Instrumenting Where it Hurts— An Automatic Concurrent Debugging Technique*

Rachel Tzoref
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
rachelt@il.ibm.com

Shmuel Ur
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
ur@il.ibm.com

Elad Yom-Tov
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
yomtov@il.ibm.com

ABSTRACT

As concurrent and distributive applications are becoming more common and debugging such applications is very difficult, practical tools for automatic debugging of concurrent applications are in demand. In previous work, we applied automatic debugging to noise-based testing of concurrent programs. The idea of noise-based testing is to increase the probability of observing the bugs by adding, using instrumentation, timing "noise" to the execution of the program. The technique of finding a small subset of points that causes the bug to manifest can be used as an automatic debugging technique. Previously, we showed that Delta Debugging can be used to pinpoint the bug location on some small programs.

In the work reported in this paper, we create and evaluate two algorithms for automatically pinpointing program locations that are in the vicinity of the bugs on a number of industrial programs. We discovered that the Delta Debugging algorithms do not scale due to the non-monotonic nature of the concurrent debugging problem. Instead we decided to try a machine learning feature selection algorithm. The idea is to consider each instrumentation point as a feature, execute the program many times with different instrumentations, and correlate the features (instrumentation points) with the executions in which the bug was revealed. This idea works very well when the bug is very hard to reveal using instrumentation, correlating to the case when a very specific timing window is needed to reveal the bug. However, in the more common case, when the bugs are easy to find using instrumentation (i.e., instrumentation on many subsets finds the bugs), the correlation between the bug location and instrumentation points ranked high by the feature selection algorithm is not high enough. We show that for these cases, the important value is not the absolute value of the evaluation of the feature but the derivative of that value along the program execution path.

As a number of groups expressed interest in this research, we built an open infrastructure for automatic debugging algorithms for concurrent applications, based on noise injection based concurrent testing using instrumentation. The infrastructure is described in this paper.

Categories and Subject Descriptors

D.2.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*
; D.2.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*

General Terms

Verification, Algorithms

Keywords

Concurrency, Debugging, Feature Selection

1. INTRODUCTION

The increasing popularity of concurrent programming—on the Internet as well as on the server side—has brought the issue of concurrent defect analysis to the forefront. This is true also on the client side as almost every CPU available these days is multi-core, so applications have to become concurrent to take advantage of it. Concurrent defects, such as unintentional race conditions or deadlocks, are difficult and expensive to uncover and analyze, and such faults often escape to the field.

One reason for this difficulty is that the number of possible interleavings is huge, and it is not practical to try them all. Only a few of the interleavings actually produce concurrent faults; thus, the probability of producing one is very low. Since the scheduler is deterministic, executing the same tests many times does not help, because the same interleaving is usually created. The problem of testing multi-threaded programs is compounded by the fact that tests that reveal a concurrent fault in the field or in a stress test are usually long and run under different environmental conditions. As a result, such tests are not necessarily repeatable, and when a fault is detected, much effort must be invested in recreating the conditions under which it occurred.

*This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

Much research has been done on testing multi-threaded programs. Research has examined detecting data races [19], [20], [14], replaying in several distributed and concurrent contexts [4], static analysis [22], [13], [7], and the problem of generating different interleavings for the purpose of revealing concurrent faults [8] [23]. Model checking [21], coverage analysis [17] [8], and cloning [12] are used to improve testing in this domain.

The work described in this paper is part of SHADOWS, an EU project whose goal is to create technology for self-healing. For intermittent bugs whose manifestation depends on specific interleaving, it may be possible to automatically detect and remove the bug-causing interleaving. The first part of this ambitious goal is to automatically pinpoint the location in the program that is the cause of the bug. There are a number of characteristics of the concurrent domain, elaborated in the paper, which cause the problem of finding the root cause to be much harder than in the sequential domain. In a previous paper [6] we showed, on small programs, that the combination of a delta debugging (DD) technique and testing via noise generation yields a concurrent debugging technique. The technique we tried was similar to the techniques expressed in a thread of papers on DD [5], [26], [27]. In these papers, a set of program changes is used to induce a bug, with the goal of finding a minimal subset. The set of changes comes from the difference between two program versions: the old one that works and the new one that contains a bug. In our domain, where the problem is to find the minimal instrumentation that causes the bug, the set of changes that induces bugs is calculated using instrumentation testing technology.

The DD search algorithm assumes the problem to be monotonic; if a set of instrumentations reveals the bug, then any superset of this set also reveals the bug. In [1] and [6] it was shown that concurrent programs are not necessarily monotonic but it was not established if real programs possess this property. One of our first findings was that real programs are not monotonic, and that therefore DD-based search algorithms do not work. In Section 2 we explain why DD algorithms do not work.

It is clear to us that creating good search algorithms that find the root causes of bugs in concurrent programs is a very interesting research topic with clear industrial applications. As we have a number of partners in our research who are interested in trying out different search algorithms, we set out to create an open infrastructure that enables researchers in the field to write search algorithms that can search on the instrumentation and can take into account runtime information such as that found by race detection tools. The infrastructure for search algorithms is described in Section 3, and is available for external users by contacting *ur@il.ibm.com*.

For our first search algorithm, we used feature selection from the domain of machine learning, treating each instrumentation as a feature. We ran the program with many subsets and looked for correlations between features and program failures. Then we tried to see if a few of the features with the best scores reveal the bugs. We tried it on a very difficult problem for which it is very hard to make the bug manifest (see Section 4) and it worked very well.

We then tried it out on programs for which it is easy to find a small subset that finds the bug, as many small subsets find the bug, but which are hard to automatically debug and find the root cause. Feature selection alone is insufficient, as instrumentations that cause the bug to manifest are not necessarily located near the root cause of the bug and therefore the locations may not be helpful to the programmers. Section 5 details our elaboration of the feature selection algorithm to pinpoint the location of the bug.

The contributions of this paper are as follows:

- Show that using feature selection when searching for the root cause of hard-to-find bugs works well.
- Create a new algorithm for pinpointing root causes of bugs when many instrumentations reveal the bug.
- Show on real programs that the search problem for instrumentations that manifest the bug is not monotonic.
- Create an open infrastructure on which search algorithms can be evaluated.

2. BACKGROUND

Debugging is one of the most common activities in the development of computer programs and much thought has been given to its automation. In concurrent programming, which is one of the domains studied, the same test may sometimes fail and sometimes succeed. The testing of multi-threaded applications by inserting schedule-modifying statements ("noise"), such as sleep and yield, has been studied in [8], [23]. This is an effective technique for discovering whether a bug exists, but it does not look for the root cause of the bug.

Studies on bug patterns in multi-threaded programs [11], [16] reveal that most bug patterns can be exposed using very few instrumentation points, and sometimes only one. However, the instrumented noise must be non-deterministic, i.e., noise that does not impact the interleaving every time it is executed. This requirement means the testing checks whether the noise is in the correct place and is itself non-deterministic. Sometimes, even if the noise is in the correct place, it fails to produce the bug. Therefore, the test succeeds in finding the bug if it finds the bug with a sufficient probability.

Once a test finds a bug, the goal of automatic debugging is to find a minimal subset of the changes required to produce the bug. Finding such a minimal subset is useful for understanding the core requirement of this bug.

2.1 Using Delta Debugging for Automatic Debugging

One approach for automatic debugging is Delta Debugging (DD) [5], [26], [27], a well-known algorithm for searching for sets of changes. Given a set of program changes that induces a bug, the algorithm searches for a minimal subset that still induces the bug. The set of changes comes from the difference between two program versions: the old one that works, and the new one that contains a bug. An example of this can be seen in [26], where two versions of the program exist—one that works and another that has a bug. The difference between these programs is 178,000 lines of code. DD automatically pinpointed the single line that caused the bug. Similar ideas are applied in another domain where the test is reduced to the essential part required to display the bug [27]. The algorithms used in these applications can be found in [26].

```
   T1                  T2

1) x=1              1) x=1
2) if (x!=0)        2) x=2
3)    y = 1/x       3) x=3
                    4) x=0
                    5) x=4
```

Figure 1: A non-monotonic program

When applied to concurrent bugs, the set of changes that induces bugs is calculated automatically using testing instrumentation technology and is not related to user program changes. In [5], DD found places in the interleaving that were indicative of failure. These locations were identified using a replay tool called DEJAVU, used on a special deterministic JVM. In a previous paper [6], we applied a new version of the DD algorithm to code from the concurrent bugs benchmark [10]. Using the DD approach and an AspectJ-based implementation of the instrumentation, we were able to pinpoint the bug location in several examples. However, the DD approach assumes monotonicity. A set of changes is monotonic if, for every set that finds bugs, all its supersets also find bugs [26]. The existence of interrelations between instrumentations may cause the problem to be non-monotonic. In the case of concurrent bugs, studies have shown that these interrelations may exist [1], i.e., inserting too much noise may mask the bug.

The example in Figure 1 illustrates the problem of non-monotonicity in multi-threaded programs. The execution starts with thread T1. For the bug to manifest itself, a thread switch must occur after T1.2, and another switch must occur after T2.4. So {T1.2, T2.4} is a minimal set that induces the bug. Its superset {T1.1, T2.2, T1.2, T2.4} also induces the bug and can be given as an initial set to DD. However, if we remove T2.2 from this set, the bug does not occur, even though the set {T1.1, T2.4, T1.2} is still a superset of the minimal set. Therefore, when we ran DD on this program, it was not able to find the minimal solution.

An important question is whether real applications are monotonic. One of our first findings was that, in fact, real applications are not monotonic, and thus the DD approach does not work for them. In Section 4, we describe such a real application. These examples led us to develop a new approach for automatic debugging of concurrent bugs, which does not assume monotonicity of the problem. The new approach is based on presenting the problem as a feature selection problem from the domain of pattern classification, and is described in Section 4. First, some background on feature selection is presented in Subsection 2.2.

2.2 Feature Selection

Feature selection is the process of finding a minimal subset of indicators that best represent the data [25]. It is well known in the machine learning field that noisy or irrelevant features (indicators) are detrimental to pattern recognition in that they cause poor generalization, increase the computational complexity, and require many training samples to reach a given accuracy [2]. In the case of non-monotonic programs, irrelevant features may mask the bug or cause it to appear with a very low probability, thereby requiring many runs of the program to elicit the bug.

A brute force approach to feature selection, i.e., trying all possible feature combinations, is possible only for very small feature sets. For example, testing all possible combinations for 100 features requires testing approximately 10^{30} configurations. However, 100 instrumentation points is well within the range of medium sized programs. Therefore, other methods should be used for selecting the best features.

In general, feature selection methods can be divided into three classes [2]: wrapper methods, where feature selection is performed around (and with) a given classification algorithm; embedded methods, where feature selection is embedded within a classification algorithm; and filter methods, where features are selected for classification independently of the classification algorithm.

In this work, we applied a filter method for selecting the most relevant features in the sample, as explained in Subsection 4.1. We chose such methods so as to create the smallest dependence between the feature selection stage and other stages of the debugging process.

3. EXPERIMENTAL SETUP

The following actions are required by a typical search algorithm whose goal is to find a minimal instrumentation that pinpoints the location of a concurrent bug:

1. Extract the set S of all possible locations of the program under test where we may want to add noise.
2. Instrument noise at any subset $s \in S$.
3. Run the instrumented program.
4. Determine if the executed test displays a concurrent bug.

As an infrastructure for our automatic debugging, we have implemented a generic search algorithm that provides these capabilities. The generic algorithm is written in Java and implements two functions: $getAllPoints()$, which returns the set of all possible locations of the program under test; and $runIteration(Set\ locations, int\ numRuns)$, which instruments the program in the given locations, runs it $numRuns$ times, and returns the number of runs in which a bug was found. Note that we enable running multiple times with the same set of locations since the runs can be non-deterministic. Thus, determining whether a set of locations finds a bug may require several runs. The algorithm is designed as an open architecture, meaning that external users can implement their own search algorithms that derive the functionality of the generic search algorithm. Using the open architecture, we implemented two variants of the DD algorithm. The first is the DD algorithm described in [26], and the second is the new DD version from [6].

Our generic algorithm is implemented on top of ConTest [8], a tool developed in IBM for testing, debugging, and measuring the coverage of concurrent Java programs. The basic principle behind ConTest is quite simple. The instrumentation stage transforms the class files, injecting calls to ConTest runtime functions at selected places. At runtime, ConTest sometimes tries to cause context switches in these places. The selected places are those whose relative order among the threads can impact the result: entrance and exit

from synchronized blocks, access to shared variables, etc. Context switches are attempted by calling methods such as yield() or sleep(). The decisions are random so that different interleavings are attempted at each run. Heuristics are used to try to reveal typical bugs. Note that there are no false alarms; all interleavings that occur with ConTest are legal as far as the JVM rules are concerned.

The generic algorithm uses ConTest to extract the set of possible locations to which noise can be added, and to instrument and run the program with a given subset of locations. ConTest does not know whether a bug has actually been revealed; it has no notion of how the program is expected to behave. The user should provide a test to the generic algorithm, indicating which program run is considered correct and which indicates a bug.

In addition to the open architecture for search algorithms for minimal noise, ConTest implements a Listener Architecture, which is a library that can be used by people writing tools that need to hook into given programs under test, for example, race detectors. ConTest Listeners provide API for doing things when certain types of events happen in the program under test. The events that can be listened to include entry and exit to synchronized blocks, calls to methods such as wait() and join(), thread start and finish, and access to member fields. Thus, the combination of architectures enables external users of the tool to use dynamic information when implementing a search algorithm. For example, one can implement a race detector using the listener architecture, and use its output as guidance to the search algorithm.

4. FEATURE SELECTION BASED AUTOMATIC DEBUGGING

As explained in Subsection 2.1, concurrent programs may exhibit non-monotonic behavior. We first encountered the non-monotonicity problem in a real application when running variants of DD on a web crawler algorithm, embedded in an IBM product. The skeleton of the algorithm has 19 classes and 1200 lines of code. In [8] it is described how ConTest found an unknown race condition in the algorithm. The fault was a null pointer exception, caused by the following code: $if(connection! = null)$ $connection.setStopFlag()$. If the connection variable is not null and then a context switch occurs, the connection variable might be set to null by another thread, before $connection.setStopFlag()$ is executed. If this happens, a null pointer exception is taken. An even more insidious manifestation of the bug, which causes a security risk, is when the connection variable is set to a non-null value, since in this case, the wrong connection is stopped. To fix this bug, the above statement and all other accesses to the connection variable should be executed within an appropriate synchronized block.

The race condition is very rarely manifest, since it occurs in a very small percentage of all possible interleavings. Without instrumentation we have never seen the bug. When instrumenting all possible locations, it appears on average only in 1 out of 750 runs. Since the bug is so rare, we set a probability of 1% (1 out of 100 runs) as a sufficient probability for a test to find the bug. In this example, we already knew the root cause of the bug: the line $if(connection! = null)$ $connection.setStopFlag()$, and another line executed by another thread that sets the connection to null. When instrumenting only the two points corresponding to these two lines, the bug appears in 10 out of 100 runs.

When we ran DD on the web crawler example, it failed to find a minimal instrumentation, due to the non-monotonic nature of the bug – supersets of the minimal instrumentation do not necessarily induce the bug with a sufficient probability. As a result, given a set of instrumentations that induces the bug, all the subsets tried out by DD did not induce the bug, contradicting the basic assumption of the algorithm.

Thus, a new approach is required for automatically pinpointing the location of the bug. Our approach is based on feature selection: we treat each instrumentation point as a feature, and generate samples containing subsets of the features. For each sample, we run the program instrumented only in the points of the sample, and determine whether the sample induces the bug with a sufficient probability. Once all samples are classified, we score each instrumentation point based on the correlation between its appearance in samples and whether they induce the bug. We then choose the highest scored points as locations that are likely to be correlated to the bug, and thus should be presented to the user as the probable root cause of the bug. The exact process of samples generation and the points scoring method are described next.

4.1 Sample Generation and Feature Scoring Method

As noted above, our API enables the instrumentation of the program under test in any subset of the instrumentation points. To achieve the best possible indications of the usefulness of the features, our first step was to use this functionality to create an optimal sampling of the input space.

Design of Experiments are a class of methods to actively manipulate a system in order to best elicit its different modes of behavior. In our work, we used the D-optimal design [24], a method which maximizes Fisher's information matrix.

Assume a program can be instrumented in N points and run for M experiments. Let X denote a matrix of size $N \times M$, where $x_{ij} = 1$ if and only if the j-th experiment was instrumented in point i. A D-optimal design maximizes the determinant of X: $|X' \cdot X|$.

Finding X which maximizes $|X' \cdot X|$ is a difficult computational problem. We therefore created 500 instances of a random sampling matrix X and chose the matrix that had the largest $|X' \cdot X|$.

This sampling matrix was then used as the input for running the program.

After running the programs with the given sampling matrix, each sample was marked as successful (found the bug) or unsuccessful (did not find the bug). Our goal was then to find those points that were most likely to find the bug. This is the feature selection stage of the algorithm.

We used the likelihood ratio test [18] scoring function to score the features. Let $P(Success|X_i)$ denote the sample probability that tests which instrumented the i-th instrumentation point will result in a bug, and $P(!Success|X_i)$ denote the same for the case where the bug is not found. We assign each point the score:

$$Score(i) = P(Success|X_i)/P(!Success|X_i)$$

The points with the highest score are assumed to be most indicative for the location of the bug. Such a scoring func-

tion assumes no correlation between instrumentation points, that is, if a bug requires that two points be instrumented in a correlated fashion, it may not be found using this scoring function. Our current experience is that, in practice, this scoring function is sufficient, but we are currently examining additional scoring functions that do take such interaction into account.

Once the features are weighted, it is necessary to select the features most indicative of the bug. In the current paper we begin by running the program with the highest ranking point, then the two highest points, etc., until the bug appears with high probability. The set of instrumentation points which caused the bug to appear in this probability is then reported to the user as the most indicative points for locating the bug.

Tarantula [15] uses a similar metric for ranking suspicious lines of code in programs. Based on their appearance in failed and successful runs of a program, Tarantula assigns a value called hue to each line. The hue is computed as:

$$hue(i) = P(Success|X_i)/(P(Success|X_i)+P(!Success|X_i))$$

This scoring is a heuristic, similar to the likelihood score. The authors suggest examining lines according to their hue, starting from the lowest values of the hue. The authors do not address the issue of how to sample the program, assuming instead that sampling is performed in an independent manner by users.

The authors of [28] developed a method for sampling programs and identifying probable bug locations in single-thread programs. Their instrumentation is performed using assertions placed in the code, which are randomly sampled at run time. This implies that they require a diverse sample of runs in order to execute all the assertions. The authors then use a utility function to build a classifier whose goal is to correctly predict the outcome of runs (success of failure) based on the outcomes of the assertions. The weights of the utility function then serve as indicators for the location of the bug. Their debugging process requires tuning the parameters of the classifier using a training set and then finding the weights of the classifier using an optimization algorithm. This method, while effective for small programs, seems incur a high computational cost and requires manually setting the assertions in the code.

GeneticFinder [9] is a noise-maker that uses a genetic algorithm as a search method, with the goal of increasing the probability of the bug manifestation, and minimizing the set of variables and program locations on which noise is made. The search is performed at run-time, i.e., all program locations are instrumented, and at each point it is decided during run-time whether to apply noise. From our experience, instrumentation alone can change the scheduling of the program. Thus, the approach of partial instrumentation is much more accurate.

4.2 Automatically Debugging A Web Crawler

In the web crawler example (described at the beginning of this section) there were 314 possible instrumentation points. We decided to generate 5000 samples. This number was determined according to our available computing resources and total run time estimation. The average number of points in each sample was 157. The program was run with each sample 100 times. A sample was considered to induce the bug if the bug appeared in at least one run out of the 100 runs. We ran the samples in parallel on 10 Linux machines. The average time for a single program run was 10 seconds.

There are two instrumentation points corresponding to the root cause of the bug. The first point is the location where the connection is set to null, and the second corresponds to $if(connection! = null)$. These two points affect the timing between the execution of the two statements. After running the first 500 samples for about 14 hours, the two highest scoring points were exactly the two points of the root cause of the bug, and they remained the highest scoring points throughout the experiment, which took 6 days. Thus, using feature selection, we were able to automatically pinpoint the location of the bug.

In fact, when we started the experiments on the web crawler example, we detected an additional bug, in the form of a "max stack exceeded" exception. Similarly to the first bug, the second bug also appeared only for some subsets of instrumentations. However, if it was induced by a certain subset, then it was consistently induced. We did not know at first what the root cause of the bug was, and decided to include it in our experiments, i.e., a sample was classified as inducing a bug if it induced either of the two bugs, without considering which of them was induced. The scoring of the points also did not consider whether the first or the second bug was found. As reported above, the fact that a second bug appeared in the results did not prevent us from finding the root cause of the first bug. After locating the first bug, we wanted to find the root cause of the second bug, based on the samples we already ran. Therefore, we removed all samples in which the root cause of the first bug appeared, and scored the points according to the remaining samples. The results were again very accurate: the first point was scored 2.03, significantly higher than the other points (the next 10 points score between 1.48 and 1.32). When instrumenting with this point alone, the second bug appeared. Using the single instrumentation point, the debugging process was very easy. The bug was in the instrumentation itself—not enough stack space was allocated when instrumenting this point. This bug was sometimes masked by additional instrumentations, since they allocated sufficient stack space for the run.

Of the 5000 samples, 777 induced a bug. 461 induced the first bug (with an average probability of 10%) and 316 induced the second bug (with a probability of 100%). We again emphasize that the process of classifying the samples and the scoring of the points did not take into account which of the two bugs was found, and yet our technique was able to pinpoint the location of both bugs. The fact that with partial instrumentation the first bug was found with a higher probability than with full instrumentation (in which the probability was 1 out of 750 runs), corresponds to the observation that too much noise may mask the bug [1].

We return to the example in Figure 1. We also ran our technique on a program that contains the two threads appearing in the example (the main thread was omitted from Figure 1 for clarity). The program contains 32 possible instrumentation points. The threshold probability of finding the bug was set to 5% (1 out of 20). 1000 samples were generated, with an average number of 16 points in each sample. The total run time was 20 minutes. 222 samples out of 1000 found the bug, with an average probability of 10.4%. The results were again accurate: the two highest scores were of the points corresponding to T2.4 and T1.3, respectively, pin-

```
   Wait Thread                  Notify Thread

1) x=1                       1) x=2
2) x=1                       2) x=2
3) x=1                       3) x=2
4) x=1                       4) x=2
5) x=1                       5) x=2
6) synchronized (lock){      6) synchronized (lock){
7)    lock.wait()            7)    lock.notify()
8) }                         8) }
9) x=1                       9) x=2
```

Figure 2: A lost notify bug

pointing the location of the bug. When instrumenting only these two points, the bug was found with a probability of 90%.

5. ZOOMING IN ON BUG LOCATIONS

The examples described so far exhibit rare bugs, i.e., a low percentage of instrumentation subsets induce the bug. In the web crawler example, 9% of the generated samples induce the first bug, and only 6% induce the second bug. In the example in Figure 1, 22% of the samples induced the bug. An interesting question is whether our approach will work well when many subsets of instrumentations induce the bug. Consider, for example, the simple program in Figure 2. The execution starts with the wait thread. This program contains a deadlock that is manifest when the call to notify() precedes the call to wait(). The main thread was omitted for clarity, as were the *try* and *catch* blocks for the call to wait(). Any instrumentation that delays the call to wait() will induce the bug. Therefore, many subsets of instrumentations induce the bug, including instrumentations remote from the bug location, making the goal of finding an instrumentation that pinpoints the location of the bug more challenging.

The program contains 63 instrumentation points. We generated 5000 samples, with an average number of 31 points in each sample. Each sample was run 20 times. The total runtime was 3 hours. The average number of times each sample found the bug was 5. We did not predetermine the threshold probability for finding the bug, but rather used the average probability in which the bug was found (including samples that did not find the bug at all) as the threshold, and classified the samples only after all of them were run. Our experiments show that using the average probability as the threshold probability is significantly more accurate than using a predetermined threshold probability. 3632 samples (72%) found the bug at least once, and 2079 (41%) found it at least 5 times and were classified as inducing the bug, confirming the fact that the bug is common, i.e., easily induced by instrumentation (without instrumentation we have never seen the bug manifest).

When we look at the feature selection scores of the points, they do not correlate well with the location of the bug. The problem is that many points receive high scores, since they induce the bug, but they are not necessarily in the vicinity of the bug. One such point is the location where the main starts the wait thread. Naturally, delaying the start of the wait thread induces the bug. However, this point does not contain sufficient information to explain the bug. We also observed in additional examples that the start() call of threads that are involved in concurrent bugs usually receives high feature selection scores. Our policy is to consider these type of points as indicating the problematic threads, but not to consider them when trying to pinpoint the exact location of the bug. Additional points that receive high feature selection scores but are not in the vicinity of the bug correspond to lines 1-5 of the wait thread. Naturally, delaying their execution also delays obtaining the lock by the wait thread, thus inducing the bug.

The conclusion is that for bugs that are found by many subsets of instrumentations, the feature selection score of single instrumentation points is not sufficient. Thus, an enhancement to our technique is required. The basic idea of our enhancement is to look not at the absolute values of the points scores, but at the derivative of the scores along the program execution path. For example, in the program in Figure 2, we expect high scores for the points corresponding to lines 1-5 of the wait thread, and then a drop in the score for the point corresponding to obtaining the lock in line 6 of the wait thread, since if notify() has not yet been called and then the lock is obtained by the wait thread, the call to wait() cannot be delayed to follow the call to notify(). Similarly, we expect a rise in the score of the point corresponding to obtaining the lock in line 6 of the notify thread, since the points corresponding to the previous lines delay the call to notify() and hence mask the bug.

For the general case, our enhanced technique works as follows: we look at the control flow graph of the program, mark each node of the graph with the feature selection score of the corresponding instrumentation point, and search for pairs of consecutive nodes with maximal decrease and increase in the score. In fact, instead of searching for pairs of nodes, we can simply mark each edge of the graph with the difference between the scores of its entry and exit nodes, and look for the edges with maximal and minimal values (indicating maximal drop and maximal rise in the score).

As a good approximation of the control flow graph of the program, we used the graph induced by the concurrent event pairs coverage provided by ConTest. When this coverage information is requested, ConTest lists pairs of concurrent event program locations that appeared consecutively in a run, and whether they were performed by the same thread. We ran the program several times with ConTest to induce different interleavings, and built the graph from the pairs of locations that were executed by the same thread.

The basic intuition behind the enhancement of our technique is that most consecutive points in the program have a similar effect on inducing the bug, except for the points that are clearly correlated to the root cause of the bug. If our intuition is correct, we expect the difference in score between consecutive points along the program execution graph to be usually around zero, unless this is the vicinity of the bug. We expect the distribution of the difference score between arbitrary points to be less concentrated around zero. The distribution of the difference score for pairs of consecutive points (in black), and for arbitrary pairs of the program points (in grey) is depicted in Figures 3 and 4, for the program in Figure 2, and for a real application that will

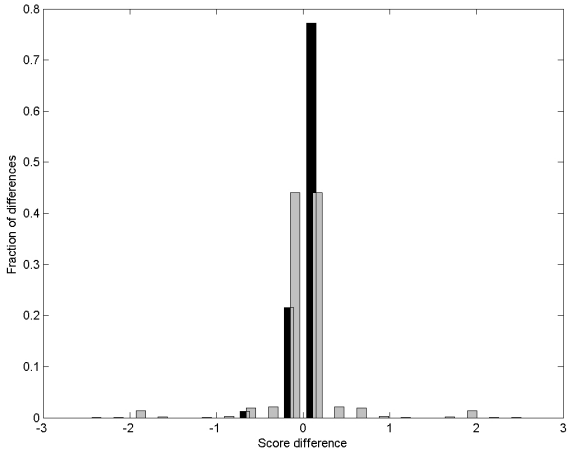

Figure 3: Distribution of difference score for lost notify. Consecutive points in black, others in grey.

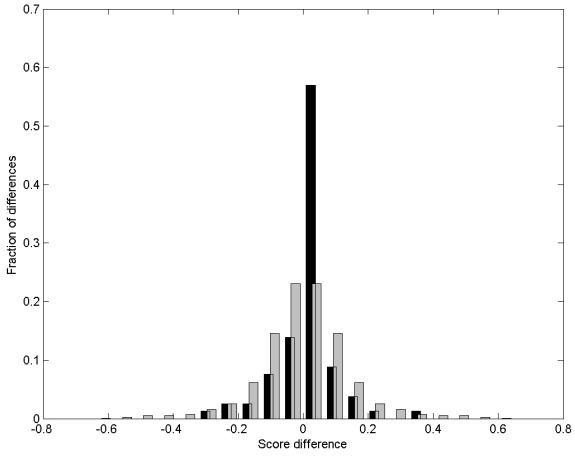

Figure 4: Distribution of difference score for server loop. Consecutive points in black, others in grey.

be described in Subsection 5.1. As we can see, the distributions are statistically significantly different ($p < 1e-4$, two-sample Kolmogorov-Smirnov goodness-of-fit hypothesis test), and support our hypothesis.

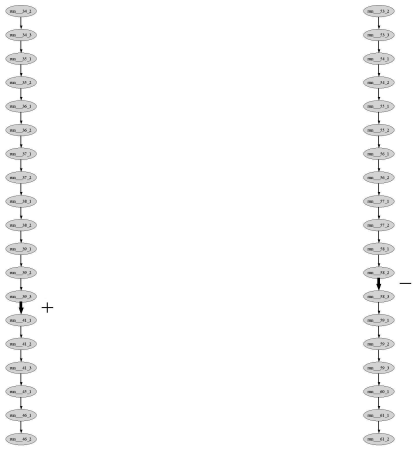

Figure 5: Difference score graph for lost notify bug

The difference score graph for the two threads from Figure 2 is depicted in Figure 5. Each node represents an instrumentation point of the program. There is a directed edge from node a to node b if and only if the pair of instrumentation points (a, b) appears in the concurrent event coverage produced by ConTest, i.e., the concurrent events in points a and b were executed consecutively by a thread in the program. The maximal and minimal edges in the graph are marked with thick lines. The maximal edge (marked with $+$) connects between the point where the lock variable is read by the wait thread and the point where it is obtained (line 6 of the wait thread). The minimal edge (marked with $-$) connects between the point where the lock variable is read by the notify thread and the point where it is obtained (line 6 of the notify thread). These two locations pinpoint the bug location.

5.1 Automatically Debugging A Server Loop

We ran our enhanced technique on a real application that implements a server loop and contains a deadlock due to a lost notify. The server loop performs database transactions according to clients requests. It loops on a global variable *stop*, waiting indefinitely for transaction requests from clients. The server can be externally shut down by calling a function that sets the value of *stop* and notifies the server. The shutdown mechanism is depicted in Figure 6. It contains a deadlock, since the reading and writing of *stop* are performed outside the synchronized block. Thus, if the server reads the value of *stop* and next the shutdown function is executed, then the call to notify() precedes the call to wait(), and the server enters a deadlock. After extracting the business code that is not related to concurrency [3], the program contains 72 possible instrumentation points.

The deadlock is easily induced. Random sampling of 2 points induces the bug in 25% of the samples, and random sampling of 3 points induces the bug in 33%. Thus, many small subsets of instrumentations reveal the bug. Despite this fact, DD was unable to find a minimal instrumentation.

```
   Server                          Shutdown Function

1) while (! stop){           1) stop = true
2)   business code              2) synchronized (lock){
3)   synchronized (lock){       3)   lock.notify()
4)     if (request){             4) }
5)       business code
6)     }
7)     else{
8)       lock.wait()
9)     }
10)  }
11)}
```

Figure 6: Skeleton of shutdown mechanism for server loop

The reason is again the non-monotonic nature of the bug. For example, if we instrument the point corresponding to reading the lock variable in line 3 of the server code, just before obtaining it, then we delay the call to wait() and therefore the bug is induced with a very high probability. However, if in addition we instrument the point corresponding to reading the value of *stop* in line 1 of the server code, then the probability of inducing the bug drops to almost 0. The reason is as follows: the wait() is still delayed due to the instrumentation in line 3, but so is the reading of *stop* by the server. Thus, in most interleavings, the shutdown function sets *stop* to true before it is read by the server, the server exits the loop, and the deadlock is avoided.

For the server loop example we generated 5000 samples, with an average number of 36 points in each sample. Each sample was run 20 times. The total runtime was 4 hours, running in parallel on 10 Linux machines. The average number of times that each sample found the bug was 1.6. Thus, the threshold probability for finding the bug was set to 5% (1 out of 20 runs). The bug was found by 3899 samples (78%), confirming the fact that the bug is common.

The difference score graph for the server loop, stripped of irrelevant business code, is depicted in Figure 7. It contains three threads, corresponding to the server thread, client thread, and main thread. For clarity, some points that are irrelevant to the explanation were omitted.

The maximal edge connects between the point where the lock variable is read by the server and the point where it is obtained. The minimal edges connect between the release of the lock by the server and reading *stop* in the loop condition, and between reading *stop* and reading the lock by the server. These locations depict the events whose timing is crucial for inducing the bug: obtaining the lock by the server should be delayed, and reading *stop* by the server should not be delayed.

We return to the web crawler example from Section 4. Its difference score graph is depicted in Figure 8. The maximal and minimal edges are marked with thick lines, and with a number denoting whether they represent the first or the second bug. Of the 400 edges in the approximated control flow graph, our technique was able to detect the few edges that pinpoint the location of the two bugs. The highest scored edge connects between setting the value of the connection variable to null and the next point, and the lowest scored edge connects between setting the value of the connection variable to null and the previous point. Another pair of edges with extreme values are before and after the statement $if(connection! = null)$. These four edges pinpoint the location of the first bug. Other edges with extreme values belong to the second bug, though the marking is less accurate (only two of the three edges are exactly before and after the problematic instrumentation point). However, an accurate difference score graph for the second bug is created when removing the samples in which the root cause of the first bug appears.

6. CONCLUSIONS AND FUTURE WORK

Concurrent, distributive, and multi-threaded applications are becoming the norm in the client as well as in the server space. The ability to find and debug concurrent bugs efficiently is becoming paramount. While there is a lot of theory in the field, most of the solutions that work well for small programs do not scale. One concurrent testing technique that scales well is that of instrumenting the application with "noise" to increase the likelihood of finding bugs. A natural debugging technique for bugs found using the "noise" testing technique is to look for small subsets of the instrumentations that cause the bugs to manifest. The intuition is that if a small set of instrumentation points can be found that causes the bug to manifest, showing the relevant locations to the developers will be useful in understanding the root cause of the bug and in fixing it.

A prominent automatic debugging algorithm that can be used on this problem is the Delta Debugging algorithm (DD). While we have shown in previous work that it can work on small programs, we were not sure that it would scale, as it requires the monotonicity property. We have shown in this paper that finding subsets of instrumentations is not monotonic and therefore DD algorithms of different flavors do not work.

We tried to find relevant instrumentations using feature selection algorithms. The feature selection algorithm proved successful in finding a small subset of the instrumentations that reveal the bug. When the bug is hard to find, i.e., very few of the small subsets reveal it, the instrumentations we found are usually in the vicinity of the bug. We showed that on a web crawler algorithm, which is part of WebSphere, the found minimal instrumentation helps locate the bug.

Many of the concurrent bugs are easily revealed using instrumentations, i.e., it is easy to find a test that reveals the bug. In this case, automatic debugging using feature selection or any other search algorithm is not as useful. The problem is that the location of the root cause of the bug may be far (in program lines) from the found minimal instrumentations and therefore showing the location of the instrumentation to the developer is of limited value. We postulated that given an event whose timing is central to revealing the bug there will be a large difference between creating the noise before the event or after it. This is in contrast to instrumenting an event, which helps in finding the bug, but so would instrumenting the previous or following events. When changing the focus from the value of events to the derivative of that value along program execution paths, we have been able to point to the root cause of the bug even when many instrumentations reveal it.

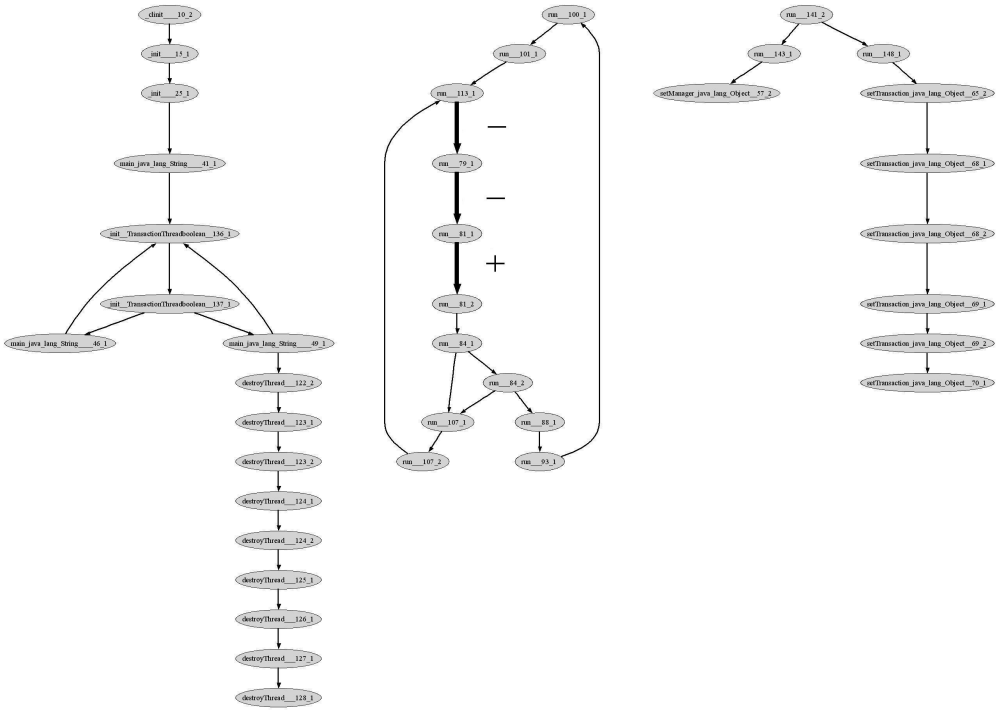

Figure 7: Difference score graph for server loop

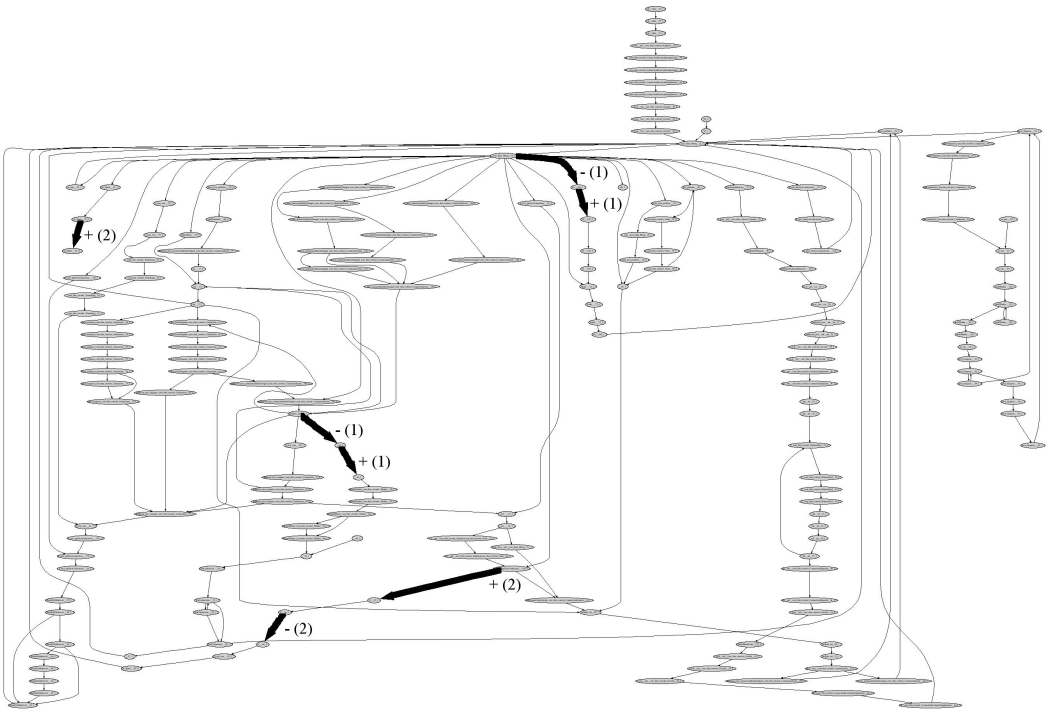

Figure 8: Difference score graph for crawler

The automation involved with automatic debugging is complex. First, there is the question of deciding if a subset of the instrumentations reveals the bug. Due to the probabilistic nature of the problem, false negatives as well as false positives are common. Therefore, every test needs to be executed a number of times and usually a threshold needs to be set to decide if the specific instrumentation reveals the bug. Second, the test needs to be prepared for the concurrent bug. If the bug, for example, is some type of deadlock, the testing environment needs to return and decide that the test failed. In addition, the manifestation of the bug may depend on the hardware used, and this issue needs to be factored out. As the creation of the infrastructure needed for testing different automatic debugging algorithms may reduce the research in this important field, we have put much thought into how to make such research as accessible as possible. To this end, we have created an open infrastructure that includes instrumentation, partial instrumentations, a number of noise-making algorithms, and testing infrastructure for running multiple tests and setting thresholds. We have also linked it to our Listener Architecture so that the search may combine compile and runtime decisions. For example, information from race detection algorithms can be used to determine the locations where instrumentation is performed. We consider the infrastructure to be one of the contributions of this research.

The work on practical automatic debugging of concurrent programs is in its infancy. Many directions can be explored for improving the search algorithms. For example, our usage of sampling and analysis methods in this article do not take into account the full possible strength of such methods. With regards to sampling, our next step will be to integrate the sampling process into the testing phase, that is, instead of generating a sampling matrix before running the program we will use active sampling methods. These methods start with a small number of sampling instances and then generate additional sampling instances based on the outcome of previous runs of the program. This should enable our methods to require much fewer samples for identifying the bugs.

Our current analysis method scores instrumentation points without taking into account any interactions between them which are more than pair-wise interactions. We plan to investigate methods which take into account higher-order interactions as well as to use the structure of the program as input to our analysis method. This should enable us to better pinpoint the location of the bugs.

Finally, using our current method only ranks points and pairs of points in order of likelihood for the location of the bug in them. This requires running the program with the highest ranked points instrumented, but it is currently an open question how many of the top-ranked points should be instrumented. We are working to solve this question so as to identify the subset of highest-ranked points which should be instrumented for the most indicative run of the program.

Another promising direction on which we are currently working is combining static and dynamic search algorithms. Each has built-in weaknesses and strengths and their combination may yield better algorithms than each by itself.

The next phase is healing concurrent bugs (or at least concealing them). This phase, on which we have done the preliminary work, is the object of SHADOWS, the EU project under which this research is funded. Healing can be achieved by identifying the location of the bug, recognizing the bug pattern and then correcting it; for example, by removing potential race using additional synchronization. Healing has to be done with care as there is a risk of inserting new bugs. Healing may be done automatically or a fix may be suggested to the developer. Concealing bugs may be done by causing the interleaving that reveals the bug not to appear; for example, by turning a section to be atomic, or by causing the interleaving that reveals the bug to be less likely.

While the work started in this paper and suggested in this section is very challenging, we expect rapid assimilation into commercial tools. There is great demand for working tools to automatically debug concurrent programs. The research shown in this paper is already translated into practice, and we see a clear path from experimental research to viable, commercial tools.

7. REFERENCES

[1] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Noise makers need to know where to be silent - producing schedules that find bugs. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*, 2006.

[2] A.L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97:245–271, 1997.

[3] H. Chockler, E. Farchi, Z. Glazberg, B. Godlin, Y. Nir-Buchbinder, and I. Rabinovitz. Formal verification of concurrent software: Two case studies. In *PADTAD '06: Proceeding of the 2006 workshop on parallel and distributed systems: Testing and debugging*, pages 11–22, 2006.

[4] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[5] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 210–220, New York, NY, USA, 2002. ACM Press.

[6] S. Copty and S. Ur. Toward automatic concurrent debugging via minimal program mutant generation with AspectJ. In *TV '06: Proceedings of Multithreading in Hardware and Software: Formal Approaches to Design and Verification*, pages 125–132, 2006.

[7] J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R., S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*. ACM Press, June 2000.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002. Also available as `http://www.research.ibm.com/journal/sj/411/-edelstein.html`.

[9] Y. Eytani. Concurrent Java test generation as a search problem. In *Proceedings of the Fifth Workshop on Runtime Verification (RV)*, volume 144(4) of Electronic Notes in Theoretical Computer Science, 2005.

[10] Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *IPDPS*, 2004.

[11] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS (PADTAD)*, page 286, 2003.

[12] A. Hartman, A. Kirshin, and K. Nagin. A test execution environment running abstract tests for distributed software. In *Proceedings of Software Engineering and Applications, SEA 2002*, 2002.

[13] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer, STTT*, 2(4), April 2000.

[14] E. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Towards integration of data-race detection in DSM systems. *Journal of Parallel and Distributed Computing. Special Issue on Software Support for Distributed Computing*, 59(2):180–203, Nov 1999.

[15] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM Press, 2005.

[16] B. Long and P. A. Strooper. A classification of concurrency failures in Java components. In *IPDPS*, page 287, 2003.

[17] Y. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. Software test coverage and reliability. Technical report, Colorado State University, 1996.

[18] A. Papoulis. *Probability, random variables, and stochastic processes*. McGraw-Hill Books, 1991.

[19] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, August 1998.

[20] S. Savage. Eraser: A dynamic race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[21] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking*, 2000.

[22] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, October 2002.

[23] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proceedings of the Second Workshop on Runtime Verification (RV)*, volume 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[24] G. Upton and I. Cook. *Oxford Dictionary of Statistics*. Oxford University Press, Oxford, UK, 2002.

[25] E. Yom-Tov. An introduction to pattern classification. In O. Bousquet, U. von Luxburg, and G. Ratsch, editors, *Advanced Lectures on Machine Learning, LNAI 3176*. Springer, 2004.

[26] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

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

[28] A. Zheng, M. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems*, 2003.