Figure 20: Analysis of Monitoring and Prediction Costs



(a) PP method



(b) SWP method

Figure 21: Warning Point Analysis of PP and SWP Methods

7 Threats to Validity

All empirical studies suffer from threats to their validities, both internal and external. Hence this work also has some threats to its internal and external validity.

In this work, our main concern is threats to external validity because they prevent us to generalize the results of the study. One of the external threats is the representativeness of the subject applications. These applications are widely used real life applications nonetheless they are three small software, therefore they only represent three data points.

Another external threat concerns the representativeness of the defects used for experiments. Even tough the subject applications and the defects were taken from an independent repository which was used in similar related studies, these defects are still hasd-seeded faults. Also, during the experiments, always one defect is activated in the applications.

Regarding the internal threads, the subject applications were quite short running. Long running test cases or software systems may introduce some scalability issues. Also since the applications are so short running resolution of the timing method was quite effective on the experiment results.

8 Conclusion

In this study, we aspired after a lightweight method for failure prediction using internal execution data collected from software systems. Our aim was to show that efficient inner execution data can be collected with low overhead cost.

To carry out this purpose, we experimented six different metric types for collecting the execution data. Three of them were traditional metrics; path, visits, and time. The other three, TOT_INS, BR_TKN, and LST_INS, were collected by using hardware performance counters. Leveraging hardware performance counters for collecting inner execution data, is a new approach in failure prediction.

We collected inner execution data on the basis of functions. After data collection, behavioral models created for each agent and failure prediction accuracy evaluations made for each function model. The selected functions which have high success in failure prediction (i.e. where F-measure is higher than 0.8) were used for later stages of prediction.

During the executions, each selected function -which we call *agents*- creates a prediction about upcoming failures. We proposed two methods for using the predictions of agents for creating a prediction mechanism; *Point Prediction* and *Sliding Window Prediction*.

We evaluated our methods in three widely used, real life applications. Conducted experiments showed that, our approaches are effective and feasible in failure prediction. To be precise, our sliding windows method performed a failure prediction by seeing 33% of the executions on the average, with less than a 2% overhead cost and with a half error rate of 4% on mean and 15% on average.

In conclusion, in this study, we showed that, inner execution data collected from program executions, can be efficiently used for failure prediction. We also made an inference that, hardware performance counters can be used to collect such execution data.

9 Future Work

Further research on this study can be carried out upon long running applications. All of the subject applications used in the experiment are short running applications, hence long running experiments may introduce some unexpected challenges.

Attempting to fine-tune the results we obtained could be another way to go for future studies. For our main purpose in this study is to evaluate the usability of inner execution data on failure prediction, we did not try to optimize some steps of the experiments. To enhance the results, for instance, one can try other threshold values but 0.8 for agent selection process or several classification techniques can be considered for the approach.

References

[1] S. L. Salzberg, *C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993*, vol. 16. Springer Netherlands, 1994. 10.1007/BF00993309.

[2] A. Srivastava, E.-H. Han, V. Kumar, and V. Singh, "Parallel formulations of decision-tree classification algorithms," *Data Min. Knowl. Discov.*, vol. 3, pp. 237–261, September 1999.

[3] D. L. Olson and D. Delen, *Advanced Data Mining Techniques.* Berlin/Heidelberg, Germany: Springer-Verlag, 2008.

[4] B. Everitt, *The Cambridge dictionary of statistics.* Cambridge University Press, 2002.

[5] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, vol. 2304, pp. 213–228. Springer, 2002.

[6] S. Forge, I. Language, and A. Cil, "Cil : Infrastructure for c program analysis and transformation," *International Conference on Compiler Construction*, pp. 1–74, 2007.

[7] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Not.*, vol. 32, pp. 85–96, May 1997.

[8] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and

W. E. Weihl, "Continuous profiling: where have all the cycles gone?," *ACM Trans. Comput. Syst.*, vol. 15, pp. 357–390, November 1997.

[9] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, (Washington, DC, USA), IEEE Computer Society, 1996.

[10] V. Bui, B. Norris, K. Huck, L. C. McInnes, L. Li, O. Hernandez, and B. Chapman, "A component infrastructure for performance and power modeling of parallel scientific applications," in *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, CBHPC '08, (New York, NY, USA), pp. 6:1–6:11, ACM, 2008.

[11] G. Krawezik, "Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors," in *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '03, (New York, NY, USA), pp. 118–127, ACM, 2003.

[12] M. M. Tikir and J. K. Hollingsworth, "Using hardware counters to automatically improve memory performance," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 46–, IEEE Computer Society, 2004.

[13] H. Sasaki, Y. Ikeda, M. Kondo, and H. Nakamura, "An intra-task dvfs technique based on statistical analysis of hardware events," in *Proceedings of the 4th international conference on Computing frontiers*, CF '07, (New York, NY, USA), pp. 123–130, ACM, 2007.

[14] R. Azimi, M. Stumm, and R. W. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, (New York, NY, USA), pp. 101–110, ACM, 2005.

[15] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, (Washington, DC, USA), IEEE Computer Society, 2000.

[16] B. Mohr and F. Wolf, "Kojak a tool set for automatic performance analysis of parallel programs," in *Euro-Par 2003 Parallel Processing* (H. Kosch, L. Bszrmnyi, and H. Hellwagner, eds.), vol. 2790 of *Lecture Notes in Computer Science*, pp. 1301–1304, Springer Berlin / Heidelberg, 2003.

[17] 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.

[18] 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*, vol. 3460, pp. 143–157. Springer-Verlag, 2005.

[19] J. Gray, "A census of tandem system availability," in *IEEE Transactions on Reliability*, pp. 40–9, 1990.

[20] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation, Third Edition.* A K Peters/CRC Press, 1998.

[21] T. Y. Lin, M. Ieee, D. P. Siewiorek, and F. Ieee, "Error log analysis: Statistical modeling and heuristic trend analysis," *IEEE Transactions on Reliability*, vol. 39, pp. 419–432, 1990.

[22] R. Vilalta, S. Ma, and J. Hellerstein, "Rule induction of computer events," 2001.

[23] R. Vilalta and S. Ma, "Predicting rare events in temporal domains," in *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, (Washington, DC, USA), pp. 474–, IEEE Computer Society, 2002.

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

[25] F. Salfner, M. Schieschke, and M. Malek, "Predicting Failures of Computer Systems: A Case Study for a Telecommunication System," in *International Parallel and Distributed Processing Symposium, Workshop on Dependable Parallel Distributed and Network-centric Systems*, IEEE, Apr. 2006.

[26] F. Salfner and M. Malek, “Using Hidden Semi-Markov Models for Effective Online Failure Prediction,” in *26th International Symposium on Reliable Distributed Systems*, pp. 161–174, IEEE, 2007.

[27] C. Yılmaz, A. M. Paradkar, and C. Williams, “Time will tell: fault localization using time spectra,” in *ICSE* (W. Schäfer, M. B. Dwyer, and V. Gruhn, eds.), pp. 81–90, ACM, 2008.

[28] P. J. Phillips, A. Martin, C. l. Wilson, and M. Przybocki, “An introduction to evaluating biometric systems,” *Computer*, vol. 33, pp. 56–63, February 2000.