Time Will Tell: Fault Localization Using Time Spectra

Cemal Yilmaz, Amit Paradkar, and Clay Williams
IBM T. J. Watson Research Center
Hawthorne, NY 10532
{cyilmaz, paradkar, clayw}@us.ibm.com

ABSTRACT

We present an automatic fault localization technique which leverages time spectra as abstractions for program executions. Time spectra have been traditionally used for performance debugging. By contrast, we use them for functional correctness debugging by identifying pieces of program code that take a "suspicious" amount of time to execute. The approach can be summarized as follows: Time spectra are collected from passing and failing runs, observed behavior models are created using the time spectra collected from passing runs, and deviations from these models in failing runs are identified and scored as potential causes of failures. Our empirical evaluations conducted on three real-life projects suggest that the proposed approach can effectively reduce the space of potential root causes for failures, which can in turn improve the turn around time for fixes.

Categories and Subject Descriptors

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

General Terms

Measurement, Experimentation, Reliability

Keywords

Fault localization, Automated debugging

1. INTRODUCTION

Program debugging is a process of identifying and fixing bugs. Identifying the root causes is the hardest, thus the most expensive, component of debugging. Developers often take a slice of the statements involved in a failure, hypothesize a set of potential causes in an *ad hoc* manner, and iteratively verify and refine their hypotheses until root causes are located. Obviously, this process can be quite tedious and time-consuming.

Many approaches have been proposed in the past to facilitate fault localization. They all have the same ultimate goal to narrow down the space of potential root causes for developers, but a different way to achieve it.

Perhaps, the most eagerly studied type of approach is program spectrum-based approaches. A program spectrum can be considered as an abstraction of a program execution. Various forms of spectra can be defined. Statements executed, branches covered, and call sequences observed in executions are just few examples.

All program spectrum-based approaches operate in the same way: Program spectra are collected from passing and failing runs, models that capture the behavior of the program as observed in passing runs are created, and then deviations from these models in failing runs are identified and scored as potential causes of failures.

The fundamental assumption behind this approach is that the "observed behavior" (as observed from passing runs) is the same with respect to the abstracted feature as the "correct behavior" of the program (as documented in requirement specifications), or at least a safe subset of the correct behavior. Therefore, any deviation from the observed behavior is considered as a deviation from the correct behavior, even if it might not always be the case. Although a highly debatable assumption, many studies suggest that it may hold in practice [15, 14, 10, 4].

Our personal experience also supports this assumption. We have often observed that comparing passing and failing runs helps developers pinpoint the root causes of failures. On the other hand, we also noticed that leveraging an adequate type of program spectrum is the key to the success, since it reduces the gap between the observed and the correct behavior models. To this end, we believe that the types of program spectra currently in use today suffer from some major limitations.

One common limitation is that the existing spectra focus only on a very specific feature of program executions and collect precise information about that feature. Although the resulting abstractions are good at answering queries on the monitored feature, they cannot answer queries on even slightly different features. For example, statement coverage information would tell which statements were executed in a run, but cannot tell if two statements in a method were ever executed together.

Another limitation is that existing spectra often do not deal with sequences of events that happen in executions. Even the ones that capture some form of sequence information limit themselves to sequences of up to a certain length [14] because of the combinatorial complexity involved in analyzing sequences. Since a program execution is a se-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE'08, May 10–18, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-079-1/08/05 ...$5.00.

quence of state transformations, one could easily imagine that effective modeling of sequences should be an important part of any abstraction mechanism.

These limitations and many others are hard to address. A program execution is a complex event. It consists of a sequence of state transformations each of which comes to existence as a result of complex interactions between many factors. Therefore, it is genuinely hard to find the right level of abstraction for program executions.

In general, it is possible to collect more detailed information at runtime to come up with better abstractions. However, the overhead cost both in terms of the time overhead required to collect the spectra and the space overhead required to store them often makes it impractical. Even if the runtime cost is manageable, it is often unclear what to collect and how to analyze such large amount of heterogenous information to identify meaningful patterns and to create observed behavior models.

In this paper we try something different. Rather than dealing with these issues individually, we, in a sense, deal with all of them at once by leveraging a simple program spectrum, called a *time spectrum*.

Time spectra, e.g., traces of method execution times, have been traditionally leveraged in performance debugging to detect performance bottlenecks in the program code. Our approach uses time spectra in a novel way, exploiting them for functional correctness debugging by identifying pieces of program code that take a "suspicious" amount of time to execute. We call this approach *Time Will Tell* (TWT).

A time spectrum is relatively easy to collect (i.e., no complex program analysis is required) and analyze. Yet it reflects everything that happens in an execution, including sequences of events and complex interactions between various factors. In a sense, time tells us about everything, yet nothing in particular.

Our empirical evaluations of the TWT approach conducted on three open source real-life projects suggest that time spectra can be used to effectively reduce the space of potential root causes for failures, which can in turn improve the turn around time for fixes.

2. RELATED WORK

Agrawal et al. were one of the pioneers in the field to state that the manifestation of failures is highly correlated with differences in program spectra collected from passing and failing runs [3]. They leverage statement coverage information to compute the set difference between the statements covered by passing and failing runs. The statements that appear in a failing run but not in any of the passing runs are reported as potential sources of failures. Pan et al. provide similar heuristics to localize faults by leveraging dynamic program slices [13].

Jones et al. improve Agrawal's work by allowing some tolerance for faulty statements to be occasionally executed by passing runs [10]. Their empirical studies suggest that this tolerance often provides for more effective fault localization. They assign a score to each statement executed in a failing run, which reflects the likelihood that the statement is faulty; the more a statement appears in failing runs and the less it appears in passing runs, the more suspicious it becomes.

Dallmeier et al. use method invocation sequences as a defect indicator [4]. They identify and score "suspicious" sequences of method calls in failing runs in a similar manner with Jones et al.

Reps et al. study the use of path spectrum in program debugging with applications to the year 2K problems [15]. Their studies suggest that comparing path spectra provides a heuristic for identifying paths in a program that are good candidates for being date-dependent computations. Harrold et al.'s empirical evaluations of several types of program spectra support Reps et al.'s results by revealing that certain types of spectra differences correlate with high frequency with the exposure of regression faults [8].

Renieris et al. suggest that comparing a failing run only to those passing runs that most resemble the failing run can improve the accuracy of the approaches in localizing faults, as opposed to comparing it to all or arbitrary passing runs [14].

Many other types of fault localization approaches exist besides the spectrum-based ones. These approaches can be categorized into four broad categories: Model-based approaches [18, 17, 12], empirical analysis-based approaches [19], static analysis-based approaches [6, 9], and dynamic analysis-based approaches [11, 7].

The TWT approach is unique among the spectrum-based approaches in its use of time to localize faults. In the next section, we describe TWT in more details.

3. THE TIME WILL TELL APPROACH

Time spectra have been traditionally used in performance debugging to identify *hotspots* in the program code, i.e., pieces of program code that account for the vast majority of execution times. For this purpose, time serves as a numerical value.

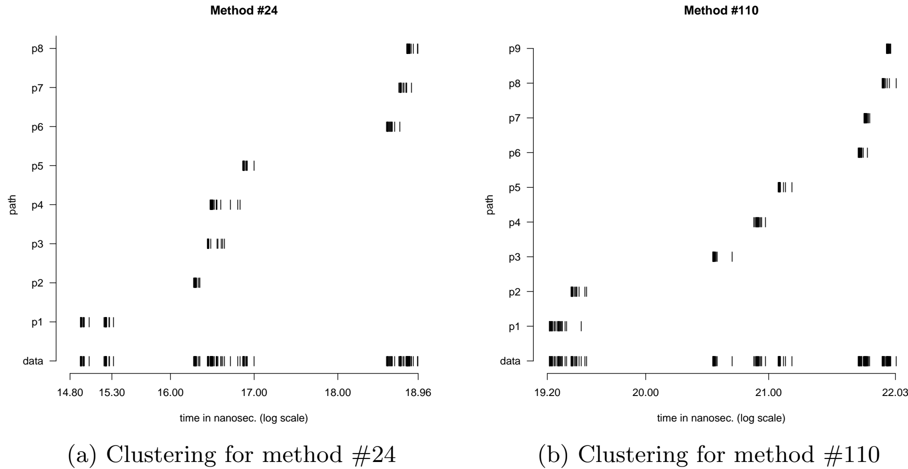
In TWT, our ultimate goal is to help developers reduce the space of potential root causes for failures. To accomplish this we use time spectra in many different ways. First, we use time as an abstraction mechanism for program executions. Second, rather than looking for hotspots, we identify pieces of program code that take a suspicious amount of time to execute. For example, in the TWT approach, it is irrelevant if a method consistently takes long (or short) time to execute across both passing and failing runs. On the other hand, if the same method takes suspiciously longer or shorter in a failing run compared to that in passing runs, we flag it.

The granularity of our analysis is at the level of a method. We chose this granularity since methods provide well-defined code and functionality boundaries. Consequently, a time spectrum for us is a trace of method execution times collected during an execution.

TWT takes as input a set of passing runs and a failing run. At a high level, the TWT approach can be summarized as follows: 1) time spectra are collected from the passing and failing runs; 2) observed behavior models are created using the time spectra collected from the passing runs; and 3) deviations from these models in the failing run are identified and scored as potential causes of the failure. The output of the TWT approach is a ranked list of method invocations observed in the failing run, sorted in descending order by the level of their suspiciousness.

3.1 Time as an Abstraction Mechanism

We believe that the time metric has several attributes that make it attractive as an abstraction mechanism to localize functional bugs.



(a) Clustering for method #24 (b) Clustering for method #110

Figure 1: Clustering of time measurements for method #24 and #110.

First, time reflects everything that happens during an execution, including complex runtime interactions between various factors (e.g., data flow, control flow, environment, etc.), some of which could otherwise require complex and expensive program analysis to analyze (if analysis is possible).

Second, time can be indicative of many things. For example, the time needed to execute a method can provide clues about the branches taken, methods invoked, and paths exercised during the execution of the method.

Last but not the least, since time is an aggregate measure, it seamlessly models sequences of events. For example, a program execution can be considered as a sequence of method invocations. A caller method invokes a callee method to handle a piece of functionality and then uses the result of the callee to fulfill its own requirements. Therefore, it is desirable that the abstraction of the caller method integrates the abstraction of the callee method. Time seamlessly supports such integrations; the time needed to execute a caller method by definition includes the time needed to execute its callee methods.

All programs in one form or another are eventually compiled into machine instructions and then executed. A time cost is associated with executing each instruction. Consequently, the time needed to execute simple sequential code without any loops and branches is directly proportional to the number and the type of instructions the code is compiled into. Methods, depending on their input arguments and the state of the program as well as the state of the environment, may exercise different paths, which may potentially be compiled into different set of instructions. Therefore, in theory, these differences should be reflected on the method execution times.

3.1.1 A feasibility study

To evaluate the plausibility of this hypothesis in practice, we conducted a feasibility study where we used an open source XML parser, called nanoXML. More information about the subject application can be found in Section 4.

We instrumented the application and collected the time spectra across 170 passing and 44 failing test cases. These test cases came with the original nanoXML distribution. In particular, we focused on the *parseDTD* method (method #109), since its implementation was fairly easy to understand. The purpose of this method is to parse the Document Type Definition (DTD) portion of XML documents. The DTD defines the legal building blocks of an XML document. The functionality of method #109 was implemented using two main methods: Method #24 which skips white space characters in a DTD and method #110 which processes DTD elements. These methods are called in a sequence until the entire DTD is parsed. Note that the actual implementation of method #109 is somewhat more complicated (e.g., it uses 6 methods not just 2). However, for the sake of simplicity, we focus only on method #24 and method #110. A total of 16 passing test cases exercised the parseDTD method.

An initial question we had about the time measurements we collected is whether they correlate with some features of executions. If they don't, then any analysis based on them will obviously suffer.

Since time is an aggregate measure, i.e., it accounts for every instruction executed from the start of a method to the end, we chose to correlate the time to execute a method with the path taken by the method.

We monitored the interprocedural path taken by method #24 and #110. These methods were invoked a total of 187 and 156 times, respectively. All the invocations originated from method #109. We then categorized similar paths into groups. Our criterion for similarity ensures that 1) each path belongs to exactly one group, 2) within a group, each path has the same sequence of statements (the superset of statements executed in loop iterations is computed an used only once in a path), and 3) the differences in the number of loop iterations within a group are at most three.

Figure 1 illustrates the clusters of similar paths we identified for each method. The horizontal axis denotes a method execution time in nanoseconds (on a log scale). The vertical axis denotes a cluster, except for the first row that depicts all the data we had for the analysis. Each bar represents a single invocation of a method. As the figure portrays, we identified 8 clusters for method #24 and 9 clusters for method #110.

(a) Time table.

body	m24	m110	...
262046	1627578	12781234	...
262882	1635777	12827700	...
200772	779428	10135366	...
267262	1699936	13145525	...
205427	789765	10108827	...
...	...	...	...

(b) Percentage table.

body	m24	m110	...
1.69	10.52	82.65	...
1.69	10.55	82.75	...
1.75	6.82	88.72	...
1.68	10.70	82.75	...
1.79	6.91	88.52	...
...	...	...	...

Table 1: Time and percentage table for method #109.

We then performed a similar type of clustering, only this time we based our clustering solely on method execution times; no path information was used. Our goal is to compare the clusters obtained solely from execution-time information to the ones obtained solely from path information and evaluate how well these two different sets of clusters correlate.

We leveraged the k-means [16] clustering algorithm to cluster the method execution times. We then used the resulting clusters (not shown here for space considerations) to predict the path taken in each method invocation. This was done by performing a classes-to-clusters analysis [16], a well-known analysis technique for the purpose at hand. The results were very encouraging. The clusters based solely on method execution times were able to predict the path taken in each method invocation with an accuracy of 85% for method #24 and 94% for method #110. These results suggest that the execution times of these methods could have been a predicator of the paths taken by them and vice versa.

Another question we had about the time measurements we collected was whether they identify some behavioral patterns among methods; not just the behavior of a single method as was the case in our previous analysis. This is important, since deviations from such patterns can potentially signal the presence of bugs.

Figure 2(a) plots the execution time of method #24 against the execution time of method #110. Each point in the figure represents a single invocation of method #109. The time measurements are aggregated and given in nanoseconds on a log scale on both axes. For example, if method #24 is called multiple times in an invocation of #109, the sum of its execution times is used.

As the figure depicts there is a strong pattern between the execution times of these two methods; when method #24 takes relatively longer to execute, method #109 takes relatively longer. In fact, we found an almost perfect correlation with a correlation coefficient of more than 0.98 between the execution times of these methods. A correlation coefficient ranges between 0 for a complete absence of correlation and 1 for a perfect correlation.

This makes perfect sense, since DTD elements are separated by white space characters. The more white space characters in a DTD definition, the more DTD elements there are to be parsed. Consequently, the more number of times method #24 is called to skip white space characters between the DTD elements, the more number of times the method #110 is called to process the elements. As the figure depicts, the number of times each method is invoked is reflected on the aggregate time measurements we collected, which allowed us to capture the behavioral pattern.

Another thing to note about this figure is that there are two obvious clusters in the data. An in-depth analysis revealed that these clusters represent the similarities between the DTD definitions used in the test cases.

We were pleased by the fact that a very small number of data points (only 31) allowed us to capture some interesting patterns. In TWT, the number of data points available for analysis is dictated by the passing and failing runs that we have no control over. Therefore, being able to capture patterns on small amount of data is crucial.

Although the results of this feasibility study are by no means conclusive, they are encouraging and were decisive in our efforts to pursue the proposed approach.

3.1.2 Caveats

Our feasibility study also helped us identify some caveats associated with using time measurements.

Imprecision in measurements. The resolution of the time measurements is limited by the resolution of the software/hardware clocks available today. A low resolution may prevent us from detecting some important patterns.

Noise in measurements. The time to execute a piece of code may differ from execution to execution because of the noise imposed by the underlying platform. If not taken into account, noise in measurements may lead to detecting spurious patterns.

Dependency on software/hardware platforms. Time measurements may vary from platform to platform. For example, the same code may run faster on a more powerful CPU than it does on a less powerful one. Factoring out these dependencies would certainly improve the overall end-user experience of the proposed approach.

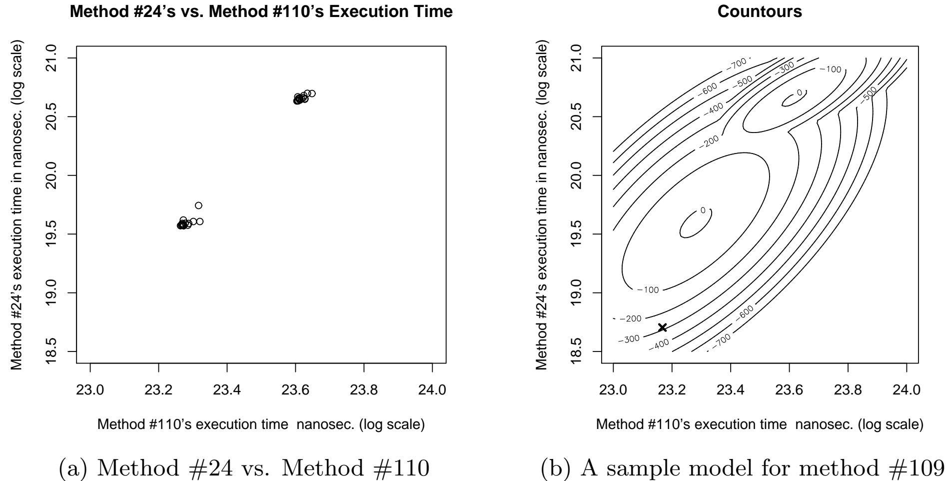
The next section describes how we address these caveats in creating observed behavior models.

3.2 Creating Observed Behavior Models

In TWT, the observed behavior models are created using the time spectra collected from passing runs. The rationale behind these models is to capture the statistical patterns across executions where we know the program exposes the correct functionality.

We create one observed behavior model for each method encountered in passing runs. Execution times of method invocations are collected, the measurements obtained from different invocations of the same method are gathered in a data set, and these data sets are used to create an observed behavior model for each method. The model of the entire program is then the collection of these individual models.

Handling imprecision in measurements. One way we deal with the imprecision in time measurements is to use a high resolution clock. In this work, we measure method execution times at the level of nanoseconds (i.e., one billionth



(a) Method #24 vs. Method #110 (b) A sample model for method #109

Figure 2: An example observed behavior model.

of a second), which is supported on almost all platforms today.

Another way we deal with the imprecision is to use some context information. In particular, we leverage caller-callee information in creating the models. We itemize the execution time of a method to reflect the time spent in the body of the method and the time spent in each callee method. The time spent in a callee method is aggregated over the invocations of the callee. If a method is called multiple times, the sum of its execution times is used. Furthermore, we compute the time spent in the body of a method as the execution time of the method minus the total time spent in the callee methods.

Table 1(a) gives as an example a portion of the time measurements collected for method #109 in a study. Each row in this table represents a single invocation of method #109 in a passing run. The number of rows depicts the number of times the method is called across the passing runs. Each column represents the time spent in a callee or in the body. For example, the first row depicts an invocation of method #109 where it spent 262046 nanoseconds in its body, a total of 1627578 nanoseconds in method #24, and a total of 12781234 nanoseconds in method #110.

In the rest of the paper we refer to these tables as *time tables*. A special value, -1, is used in these tables to indicate the absence of the invocation of a callee method.

Each callee performs a portion of the caller's functionality. Therefore, the way we itemize the execution time of a caller captures the fact that how much time the caller spends in performing a particular functionality. This could provide valuable information towards localizing bugs, since, for example, the changes in ratios can signal deviations.

We have so far described how we prepare the data files we use in creating observed behavior models. We now present the way we create the models.

Handling noise in measurements. Two major factors were effective in our decision on how to create the observed behavior models. The models should be able to deal with 1) noise in measurements as discussed in Section 3.2 and 2) clusters in data as discussed in Section 3.1.1.

Consequently, we opted to create observed behavior models using Gaussian Mixture Models (GMMs). Given a data set, a GMM model first identifies the clusters in the data and then models each cluster using a Gaussian distribution (a.k.a. normal distribution). In other words, a GMM model is a mixture of Gaussian distributions that fits a given data set. GMM models are among the most statistically mature methods. Furthermore, they are able to smooth over gaps resulting from sparse data. This is a desirable property for us, since we don't have any control over the amount of data we collect (in the sense that it all depends on the passing and failing runs) and the data we collect may be sparse.

Figure 2(b) visualizes a GMM model created for method #109 using the data set given in Figure 2(a). This type of plot is called a *density contour plot*. A density contour plot is a graphical technique for representing a 3-dimensional density surface by plotting constant density slices, called contours, on a 2-dimensional format. Given a value of density, lines are drawn for connecting the (x, y) coordinates where that density stays the same.

As the figure depicts, the constructed GMM model correctly identifies the two clusters in the data, which are represented by the innermost ellipses. Consequently, the entire data is modeled by a mixture of two bivariate Gaussian distributions, one for each cluster. The centers of the clusters represent the mean of the distributions whereas the distance between contour lines provide clues about the variance and the skewness of the distributions. For example, in Figure 2(a), the data points in the lower left cluster are more scattered compared to those in the upper right cluster. Therefore, the isodensity contour lines in the lower left portion are farther away from each others compared to those in the upper right portion. The numbers on the contour lines denote the densities in log scale. As expected, the densities drop as we start moving away from the centers of the clusters.

Note that the time table given in Table 1(a) provides a portion of the data set used in Figure 2(a). In practice, the GMM models are created using the entire time tables. However, for visualization purposes, the GMM model given

in Figure 2(b) is created by using only two columns from Table 1(a), namely column m24 and m110.

In TWT, we first prepare the time tables for methods and then use them to create the GMM models, one for each method. We refer to these models as *time models*. For us, a time model such as the one given in Figure 2(b) captures the observed behavior of a method.

Handling dependency on software/hardware platforms. In an attempt to reduce the dependency of the observed behavior models on platforms, we used time percentages (as opposed to using actual time values) in creating the models.

Table 1(b) illustrates an example *percentage table* which was created from the time table given in Table 1(a). Each row in a percentage table corresponds to a row in a time table and is created by computing the percentages of time for each column. For example, the first row in Table 1(b) indicates that in this particular invocation 1.69% of the total execution time is spent in the body, 10.52% in method #24, and 82.65% in method #110. We refer to the GMM models created from percentage tables as *percentage models*.

The rationale is that since time percentages are *relative* to total execution times which are platform dependent, leveraging them could factor out the unwanted effect of platforms. Although, in Section 4, we evaluate the performance of percentage models alongside with time models in localizing faults on a single platform, we leave the evaluation across heterogenous platforms to another study.

3.3 Detecting and Scoring Deviations

Observed behavior models are created using the time spectra collected from the passing runs. In effect, they capture the behavior of the program as observed in the passing runs. In TWT, we report deviations from these models in failing runs as potential causes for failures. This section describes how we detect and score such deviations.

For a given failing run, we first create the time and percentage tables in the same manner as with the passing runs, one table for each method. We then feed these tables to the corresponding observed behavior models. For each invocation record (i.e., each row) in these tables, the output of a model is the density at the point represented by the record. We treat these densities as a measure of deviation from the normal behavior. Lower density readings signal deviations whereas higher readings are a sign of normality. We then use the density reading for an invocation record as the score of the suspiciousness of the invocation. For example, the symbol '×' given in Figure 2(b) represents an invocation of method #109 in a failing run. From the perspective of TWT, this invocation is a suspicious one, since the point is located away from the centers of the clusters. This fact is reflected in its low score, which is about -300 in log scale.

Once the scores are computed, they are sorted in ascending order and presented to the end-user as a diagnosis report.

4. EXPERIMENTS

We conducted a set of experiments to evaluate the TWT approach. This section reports the results of these experiments. We start with presenting our evaluation framework.

4.1 Evaluation Framework

The first challenge we faced in our experiments was evaluating the quality of the diagnosis reports in localizing bugs. A diagnosis report for a failing run is a ranked list of method invocations encountered in the run, sorted by their level of suspiciousness from the most suspicious one to the least.

One way of evaluating these diagnosis reports is to give them to the developers of the system and observe how much time it takes for them to locate the bug with and without the report. Unfortunately, we didn't have access to the developers of our subject applications. Instead, we chose to automatically score each diagnosis report. We surveyed the literature for a scoring scheme. However, we identified some issues with the proposed schemes [10, 14].

The first issue we encountered is that the score given to a diagnosis report by these schemes often reflects the percentage of the code entities, such as statements, branches, and methods, that need to be examined before the bugs can be located. However, the percentages are given relative to the size of the entire program code [14]; not relative to the execution trace as it should be. Since each failing run exercises often a small portion of the program code, such scores can be misleading.

Another issue is that it is almost always forgotten [10] that the diagnosis reports are meant to be processed by human beings which have very limited tolerance to false positives. For example, although a diagnosis report that ranks the faulty statement as hundredth in ten thousand lines of code may seem to be of very high quality in theory (i.e., it requires only 1% of the code to be examined), it might not be the case in practice. This is mainly because of the fact that this score does not give clues about the proximity of the first 99 reported statements to the actual faulty statement; they could be scattered all around the code and away from the faulty statement. Intolerance to false positives could prevent developers from starting with the first reported statement and progressing down to the hundredth reported statement.

Consequently, we developed our own scoring scheme. In this scheme, we consider an execution as a sequence of method invocations, which we call a *call sequence*. To address the issue of intolerance to false positives, we consider only the top ten ranked method invocations reported in the diagnosis reports; the rest is ignored.

We assume that a breadth-first search is performed over these top ranked invocations. In the first iteration, only the reported invocations are considered, starting with the most suspicious one working down to the least. At each step only one invocation is examined and the search stops as soon as the bugs are located, assuming that bugs are recognized on sight. If the bugs are not hit in the first iteration, then a second iteration starts by investigating the method invocations which in the call sequence are located right next (in either direction) to the invocations examined in the previous iteration. These invocations are examined in the same order as the previous iteration. The search continues iteratively until the bugs are found. The score for a diagnosis report is then defined as the percentage of the method invocations observed in the failing run that need to be examined before the bugs can be located. Note that the percentages computed in this scheme are relative to the dynamic call sequence and not to the program code, which addresses the first issue discussed above.

This scheme requires that the locations of the faults are known, which was the case in our experiments. Once the diagnosis reports are created for each failure, we use our

subject	LOC	classes	methods	base versions	faulty versions	total tests	passing tests	failing tests
nanoXML	7646	24	574	4	25	2703	2299	404
XMLsecurity	16800	143	1378	3	12	1098	902	196
ant	80500	627	8399	6	20	12322	12231	91

Table 2: Subject applications used in the experiments.

scoring scheme to score them. The lower the score, the better the diagnosis report is.

4.2 Creating Models

The next issue we faced in our experiments was how to create the actual GMM models. For this purpose, we used the MClust library of the R statistical package [2], which fully automated the creation of the GMM models from the time and percentage tables.

In the experiments, to improve the scalability of the approach, we actually took a fixed-size sample from each data table before we fed them to MClust. Samples should be taken with regard to the type of analysis that needs to be performed on them. Since GMM models are concerned with clusters and densities in data, we opted to take (in a sense) cluster- and density-preserving samples. For each data table, we automatically identified the clusters in the data using the k-means clustering algorithm (a more scalable analysis compared to MClust) and then took a weighted sample from each cluster. The weight of a cluster was directly proportional to the size of the cluster.

The size of our samples was around 1000 invocation records. The strong patterns we observed in the data we collected and the GMM's ability to work with small amount of data (as we haven seen in Section 3.1.1) were the two major factors in choosing the number 1000. The time and percentage models were then created using the sampled data.

In addition to the time and percentage models, we created two more models, namely *bare models* and *random models*. We now explain the intention behind creating these models and how they are created, starting with the bare models.

Our observed behavior models leverage two types of information; execution time information and context information, i.e., caller-callee information. In order to single out the effect of using time spectra in localizing faults from the effect of using context information, we created bare models.

Bare models were created using only the caller-callee information. This information was obtained by removing the time measurements from the time tables. In other words, each invocation record in a *bare table* reflects for each callee method whether the method is called (indicated by 1) or not (indicated by -1) in the invocation. Since the only difference between the time/percentage models and the bare models is the use of time, differences between the performance of these models can safely be attributed to using time spectra. As is the case with other types of observed behavior models we create, bare models are in the form of a GMM model.

To demonstrate that the results we obtained are not by chance, we created random diagnosis reports. A random report for a failing run was obtained by randomly selecting n method invocations from the run. To be fair in comparisons, n is chosen in a way so that the number of suspicious method invocations reported in the top ten rank is the same in the random reports and in the reports obtained from the time models. Note that multiple method invocations in a report may share the same rank. We created 100 random reports for each failing run, scored them, and used the average scores in our analysis.

4.3 Subject Applications

We used three open source real-life applications written in Java as our subject applications, namely nanoXML, XMLsecurity, and ant. NanoXML is a lightweight XML parser for Java. XMLsecurity is a component library implementing XML signature and encryption standards. Ant is a build tool, which is similar in terms of its functionality to the UNIX's *make* tool.

All the subject applications were taken from the Software-artifact Infrastructure Repository (SIR) [5]. The purpose of this repository is to provide a one-stop repository of open source real-life programs for use in experimentation with testing and program analysis techniques. SIR provided us with several sequential versions of the subject applications with seeded faults. To increase the external validity of results obtained on these faults, the faults were inserted by following a list of fault seeding guidelines [5]. To further ensure that faults will correspond, to the extent possible, to faults found in practice, fault seeding was performed by third-party experienced developers who did not possess knowledge of any specific experimental plan.

Table 2 provides further information about the subject applications used in the experiments.

4.4 Experimental Setup

Each application in SIR comes with two flavors: a base version and a faulty version. A base version represents the original version of the application whereas a faulty version is obtained by inserting a fault to a base version.

We first ran the test suites supplied by the base versions across all the faulty versions created from them. The test cases we used came with their own test oracles. Not all the seeded faults were revealed by the test cases. To better evaluate the TWT approach, we used only those faulty versions whose faults were revealed by at least one test case. The rest of the faulty versions were ignored. Furthermore, among the failing test cases only the ones that revealed a fault were considered for evaluation. That is, we used only those test cases that failed on a faulty version but passed on the corresponding base version.

The columns 5-9 in Table 2 depict the number of base and faulty versions, the total number of test cases, and the total number of passing and failing tests for each subject application used in the experiments.

We then implemented an instrumentation tool to collect the time spectra, which is built on top of the Byte Code Engineering Library (BCEL). Execution times were measured in nanoseconds using the *System.nanoTime()* method and persisted to disk for offline analysis. The collected time spectra are in the form of a call tree where each method invocation is annotated with its execution time.

subject	runtime overhead	passing call tree	failing call tree	sampling	Creating Models time	percentage	bare	Diagnosing time	percentage	bare
nanoXML	0.66	2175.34	2167.29	0.50	3.10	3.40	3.10	60.79	60.82	60.71
XMLsecurity	0.54	1841.20	3451.24	0.43	2.21	1.71	1.86	74.18	74.18	74.13
ant	0.45	11887.73	10386.84	1.18	5.38	5.48	4.75	372.28	447.29	447.50

Table 3: Statistics about the experiments.

Our major concern in the instrumentation was to be as unobtrusive as possible. For this purpose, we factored out the runtime cost of instrumentation as much as possible. For example, we adjusted the clock whenever the allocated data buffers (used for storing spectra) are dumped to the disk.

Lastly, we implemented the TWT approach as a fully-functional research prototype tool. Once the build files for the subject applications were given, this tool automated the entire process; no human intervention was required. For each faulty version, the TWT tool instrumented the binaries, ran the passing and failing tests, collected and parsed the time spectra to produce the data tables, sampled the data tables, created the observed behavior models, diagnosed each failing test, and then scored the diagnoses.

All the experiments were performed on an Intel Core Duo machine running Windows XP OS with 2Gb of memory.

4.5 Evaluation

We evaluated the TWT approach using a total of 15432 passing and 691 failing test cases across the subject applications. The breakdown of these numbers for each application is given in Table 2.

Figures 3-5 report the results we obtained. The plots given in the first column depict the histograms of scores obtained from the time models. In these plots, the horizontal axis denotes a score interval and the vertical axis denotes the number of diagnosis reports that fall into each score interval. The plots given in the second column compare the effectiveness of various types of models using Box-and-Whisker plots. The horizontal axis denotes a type of model and the vertical axis denotes a score. Each box in these plots illustrates the distribution of scores obtained from a model. The lower and upper end of a box represents the first and third quartiles and the horizontal bar inside represents the median value.

As the histograms depict we obtained very encouraging results. Out of 691 failing runs across the subject applications, 25% of the faults were located by examining up to 1% of the method invocations, 50.0% by examining up to 5%, and 63.7% by examining up to 10% of the method invocations in failing runs. The breakdown for each subject application can be computed from the histograms.

Furthermore, comparing these results to those of random experiments revealed that they are not by chance. A non-parametric Kruskal–Wallis test [2] identified statistically significant differences between the time and random models with a confidence interval of more than 99.99% in each case.

As is the case with all spectrum-based approaches, TWT depends on having an adequate suite of passing runs in order to reliably capture observed behaviors. An inadequate number of test cases may degrade the quality of the diagnoses. We observed this phenomenon in the experiments conducted on XMLsecurity. The average ratio of the number of passing runs to the number of failing runs in a version of XMLsecurity was around 4.6, a very small ratio compared to those of the other subject applications. This low ratio reflected on the quality of the diagnoses obtained; although 25% of the scores were below 5 and around 56% were below 10, only a small fraction of the scores was below 1. For nanoXML and ant, the ratio was 8.4 and 123.9 and 31.7% and 42.9% of the scores were below 1, respectively.

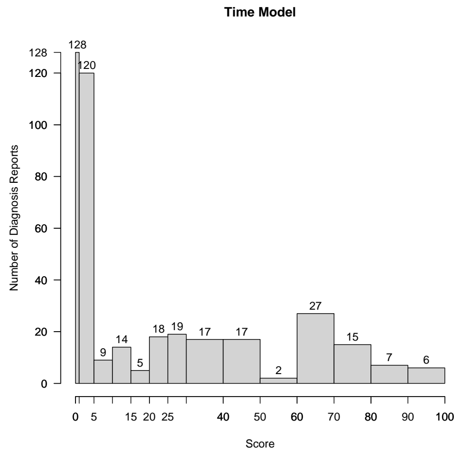
Another interesting observation is that sometimes the faulty methods were invoked only in failing runs. Since no information from passing runs was available for such methods, we were not able to create their observed behavior models. Therefore, we could not score their level of suspiciousness in failing runs, which in turn adversely affected the quality of the diagnoses. The 6 faults whose diagnosis reports got a score between 90 and 100 in Figure 3(a) were of this kind, for example. Although such method invocations could have been automatically assigned to the highest level of suspiciousness possible, we deliberately opted not to do so in order to be able to reliably single out the effect of leveraging time spectra in localizing faults.

Comparing time models to bare models revealed that using time spectra greatly improved the diagnosis of faults as compared to not using them. For the nanoXML case, the differences between the time and bare models were statistically significant (according to the Kruskal–Wallis test) with a confidence interval of more than 99.99%. For the rest of the subject applications, we observed pronounced practical differences; time models (compared to bare models) provided better diagnosis reports for 43% and 53% of the failures on XMLsecurity and ant, respectively.

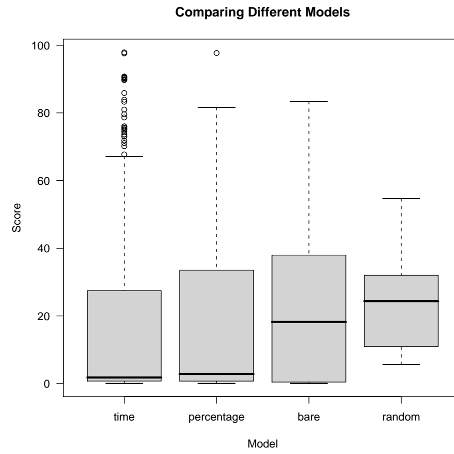
In almost all the cases where the time models performed better than their bare counterparts, we observed that the faults didn't cause observable changes in the caller-callee relations; the failing and passing runs exposed similar caller-callee relations. However, the faults affected what methods exercised in their body, which was captured by the time models, but not by the bare models.

Comparing time models to percentage models on the other hand revealed that the percentage models are almost always as good as the time models. We created the percentage models as a means of factoring out the unwanted effects of underlying platforms by normalizing the time measurements. Although in these experiments we used a single platform, our results suggest that if the normalization idea works in practice, then the percentage models can safely be used to localize faults across different platforms.

Table 3 presents some performance statistics about the experiments. The columns of this table depict the subject application used, the average runtime overhead of collecting and persisting time spectrum for a run, the average number of method invocations encountered in passing and failing runs, the average time needed to sample data tables, the average time needed to create observed behavior model for a method, and the average time needed to diagnose a failure, respectively. Note that model creation times are given per method whereas the diagnosing times are given per failure. Furthermore, all the time measurements are in seconds.

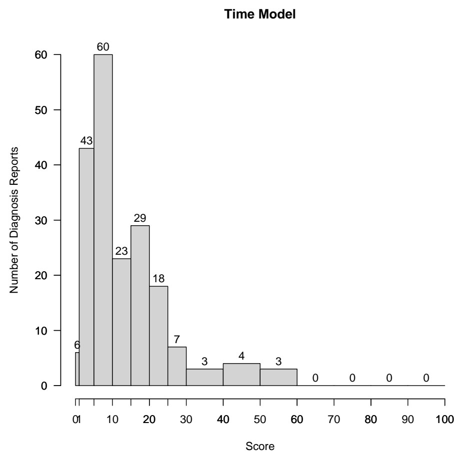


(a) Histogram for the scores of diagnoses obtained from the time model.

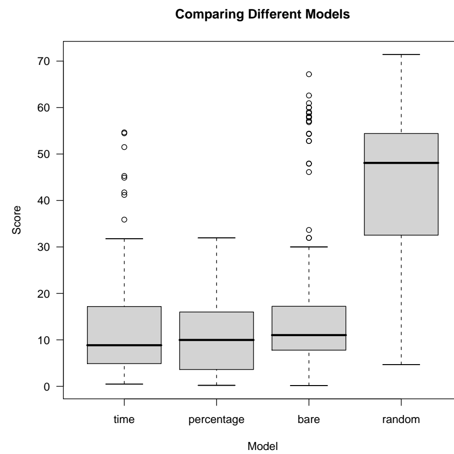


(b) Comparing different models.

Figure 3: The results of experiments on nanoXML.

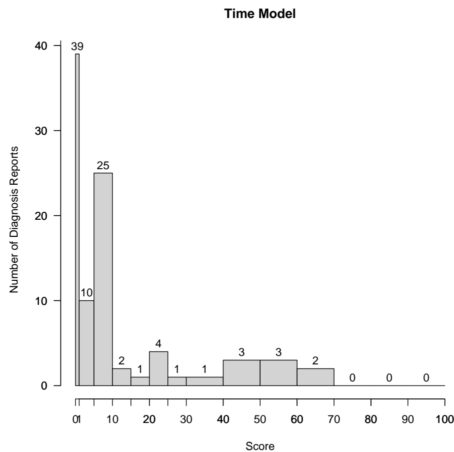


(a) Histogram for the scores of diagnoses obtained from the time model.

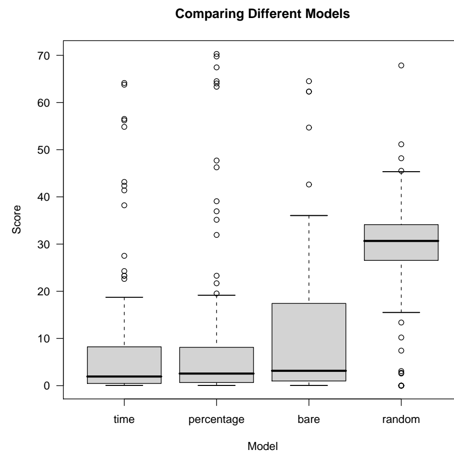


(b) Comparing different models.

Figure 4: The results of experiments on XMLsecurity.



(a) Histogram for the scores of diagnoses obtained from the time model.



(b) Comparing different models.

Figure 5: The results of experiments on ant.

Although in our instrumentation tool we tried to be as unobtrusive as possible by adjusting system clock to factor out the cost of instrumentation from measurements, we didn't pay much attention to the overall performance of the tool. Therefore, in order to realistically assess the runtime overhead of collecting time spectra, we decided to use a commercial tool, called JProfiler [1]. With the right configuration, this tool collected a proper superset of the information needed by TWT. The runtime overheads given in Table 3 were obtained from JProfiler and expressed as a fraction relative to the execution time of the original program.

The overhead cost of collecting time spectra was reasonable; around 55% of the original execution times on average. Although the test cases used in our experiments had a short lifespan (often around couple of seconds) and the overhead ratios may vary for long-living test cases, we believe that these results are encouraging. The cost of sampling and creating observed behavior models per method was affordable; 0.70 and 3.44 seconds on average, respectively. Once the models were created, the cost of diagnosing a single failing run was practical; around 60.77, 74.16, and 422.36 seconds on average for nanoXML, XMLsecurity, and ant, respectively. Note that the differences between the number of method invocations observed in passing and failing runs across the subject applications were reflected on the performance. For example, diagnosing a failing run on ant took longer, since failing runs on ant exercised more method invocations on average.

4.6 Threats to Validity

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

One threat concerns the representativeness of the subject applications used in the experiments. Although they are all real-life applications, they only represent three data points. A related threat concerns the representativeness of the seeded faults. Although these faults were taken from an independent repository and they were seeded by experienced developers, they are still hand-seeded faults. On a related note, the faulty versions we used had a single fault only.

Another threat is that a majority of the test cases used in the experiments designed for unit testing which is only one type of testing. One characteristic of the unit test cases is that they often have a short lifespan. Long running test cases may introduce some scalability challenges especially in terms of the amount of information that needs to be collected from the executions. Handling such situation may require replacing some of our offline analyses, such as sampling, with their online counterparts.

While these issues pose no theoretical problems, there is clearly a need to apply TWT to larger applications with real faults in future work to understand how well it scales.

5. CONCLUDING REMARKS

In this paper we present an automatic fault localization technique, called *Time Will Tell* (TWT). TWT leverages time spectra as abstractions for program executions. Time spectra have been traditionally used for performance debugging. By contrast, here we use them for functional correctness debugging by identifying pieces of program code that take a "suspicious" amount of time to execute.

The proposed approach can be summarized as follows: Time spectra are collected from passing and failing runs, observed behavior models are created using the time spectra collected from passing runs, and deviations from these models in failing runs are identified and scored as potential causes of failures.

We conducted several experiments on three open source real-life applications to evaluate the TWT approach. Despite the external threats to validity discussed, we believe that the results of our experiments support our basic hypotheses: 1) execution times can be indicative of many things that happen in executions, e.g., path taken, branches covered, methods invoked etc., 2) by using time spectra important behavioral patterns in executions can be detected, 3) these patterns can effectively be captured in "observed behavior models", and 4) deviations from these models in failing runs can give clues about the location of faults.

We believe that this line of research is novel and interesting. As a next step, we are currently in the process of running large scale comparative studies to evaluate various well-known automatic fault localization techniques.

6. REFERENCES

[1] ej-technologies. http://www.ej-technologies.com.

[2] The Rstat Project. http://cran.r-project.org.

[3] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *ISSRE '95*, pages 143–151, 1995.

[4] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP'05*, pages 528–550, 2005.

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

[6] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI'00*, pages 1–16, 2000.

[7] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02*, pages 291–301, 2002.

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

[9] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02*, pages 467–477, 2002.

[11] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03*, pages 141–154, 2003.

[12] C. Mateis, M. Stumptner, D. Wieland, and F. Wotawa. Model-based debugging of java programs. In *AADEBUG '00*, 2000.

[13] H. Pan and E. Spafford. Heuristics for automatic localization of software faults. Technical Report SERC-TR-116-P, Purdue University, 1992.

[14] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE '03*, pages 30–39, 2003.

[15] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *Softw. Eng. Notes*, 22(6):432–449, 1997.

[16] P. N. Tan, M. Steinbach, and V. Kumar. *Introduction to data mining*. Addison Wesley, 2006.

[17] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *SIGSOFT Softw. Eng. Notes*, 29(4):45–54, 2004.

[18] C. Yilmaz and C. Williams. An automated model-based debugging approach. In *ASE '07*, pages 174–183, 2007.

[19] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10*, pages 1–10, 2002.