Automatic debugging of concurrent programs through active sampling of low dimensional random projections

Elad Yom-Tov, Rachel Tzoref, Shmuel Ur, Shlomo Hoory
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
{yomtov,rachelt,ur,shlomoh}@il.ibm.com

Abstract

Concurrent computer programs are fast becoming prevalent in many critical applications. Unfortunately, these programs are especially difficult to test and debug. Recently, it has been suggested that injecting random timing noise into many points within a program can assist in eliciting bugs within the program. Upon eliciting the bug, it is necessary to identify a minimal set of points that indicate the source of the bug to the programmer. In this paper, we pose this problem as an active feature selection problem. We propose an algorithm called the iterative group sampling algorithm that iteratively samples a lower dimensional projection of the program space and identifies candidate relevant points. We analyze the convergence properties of this algorithm. We test the proposed algorithm on several real-world programs and show its superior performance. Finally, we show the algorithms’ performance on a large concurrent program.

1. Introduction

Concurrent computer programs are simultaneously executed programs or program threads. Concurrent programs can be run on a single multi-threaded processor, on several processors in a single computer, or using several computers distributed across a network. The increasing popularity of concurrent programs has brought the issue of concurrent defect analysis to the forefront. This is true for both servers and client machines, since almost every CPU available these days is multi-core.

Concurrent program debugging usually refers to specific examples of bugs that are found only in concurrent programs. One example is race condition, where the output of a process is unintentionally dependent on the sequence or the timing of other events. Another example unique to concurrent programs is deadlocks, where two threads simultaneously wait for the other to produce some action, and thus can never progress. The technique described in this paper is relevant to all types of concurrent bugs.

Much research has been devoted to testing multi-threaded programs. This research has examined aspects such as detecting data races[19, 20, 13], replaying in several distributed and concurrent contexts[3], static analysis [23, 12, 5], model checking [22], coverage analysis [17, 2], and cloning [11]. Additionally, generation of different interleavings as a way of improving the efficiency of testing by revealing concurrent faults was demonstrated in [6, 24].

There are a number of distinguishing features between concurrent and sequential testing. These differences make it especially challenging to find concurrent bugs if the set of possible interleavings is huge and it is not practical to try all of them. First, only a few of the interleavings actually produce concurrent faults; thus, the probability of producing a concurrency fault can be very low. Second, under the simple conditions of unit testing, executing the same tests repeatedly to search for concurrent bugs does not help. As a result, concurrent bugs are often found in stress tests or by the customer. The problem of testing multi-threaded programs is even more costly because 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. When the conditions of the bug are finally recreated, the debugging itself may mask the bug (the observer effect).

Recently, it has been proposed [6] to elicit bugs by injecting timing noise at randomly selected points within a program, known as instrumentations. Instrumentations are created in any point in the program whose relative execution order can impact the result of the program. In general, instrumentations are done on accesses to shared variables
and on concurrent instructions such as yield and sleep. Edelstein et al. [6] contains an extensive description of the locations that need to be instrumented and the amount of noise that needs to be introduced in order to manifest the bug. Copty and Ur [4] suggested to use search methods such as delta debugging (DD) to identify a minimal set of the most indicative points where injecting noise causes the bug to appear with the highest probability. These points are assumed to be indicative of the location of the bug. Previously [25], we demonstrated how these instrumentation points can be used to pinpoint the locations of the bug.

DD is a well-known algorithm for searching for sets of changes. The DD algorithm suggested by Zeller [28] works as follows: start with two sets of instrumentation points \( c \subseteq \hat{c} \), such that the program works with \( c \) and does not work with \( \hat{c} \). Start with \( c \) as the empty set and \( \hat{c} \) as the full set of changes that finds the bug. Then, roughly divide the changes in \( \hat{c} \) in two. If testing with the first part yields the bug, continue recursively with that part. If not, try the second part. If that part yields the bug, continue recursively. Otherwise a subset of the solution must be in the first part and another subset in the second part. Continue recursively searching the first part, while implementing all the changes from the second. At the same time, search the second part while implementing all the changes to the first. The minimal solution is the union of the two searches.

The DD search algorithm assumes the problem to be monotonic; such that if a set of instrumentations reveals the bug, then any superset of this set also reveals the bug. In Ben-Asher et al. [1] and Copty and Ur [4] it was shown that concurrent programs are not necessarily monotonic, and [25] later showed that real concurrent programs are not monotonic. The reason that concurrent programs are non-monotonic is that there are three kinds of instrumentation points: relevant points, which are points that, if instrumented, increase the probability of eliciting a bug; irrelevant points, which are instrumentation points that have no effect on the probability of eliciting the bug; and blocking instrumentations, which cause the bug not to appear if they are instrumented.

Studies of bugs in multi-threaded programs [9, 16] reveal that most bug patterns can be exposed using very few instrumentation points, and sometimes only one. However, the appearance of a bug is non-deterministic in the sense that "correct" instrumentation only increases the likelihood of the bug appearing, but does not guarantee it.

Thus, the problem that needs to be solved to apply instrumentation to debugging of concurrent programs is to find a minimal subset of relevant instrumentation points, discarding both irrelevant and blocking instrumentations. This process is akin to feature selection, albeit in a significantly different setting from that assumed by most feature selection algorithms. In our case, the features, i.e., the instrumentation points, are binary variables, because noise can either be injected into an instrumentation point or not. Furthermore, it is possible to actively probe the tested program with a specific instrumentation pattern and obtain, most probably, the appearance of a bug. Finally, the existence of blocking points means that even if a correct set of relevant points is identified, they might not elicit a bug if a blocking point is instrumented with them.

Feature selection is a basic problem that has received much attention over the years. It is usually defined as the process of finding a minimal subset of indicators that best represents the data [27]. 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 one hundred features requires testing over \( 10^{30} \) configurations. However, a program containing one hundred instrumentation points is just a little bigger than a toy program. In practice, simple heuristics such as sequential forward search and sequential backward search (also known as sequential growing and sequential pruning, respectively), usually work well. In the first case, the algorithm starts with an empty set and sequentially adds the feature that, together with the current set of selected features, best improves prediction. In the latter case, the algorithm begins with all the features and sequentially removes the least indicative feature upon each iteration. A review of recent, more sophisticated methods, is given in Guyon et al. [10]. Most of these feature selection methods are of limited use in the present setting because the probability of eliciting a bug given a random subset of instrumentation points is very small. Therefore, a large number of samples are required before these algorithms can run successfully.

In the past, there were several attempts at using learning to debug programs. The authors of Zheng et al. [29] 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 that are randomly sampled at runtime. This implies that they require a diverse sample of runs 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 or 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 to incur a high computational cost, and requires setting the assertions manually in the code.

GeneticFinder [7] 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 runtime, i.e., all program locations are instrumented, and at each point it is decided during runtime whether to apply noise. From our experience, instrumentation alone can change the scheduling of the program. Thus, due to the phenomenon of blocking of partial instrumentation is much more accurate.

Tarantula [14] uses a probabilistic 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 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 users perform sampling in an independent manner.

Finally, a recent article we [25] proposed a technique based on Design of Experiments (DoE) methods. These methods find the best combination of sampling points given a budget (in the form of the possible number of program executions). Once the program is sampled, each instrumentation point is scored for its likelihood of inducing the bug and the highest scored points are reported to the user. This method is lacking in two aspects: first, the sampling points are generated before program execution, so there is no feedback between the results of one program execution and the next. Second, it uses D-Optimal, so-called because it is optimal where there is a linear relationship between the input and output of a system. However, in the case of program debugging, it is not obvious that such a relationship exists. This article aims to solve these two problems.

2. The active sampling algorithm

In the current work, we attempt to find the relevant points in a program by actively searching the space induced by the instrumentation points. However, because this space is very large, we propose to project it onto a lower dimension using a random projection, which is then sampled exhaustively. Irrelevant or blocking points are identified at the lower dimension and discarded, and the process continues until a minimal set is found.

Following is a description of the proposed algorithm, which we call the Iterative Group Sampling (IGS) algorithm. Line numbers refer to Algorithm 1.

The algorithm starts with a set \( \langle S \rangle \) of all \( D \) instrumentation points, which are all initially considered as candidate relevant points, and a low dimension parameter \( L \). We discuss the selection of \( L \) below. The algorithm generates a full factorial sampling matrix at the lower dimension \( L \). A full factorial matrix is a sampling matrix which tests all possible combinations of \( L \) (binary) inputs. This matrix, denoted by \( X_L \), has \( 2^L - 2 \) rows and \( L \) columns (the cases where no instrumentation point is activated and when all instrumentation points are activated are superfluous). At each iteration, a random matrix \( H \) of size \( L \times |S| \), whose entries are \( \{0, 1\} \), is generated. Entries in the matrix are generated such that each column of the matrix contains exactly one non-zero entry. This induces a partition of \( \langle S \rangle \) into \( L \) disjoint groups.

The actual sampling matrix according to which the program is sampled is generated by projecting \( X_L \) onto the original instrumentation space using the random matrix \( H \) (Line 5). Each of the \( 2^L - 2 \) instrumentation configurations are then tested, and if the bug is elicited it is recorded.

After running the tests, each instrumentation point is ranked according to its ability to elicit bugs. We use the likelihood ratio test [18] (which was also used in Tzoref et al. [25]) scoring function to score the points. Let \( P(\text{Success}|X_i) \) denote the sample probability that tests which instrumented the \( i \)-th instrumentation point actually results in a bug, and \( P(\|\text{Success}|X_i) \) denote the same for the case where the bug is not found. We assign each point this score:

\[
\text{Score}(i) = P(\text{Success}|X_i)/P(\|\text{Success}|X_i)
\]  

(1)

The points with the highest score are assumed to be most indicative of the location of the bug. Such a scoring function 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. Therefore, instead of keeping points with a high score, we conservatively discard from \( \langle S \rangle \) those points that obtained the lowest score. Note that these points are members of one or more groups. The process is then repeated with a smaller set \( \langle S \rangle \) until the size of the candidate group is small enough to be tested exhaustively using a full factorial design. We note that we tested other scoring functions such as the conditional entropy of the output given the sampling matrix, but our experience is that these functions are less sensitive than the likelihood ratio.

A program may contain more than one bug. Frequently, it is relatively easy to classify runs which ended prematurely to a specific bug, according to the error messages they generated. Thus, even if there is more than one bug in the program, it is possible to focus on a specific bug by observing the error produced by each run, and identifying its associated bug. In fact, during early stages of this work, we discovered an additional bug in one of the tested programs, which was previously unknown to the programmers. Both this and the other, known bug, were correctly pinpointed by the proposed algorithm.

The main parameter to set in Algorithm 1 is the low dimension at which points are sampled, \( L \). This parameter determines the number of iterations required for identifying the set of relevant features and discarding all irrelevant and blocking points. Intuitively, a low \( L \) is useful when there
can be computed as follows:
where relevant and blocking points are in different groups, all groups will result in a good configuration, that is, one dimensional search space.

The probability that a random partition of the points into $L$ groups results in blocking and relevant points is low when there are many blocking points, it is likely that there are few blocking points. However, if $L$ is chosen at a low value such that $|H| = L \times |S|$ and $\sum_{i=1}^{L} H_{i,j} = 1 \forall j = 1, 2, \ldots, |S|$. 

Compute the high dimensional sampling matrix $X_H = X_L \cdot H$.

for $i = 1$ to $2^L - 2$

Sample the program using $X_H$ such that if $H_{ij} = 1$ then point $S_j$ is instrumented.

If a bug is discovered, assign $T(i) = 1$, otherwise $T(i) = 0$.

end for

If the bug was discovered at least once during this iteration, identify groups that caused the smallest number of bug appearances using $T$ and discard all points within these groups from $S$.

end while

Return a set of relevant points $S$

are few blocking points. However, if $L$ is chosen at a low value when there are many blocking points, it is likely that the partition of points results in blocking and relevant points in the same group, which prevents their identification until a valid reparation of the points is found. Alternatively, a high $L$ requires running many tests to fully sample the low dimensional search space.

The probability that a random partition of the points into $L$ groups will result in a good configuration, that is, one where relevant and blocking points are in different groups, can be computed as follows:

$$Pr_{Good}(K_r, K_b, L) = \prod_{i=1}^{\min(K_r, L)} P_r(K_r, L, i) \cdot P_b(K_b, L, i)$$

where $P_r$ is the probability that all relevant points are distributed between exactly $i$ groups and $P_b$ is the probability that all blocking points are distributed into the remaining groups. The number of relevant points is denoted by $K_r$ and that of blocking points by $K_b$. Using this notation,

$$P_b(K_b, L, i) = \left( \frac{L - i}{L} \right)^{K_b}$$

and by applying the Inclusion-Exclusion principle,

$$P_r(K_r, L, i) = \frac{(L)^{i-1}}{K_r} \sum_{j=0}^{i-1} (-1)^j (i-j)^{K_r}$$

the total number of tests that are run by the algorithm until a good partition is found is given by $T_{Total} = (2^L - 2) \cdot N_{Iter}$, where $N_{Iter}$ is the number of iterations. To ensure that within this number of iterations a good partition is found with probability $\theta$,

$$(1 - Pr_{Good})^{N_{Iter}} \leq \theta$$

Alternatively, the total number of tests until a good partition is found is given by this equation:

$$T_{Total} (K_r, K_b, L, \theta) = \frac{\log(\theta)}{\log (1 - Pr_{Good})} \cdot (2^L - 2)$$

Thus, if $K_r$ and $K_b$ are known, it is possible to find an $L$ which minimizes $T_{Total}$. However, in practice, it may be difficult to obtain a value for these parameters. Therefore, we investigated how the number of groups should be selected, based on worse-case assumptions.

The largest probability for a bad partition is when $K_r = 1$. In that case, Equation 6 is reduced to this:

$$T_{Total, worst case} \geq \frac{\log(\theta)}{\log (1 - (\frac{L-1}{L})^{K_r})} \cdot (2^L - 2)$$

It is possible to approximate the total number of program executions using the following derivation:

$$\min_L T_{Total, worst case} \approx \min_L \frac{2^L - 2}{\log (1 - e^{-K_b/L})}$$

$$\approx \min_L \frac{2^L}{e^{-K_b/L}} \approx \min_L e^{\log(2)L + \frac{K_b}{L}}$$

Therefore, in the worst case, the minimal number of $T_{Total}$ is obtained when $L \approx \sqrt{K_b}/\log(2)$.

It is also possible to compute a worse-case estimate $K_b$ in the following manner: start the algorithm with a reasonably small $L$ and run it for a single iteration. Based on the empirical probability of finding a bug in this iteration, use Equation 2 to estimate $K_b$, and later use this estimate to compute the optimal number of groups, $L$, in Equation 7.

Theoretically, once a single good partition is found, all groups containing relevant points are identified, and a simple implementation of the delta debugging technique suffices to identify these points. In practice, however, it is expected that not all blocking points will be identified in a single iteration and the number of tests computed in Equation 6 is only an approximation of the total number of required tests.

The IGS algorithm has three possible sources of error. The first is when a bad random partition is chosen. Such a partition is one where both relevant and blocking points are in the same group. This prevents the bug from appearing during all $2^L - 2$ program executions. In such a case, the
simple solution is to repartition the points using a different random partition, and resample the program. The cost of such an error is $2^L - 2$ wasted program executions. A related problem is when a group which contains good points that were masked by bad points is removed. We have observed such behavior in practice. However, this is a rare occurrence, and is random in nature, rerunning the algorithm will usually find the correct solution. Therefore, occasional mistakes that IGS makes are more than offset by the gain in performance, as we describe below.

The second source of error is when there are redundant instrumentation points; that is, several points may independently induce the bug with similar probability. In this case, IGS may identify only some of these points. This, however, is considered to be a better choice than allowing the redundant points as well as irrelevant points to be displayed to the programmer.

In software testing, a tool that reports many false alarms will not be used. It is obvious that a tool that generates a flood of false alarms may be of little use. However, it is a common mistake for tool makers, especially static analyzers, to err on the side of caution. The tool maker may reason, for example, that it is acceptable if one of every two alarms is false, claiming that if it takes half an hour to invalidate or validate an alarm, then on average, a bug is found every hour, which is very efficient compared to testing. The problem is that while the economic reasoning is sound, in practice developers will not use such a tool as they stop trusting its output.

Therefore, the design decision taken by most successful tools will be to have fewer warnings with less likelihood of false alarms. In our case, when IGS automatically find places in the code that we think may be close to the source of the bug, it should be very careful not to choose too many irrelevant places. Thus, the fact that IGS may under-report relevant points is not a serious drawback of the algorithm. In hardware verification, where typically the cost of bugs is much higher, the design decision taken by the tools is sometimes different. They tend to err more on the side of caution as it is very important not to miss bugs. In this case, the reports may contain a higher proportion of false alarms.

Finally, the third source of error for the algorithm can occur when the bug appears by chance, regardless of the specific instrumentation. This happens (as detailed below) with a relatively low probability. In such cases, the wrong group may be discarded by the algorithm and cause it to diverge. Backtracking may be required; that is, if the algorithm fails to elicit the bug after too many random partitions, the last discarded points are added back to the set of feasible instrumentation points.

We note that IGS is easily amenable to parallelization because each of the $2^L - 2$ tests run at a given iteration can be executed in parallel. This provides additional speedup to the debugging process.

3. Benchmarking the proposed algorithm

3.1. Tested programs

We compared the performance of the IGS algorithm to that of random sampling and the RELIEF algorithm, on three concurrent Java programs. The programs and the code required for sampling them can be seen in the benchmark described by Eytani et al. [8].

The first of these programs is a web crawler embedded in an IBM product, which collects web pages by following the links from one page to the other. Delta debugging does not work for this example. The skeleton of the algorithm (that is, only the code that involves concurrent programming, without the operational code) has 1200 lines of code and 314 instrumentation points. The program contains a race condition that is very rarely manifested, since it only occurs in a very small percentage of all possible interleavings. Without instrumentation the bug did not occur in 2000 program executions. When instrumenting all possible locations, it appears on average only in one out of 750 program executions. We did not observe the bug in any of 2000 executions when randomly instrumenting two of the points, which is the minimal number of points required to elicit the specific bug (according to an analysis of the bug itself).

The second program we tested was a server loop program that continuously performs database transactions according to client requests from another IBM product. This program contains a deadlock. This program has a skeleton comprising 152 lines of code and 72 instrumentation points. The deadlock is easily induced. Randomly sampling two points induces the bug in 25% of the samples. Thus, many small subsets of instrumentations reveal the bug. Despite this fact, DD was unable to find minimal instrumentation [25], which is the goal of the debugging algorithm.

The third program we used is an example of a buffer overflow and underflow in a standard producer-consumer program. The goal is to overflow the central buffer that connects the consumers to the producers. The test program consists of four threads (two producers and two consumers) running concurrently. This program has a skeleton of 131 lines of code and 160 instrumentation points (some of the instrumentation points are in wrapper code and are not essentially a part of the actual program). Inducing the bug in this program using a random instrumentation is very difficult: only 10 of 5000 (0.2%) random instrumentations with equal probabilities caused the bug to appear. This is because in order to generate the bug it is necessary to instrument the bug in both the correct locations and in these location to cause delays only in the correct timings. In our scenario we do not have control over the latter.
3.2. Testing algorithms

RELIEF [15] is a highly successful feature weighting algorithm. It works by randomly selecting a point from the data set, finding the closest example from the data set with the same label and the closest sample with a different label, and modifying the weights so that the point from the same class becomes closer and the point from the other class more distant. In our case, the label would be whether or not the bug was observed for a given instrumentation pattern.

The algorithms are measured in terms of speed and accuracy. Speed is the number of times that a program must be run to collect enough information for identifying the relevant instrumentation points. Obviuously, running the tested programs for a very large number of times is impractical. Accuracy (or specificity) is the fraction of features identified that are in the minimal set that can be used by the programmer to understand the source of the bug.

IGS was run using between two and eight groups for each of the three test programs. For each number of groups we ran IGS twice. To maintain a valid comparison between the IGS algorithm and RELIEF, we trained the latter algorithm using the same number of runs used by IGS and selected the same number of instrumentation points as that chosen by IGS. The training points were generated so that each point was instrumented with a probability of 0.5.

3.3. Results

Figure 1 shows the number of times that each program was run as a function of the number of groups when using the IGS algorithm. As this figure demonstrates, the number of groups that minimizes the number of tests is relatively small.

A second-order polynomial was fit to the experimental data. The reason for using such a polynomial is explained by our assumption that values of $K_b$ and $K_r$ in Equation 2 are small. Under this assumption, an analysis similar to that in Equation 8 results in a second-order polynomial relationship between the number of groups and the total number of tests. For example, if $K_r = 2$, Equation 8 scales as $e^{L0.5}/L$, for which a good approximation is a second order polynomial. There is strong correlation ($R^2 > 0.75$) of a quadratic fit between the number of groups and the number of tests for the first two programs. This fit is worse for the third program we tested, which we attribute to the fact that correct instrumentation of this program requires correct instrumentation of both location and time, but our instrumentation has no control over the latter.

A human expert labeled the points in each of the programs that would assist a programmer in understanding the source of the bug. Table 1 shows the average precision obtained by the IGS algorithm compared to that of RELIEF, that is, the fraction of relevant points from all the points labeled by the algorithm as relevant. The average was computed using the results of runs with all the different group numbers. This table shows that the IGS algorithm has a relatively high precision, which, as noted above, is an important design consideration for real debugging systems. Table 1 also shows the average detection rate, which is the fraction of runs that resulted in any relevant points in the points labeled by the algorithm as relevant (i.e., a recall of at least one point). This is a different measure from the average precision in that it gives an indication of the usefulness of the search algorithm to a user. A high detection rate means that at least some of the points labeled as relevant are indicative of the bug. IGS is, again, superior in its performance compared to RELIEF, scoring a perfect detection rate for the first two programs.

We note that these 3 programs are the same as those used in Tzoref et al. [25]. The IGS algorithm identified points similar to those found by the batch method in that paper. However, in contrast with that method, where 5000 points were required for identifying the bugs (1-3 orders of magnitude more than the number of instrumentation points), IGS always required fewer program executions, sometimes up to two orders of magnitude less (See Figure 1). This reduced execution time is further exemplified in the example below.

<table>
<thead>
<tr>
<th>Program number</th>
<th>Average precision</th>
<th>Average detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGS</td>
<td>RELIEF</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.54</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>0.46</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>0.18</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table 1. Average precision and average detection rate obtained by IGS and by RELIEF.

4. Bug location enhancement

As noted in Tzoref et al. [25] it is frequently not enough to examine the score of single points in order to pinpoint the location of a bug. This is because some bugs are induced by many subsets of instrumentations, and the absolute score of each point is not indicative enough of the bug’s location. Therefore, it was suggested to look not at the absolute values of the points scores, but at the derivative of the scores along the program control-flow graph.

However, when using the IGS algorithm, not all points are run for a comparable number of runs, and thus computing their score according to the runs in which they took part may skew their scores. Therefore, we propose the following modification to the derivative score method described in Tzoref et al. [25]: Once instrumentation points which
elicit the bugs are identified, new subsets of the instrumentation points are run by traversing the program execution paths forward and backward from the instrumentation points identified by IGS. Each run contains all but one of the points identified by IGS and additionally one point which is before or after the remaining point identified by IGS, along the program execution path. This is repeated until a drop in the number of program bugs is identified.

Such a process results in a local derivative score around the instrumentation points identified by IGS. Below we show results of this method.

5. Speeding up the IGS algorithm

The basic design of the IGS algorithm as described in Algorithm 1 calls for the removal of all groups which caused the bug to appear in the fewest program executions during an iteration (See line 10). However, since a bug may appear randomly using any (or none) of the instrumentation points, it is beneficial to relax this step by allowing the removal of additional groups. This can be performed when some of the groups elicited the bug in a frequency that is significantly higher compared to other groups. For example, suppose that IGS is run with three groups, and at one of the iterations the first group did not elicit the bug, the second caused it to appear in 2% of the runs and the third in 90% of runs. Intuitively, it seems useful to discard the first two groups and keep only the last.

We propose the following procedure for deciding if some groups elicited the bug significantly more than others. After running the program, instrumented as detailed in lines 6-9 of Algorithm 1, the groups are sorted in increasing order of the number of times that the bug appeared. For each pair of consecutive groups, a contingency table is constructed for these groups, as shown in Table 2.

From this table, we can compute the probability that the difference in bug elicitation is significant, using the Normal approximation to the Binomial distribution [26], that is, assuming $a \geq c$ and labeling the total number of times that the program was executed by $n = a + b$, a test score is computed:

$$S = \frac{a - c}{\sqrt{(a + c) \times (2 \times n - a - c) / (2 \times n)}}$$

If the difference is statistically significant as determined by a test score $S$ greater than a predetermined threshold, this difference is marked as significant. We used a conservative threshold of $p < 10^{-6}$, so as to avoid possible errors due to repeated hypothesis testing. Only groups with the largest number of times that the bug was elicited and which are above a statistical significance threshold are kept. Other groups are discarded. If no statistically significant difference is found, the basic IGS procedure is taken, as shown in Algorithm 1.

6. Scaling up to larger programs

In this section we describe our experiments on a much larger program compared to the ones described in Section 3. Because of its size the batch algorithm proposed by Tzoref et al. [25] could not have analyzed this program within reasonable time. We did not make any changes or parameter adjustments when applying IGS to this example. This demonstrates the strength of IGS in that it requires no knowledge of the specific properties of the bug (other than

<table>
<thead>
<tr>
<th>Number of times</th>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found bug</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>Did not find bug</td>
<td>b</td>
<td>d</td>
</tr>
</tbody>
</table>

Table 2. Contingency table for comparing two IGS groups
a way to identify its occurrence) or the program under test in order to converge to the correct solution.

We tested the enhancement of the IGS algorithm on the Java 1.4 collection library that was used as a case study in Sen and Agha [21]. This is a thread-safe Collection framework implemented as part of the java.util package of the standard Java library provided by Sun Microsystems. It consists of 141 classes and has 8067 instrumentation points. In Sen and Agha[21], they describe several concurrent bugs found in this library. One of these bugs is a race condition that may occur on the field size of the LinkedList class when the methods l1.clear() and l2.containsAll(l1) are concurrently executed. The value of size of l1 is set to zero while l2 is iterating on the elements of l1. If the write access to size occurs after l2 consumes an element and increments the elements counter, the race leads to an infinite loop. However, if size is set to zero a short while before, immediately after l2 consumes the element but before it increments the counter, this results in an uncaught exception due to a failure of a sanity check on the values of size and the counter, which is performed by l2.

We used two threads for each test. Two LinkedList objects were instantiated, and concurrently called the library methods clear() and containAll(). Two points were labeled for each bug, indicating the accesses that cause the race.

When randomly instrumenting two points out of the 8067 instrumentation points, which is the minimal number of points required to manifest the race, we have never seen either of the errors occur.

IGS was run twice on this example, each time focusing on a different manifestation of the race. The number of groups used was five. Thirty-four iterations were required for convergence when focusing on the infinite loop, and 27 iterations when focusing on the uncaught exception.

For both cases, IGS identified two points as eliciting the bug. The point just before setting size to zero appeared in both pairs. For each pair of points (p1, p2), we measured the local derivative score as described in Section 4. In each iteration, we instrumented two points and ran the program multiple times, where at each run one point is an original point p1, and the other is a point in the surroundings of p2 on the control flow graph. We continued until observing a drop in the number of times the bug was manifested. We then repeated the process while switching the roles of p1 and p2. The results for both bugs are depicted in Figure 2. A partial control flow graph of the program is shown where nodes denote instrumentation points and edges possible execution flows. The two best points (nodes) found by IGS are highlighted as are the maximal and minimal derivative scores in the surrounding of the best two points found by IGS. The specific details of the instrumentation points in the figure is of lesser importance. We use it to demonstrate the localization of the bugs achieved using IGS.

For both cases, the derivative score reaches its peak at the points of the race that lead to the bug, i.e., for the infinite loop the points are right before size is set to zero and right after the sanity check is performed, and for the uncaught exception the points are right before size is set to zero and right before the sanity check is performed. Thus, for both cases, we were able to pinpoint the race and the exact scenario that leads to the error.

Finally, when using the speedup procedure described in Section 5, we found that the run-time of the IGS algorithm was reduced by approximately 50% when locating the element exception bug, but without significant reduction for the infinite loop bug. In both cases, however, similar results were obtained with regards to the instrumentation points which caused the bugs.

7. Discussion

In this paper we present an efficient automatic method for debugging concurrent programs. Using instrumentation of points within a program and noise injection it is possible to elicit bugs and pinpoint locations in the program code that are strongly suggestive of the source of the bugs. We demonstrated that finding an indicative minimal set can be posed as a problem of active feature selection and suggested the use of the IGS algorithm for solving this problem. Our analysis and results suggest that the IGS algorithm is much more accurate than the RELIEF algorithm, obtaining good results with much fewer runs compared to previously-used batch algorithms.

We also demonstrated two improvements of IGS which assist in bug localization and improve run-time. Finally, we demonstrated that IGS can scale to large concurrent programs which could not be debugged in reasonable time using the previously suggested batch algorithm.

One major drawback of IGS is that it does not take into account the sampling history; that is, the information collected in previous iterations about each instrumentation point. In future work we intend to find methods for a principled way in which to incorporate such information, which, it is hoped, will speed up the convergence of IGS even further.

Acknowledgment

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...
Figure 2. Difference score graph for the Java 1.4 collection library. The top graph shows the scores for the element exception bug and the bottom graph shows the scores for the infinite loop bug. The original points found by IGS as well as the largest and smallest derivative scores in the surrounding of the best points found by IGS are highlighted.
of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

References


