Producing Scheduling that Causes Concurrent Programs to Fail

Yosi Ben-Asher[1], Yaniv Eytani[2], Eitan Farchi[3], Shmuel Ur[3]

[1] Computer Science Department, University of Haifa, Israel

yosi@cs.haifa.ac.il

[2] Computer Science Department, University of Illinois at Urbana Champaign, USA

yeytani2@uiuc.edu

[3] IBM Haifa research labs, Israel

{farchi, ur}@ibm.il

Abstract. A noise maker is a tool that seeds a concurrent program with conditional synchronization primitives (such as yield()) for the purpose of increasing the likelihood that a bug manifest itself. This work explores the theory and practice of choosing where in the program to induce such thread switches at runtime. We introduce a novel fault model that classifies locations as "good", "neutral", or "bad," based on the effect of a thread switch at the location. Using the model we explore the terms in which efficient search for real-life concurrent bugs can be carried out. We accordingly justify the use of probabilistic algorithms for this search and gain a deeper insight of the work done so far on noise-making. We validate our approach by experimenting with a set of programs taken from publicly available multi-threaded benchmark. Our empirical evidence demonstrates that real-life behavior is similar to what our model predicts.

1 Introduction

The increasing popularity of concurrent programming, for the Internet and on the server side, has brought the issue of concurrent defect analysis to the forefront. Concurrent defects such as unintentional race conditions or deadlocks are difficult and expensive to uncover, and such faults often escape to the field. Starting in 2006, all processors will have dual cores and personal computers will have hyper-threaded processors. Programs that used to work well on single-threaded and single CPU core processors are now exhibiting problems. This means the testing of multi-threaded programs and the development of technology and tools to identify concurrent defects

are more crucial than ever. As a result, major companies such as Intel and Microsoft have addressed this issue by providing appropriate testing tools.

There are a number of features that distinguish concurrent defect analysis from sequential testing. These differences are especially challenging because 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 concurrent fault can be very low. Second, under the simple conditions of unit testing, the scheduler is deterministic; therefore, executing the same tests repeatedly produces the same set of interleavings. As a result, concurrent bugs often escape unnoticed and are found only in stress tests or by the customers. Furthermore, the tests that reveal faults are usually long and run under different environmental conditions. Consequently, they are not necessarily repeatable and when a fault is detected, extensive efforts must be invested in recreating the conditions under which it occurred. When these are finally recreated, the debugging itself may mask the bug (the observer effect) for example, adding debugging printing may change timing conditions and mask a concurrent bug.

A large amount of research is being carried out, both in academia and industry, to improve the quality of multi-threaded software. Progress has been made on many fronts and it now seems evident that a high quality solution must contain components from many domains [9]. Work on race detection and atomicity is a primary effort that has been going on for a long time [19][21][1] [12]. A different approach is taken by a set of tools called noise makers, which increase the likelihood of bugs discovery by inducing different timing scenarios through the addition of scheduling perturbations [23][2] [7]. The noise makers themselves do not report results, rather, they strive to make the tests fail. Other tools, used for replay and partial replay [4][7][22] are necessary for debugging and contain technology that is useful for testing. Static analysis tools of various types [16] as well as formal analysis tools [24][17] [5], are also being developed to detect faults in this domain . Other tools present specific and interesting views of the interleaving space to help analyze both coverage and performance [3] [15].

This paper explores the theory and practice of deciding where in the program to induce thread switches. Previous research in the area used a white box approach. In [23], static analysis was suggested for use in detecting the locations in which to thread switches that will help reveal bug patterns. In [7], noise-making decisions were made based on coverage information. Experiments carried out in [2] showed that focusing one-at-a-time on the locations related to variables will improve the probability of finding bugs. The work in [23] [7] showed that performing context switches at random locations using a uniform probability (referred to as white noise), improves the likelihood that a bug will manifest.

Conversely, this work uses a black box approach and develops a theory for good, neutral, or bad program locations for thread switches. This allows us to reason about concurrent bugs and provides a deeper insight of the work done so far. In addition, we demonstrate that there are testing scenarios where using a uniform probability for choosing the location to be seeded, results in a very low probability of revealing bugs. Such scenarios are typical of bugs that only manifest in specific timing win-

dows, where using a uniform probability will either not chose all good events or will be likely to choose some bad program locations. Thus, we motivate the use of a two-level scheme (presented in a limited form in [2]) that involves first selecting a subset of the program locations and then performing context switches with a high probability. We present experiments that validate our theory and show how it can be used to improve our testing tool.

1.1 Motivation and complexity of the problem

We begin with several examples that demonstrate the complexity of the problem and explain the motivation of using a heuristic approach. In the first example, we demonstrate that adding or removing context switches can significantly change the probability of bug manifestation. Consider the program in Figure 1 with three threads (*T1*, *T2*, and *T3*) and nine program locations. For the sake of simplicity, assume the scheduler always starts with *T1* and upon hitting a context-switch or end-of-thread the scheduler continues on any of the remaining threads with equal probability. If there are no available threads, the scheduler continues executing the current thread. The decision trees for the example depict the choice made by the scheduler regarding which thread to advance to upon hitting a context switch or an end of thread.

When a thread switch is inserted after location *T1.2* there are four possible interleavings, of which two reveal the bug (i.e., executing y=1/x while x=0) as shown in Figure 1a. As all interleavings in this case have the same probability, the probability that the bug manifests is ½. If we insert an additional context switch after T2.1, we find there are six interleavings, of which only one reveals the bug (Figure *1b*). However, the probability of selecting a given interleaving is not uniform, since interleavings that contain more decision points have a lower probability of manifesting. Therefore, the probability of bug manifestation will actually be smaller here (i.e., *1/8*). Figure 1b depicts the change in the interleaving space (scheduling decision tree) as a result of inserting an additional context switch at *T2.1*. Clearly, adding or removing the second context switch at *T2.1* changes the probability of the bug manifestation.

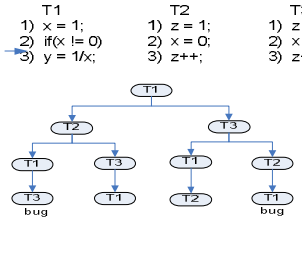

Figure 1a

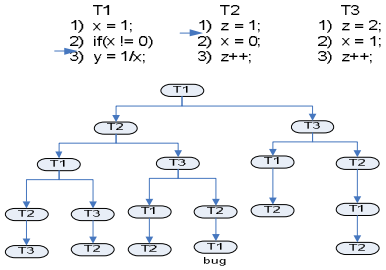

Figure 1b

We further argue that choosing a program location with uniform probability may in practice be insufficient for detecting a concurrent bug. Consider the scenario illustrated by Figure *2a*. For the sake of clarity, we introduce only two threads (*T1* and *T2*) although the end results are similar to having three threads. This example is similar to the one presented in Figure 1, except for having many locations in thread T2 that are considered irrelevant to the bug. For the bug to manifest itself, a thread switch must occur after location *T1.2*. In addition, a context switch must not occur at any of the points in thread *T2* that are before *T2.N*; otherwise, the assignment of the value of zero to variable 'x' to will not precede the division of 'x' and will hence mask the bug. Consequently, if we select the seeding points using a uniform probability *p*, the probability that the bug will manifest is as follows: the probability *p* of choosing to perform a thread switch in location T1.2, multiplied by the probability $(1\text{-}p)^N$ that it will not choose to perform a thread switch in locations *T2.1* … *T2.N*. Hence, the probability of manifesting the bug is small for any uniform *p*.

The decision whether a program location should be considered 'good' or 'bad' may well depend on previous or future decisions regarding the seeding of other program locations. Consider the example in Figure 2b. In order for the bug to manifest itself in this example, a single thread switch (and no more than one) must occur before the assignment of variable 'x' to be zero occurs at *T1***.** Clearly, the benefit of seeding *T1.2* strongly depends on whether or not a context switch has been inserted after *T1.1* Thus, we cannot absolutely classify location *T2.2* as either good or bad without knowing the location of the other context switches (referred to as the 'context').

Furthermore, there exist worse scenarios where attributing a deterministic decision, such as always or always not performing a thread switch to a specific location, will mask the bug entirely. For example, consider the program in Figure 2c. For the bug to manifest (i.e., for the program take a divide by zero exception) a thread switch must occur after the instruction *if (i!=0)*. However, the thread switch must occur only in the iteration when *i=5*, otherwise the bug will not manifest. In short, a deterministic classification of the program's locations (such as *T1.3*) as either good or bad will not identify the bug in this program.

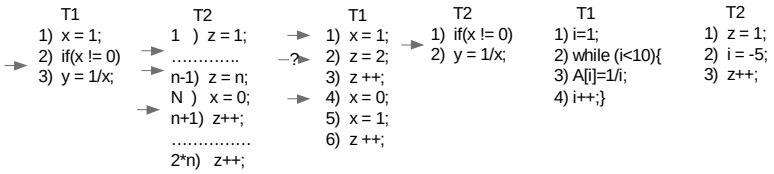

Figure 2a Figure 2b Figure 2c

Thus, we consider it is too difficult to decide on-the-fly whether or not to make a context switch when a given program location is reached. In this work we address this problem by providing a model and a static classification of the program's locations. The main contributions of this work are:

- Provide a formal definition and a new novel taxonomy that distinguish between three types of events: “good”, “bad” and “neutral”, with respect to whether forcing a context switch will help to reveal a bug.
- Observe that choosing locations to insert random context switches at a uniform probability is likely to include bad program locations. We introduce a two-phase scheme that involves first selecting a subset of the program locations and then performing context switches with a high probability. This is demonstrated as essential for masking out bad program locations.
- Develop a mathematical model for noise-making; including experimental validation that scheduling noise behaves in a convex-like manner. This implies there are some locations with significantly higher probabilities of being good and bad than the rest of the locations.
- Explore the terms in which efficient search for real-life concurrent bugs could be carried out and justify the use of probabilistic algorithms for this search. Studying known concurrent bug taxonomies, we spouse these are usually composed of a small number of good and bad locations.

The rest of the paper is organized as follows. Section 2 provides a practical definition and a metric for classifying locations as good, bad, and neutral, along with the understanding of why this approximation is useful. Section 3 presents the mathematical model associated with this classification, its expected behavior. Section 4 explores the terms under which we can perform efficient search in the model, motivates the creation of new black box heuristics and the possible extension of our model to be used with other tools. In Section 5, we discuss the experiments conducted to support our intuition and the mathematical model. The final section lists our conclusions and provides details about future work.

2 A model for adding scheduling perturbations

Our methodology facilitates the use of a noise maker as a testing tool. In this way, we control the probability that a thread switch will occur when each event is executed [23]. Our goal is to choose the optimal subset of the program's locations in which a context should be performed. By studying the taxonomy of concurrent bugs [10] [18] [13], we determine that a concurrent bug manifests when a few concurrent events (referred to as events [4]) occur in a specific order. We can thus identify a concurrent bug with that sequence of concurrent events. A context switch is highly desirable after or before some of these events for the bug to manifest. Ideally, no context switches should occur in other events. Since, whether a context switch is desirable or not may well depend on other decisions made.

Because it is too difficult to decide on-the-fly whether or not to make a context switch when a given program location is reached, we suggest approximating the decision by providing a static classification for program locations. For this aim we divide the program’s events into three categories:

- ***Good events*** – in which a context switch greatly increases the probability that a concurrent bug manifests. As demonstrated previously, it may be preferable to increase the probability of a context switch when such an event is reached and not to always make a context switch.
- ***Bad events*** – in which a context switch generally decreases the probability that a concurrent bug manifests or completely masks the bug.
- ***Neutral events*** – in which a context switch does not greatly change the probability that a concurrent bug manifests.

As the classification of the program's events is not known a-priori we need a method to for determining for each event *e* whether it is good, bad, or neutral. A possible mechanism could be defined as follows: run the program many times and in every execution, set every event but *e* to be good or bad with equal probability. In first half of the executions we set *e* as good (e.g., perform a context switch in *e* with high probability) and in the second half of the executions we set *e* as bad (e.g., perform no context switch in *e*). If the probability of revealing a bug in the first half of the execution equals that in the second half of the execution, *p* is neutral; otherwise, *p* is either bad or good, based on which half manifests more bugs.

In addition to being impractical, due to the possibly high number of executions required, this method has a number of theoretical problems. The definition of *e* as good or bad is related to decisions made at other events. However, the former definition does not take into account what decisions were actually made. (i.e, the execution's context) and it might be not reasonable to choose all other events to be good or bad with equal probability). Thus, a more rigorous definition is required. In the next sub-section we detail our proposed metric for approximating the static classification of the program's events, regardless of the execution's context.

2.1 A metric for static classification of the program's events

As explained earlier, the effect that a context switch at an event may have for manifesting a bug is relative to a specific execution.. In this section, we propose a metric to statically approximate this effect at each event. The intended meaning is that performing (or not performing) a context switch at such an event will increase the likelihood of the bug manifesting regardless of the specific execution's context. This is done by considering the effect a context switch may have over all possible contexts. The classification is introduced here to motivate the creation of the model. Learning the classification at run-time could prove helpful for debugging purposes; however, it is beyond the scope of this work.

We evaluate the classification of each of the program's events as follows:. At each event, we calculate the average probability that the bug will manifest itself over the introduction of context switches at all possible *k* other events, where bug taxonomies (and also [20]) shows we can choose a small *k*. An event is classified as good if the average probability of manifesting the bug is sufficiently high. Similarly, if the aver-

age probability is very small, the location will be classified as bad. If it is neither bad nor good, it is classified as neutral.

As an example, consider the program in Figure 1 and assume we intend to insert two context switches (possibly after each program location). Thus, there are six relevant events (e.g., program locations) that need to be classified *T1.1, T1.2, T2.1, T2.2, T3.1,* and *T3.2*. In order to classify a program's locations we need to check every possible interleaving over all the possibilities of selecting the other location. For example, in order to classify *T2.2* we need to cover all the interleavings when the second context switch is inserted at *T1.1, T1.2, T2.1, T3.1, T3.2* and average the probabilities for bug manifestation. Figure 3 summarizes these probabilities and the resulting classifications .

The approximated classification in Figure 3 fit nicely with the intuitive one described in the introduction. Analyzing the average probability for each location we observe that a context switch should occur after checking that 'x' is not equal to zero and must not occur before the assignment of zero as the value of ' x'. It is best that no context switch is performed, (but not absolutely necessary) at all other locations.

Program location	Prob. over all contexts	Resulting classification		Prob. over all contexts	Resulting classification
T1.1	2/40	Neutral	T2.2	5/40	Neutral
T1.2	12/40	Good	T3.1	3/40	Neutral
T2.1	0	Bad	T3.2	3/40	Neutral

Figure 3

Events can be naturally grouped using different levels of granularity. Such groups may be composed of a set of events that access a shared variable, events related to a given program location, or locations that share the same context. Thus, for each group of events (e.g., shared variable or program location) we loosely assign a probability of whether it is good, bad, or neutral, based on the probability that good, bad, or neutral events associated with it are executed.

3 Analyzing and using the model

In this section, we analyze the model introduced above. Our goal is to present a mathematical model to allow us to analyze the possible scheduling noise behaviors to latter on use this information to improve our testing tool. As explained above, a program under test is viewed as a set of subsets of events $\{e_1 \ldots e_n\}$. In what follows, e_i could be the set of all events that access a given shared variable or a set of events associated with the same program location. PBi, PG_i and PN_i are the probabilities of e_i being bad, good or neutral respectively ($PB_i+PG_i+PN_i = 1$). The intended meaning is that, for a given execution, e_i will behave either as good, bad or neutral in the above probability distribution. Thus, we choose a type for each element, and independently obtain a subset Q of P's elements by choosing every element using a probability p. Probability p models whether or not a context switch is forced when the

program element is accessed. Q is “successful” if it contains no elements of type bad and all the elements of type good. Otherwise, Q is "unsuccessful". The model equates the probability of Q being successful with the probability of manifesting the bug. We consider two general variants of the model:

- All elements have the same probability of being good, bad, or neutral. Note that the same element can be bad in one execution of the program, good in another execution, and so forth.
- Each element is either good, bad, or neutral, implying a fixed but unknown number of good/bad/neutral elements.

For both cases, we need to solve a search problem where we have to uncover all good elements and no bad elements. This search uses one query to select a subset of the elements (*Q*), seeds them, and runs the seeded program. If a bug occurs, we have chosen all the good elements and none of the bad elements. If the query fails (i.e., no bug is detected), it is not possible to gain any information regarding the chosen elements in *Q* and another subset must be selected for the next query.

We first consider an average case analysis for the first variant of the model. Assume that an average, *p*n*, number of elements are selected. For the selected *p*n* elements to be "successful" (i.e., contain all good elements and no bad elements) the following must be true: all the *p*n* elements should contain either good or neutral elements while the remaining (*n-p*n*) elements should be either bad or neutral elements. Thus, the probability $p(Q_{successful})$ is given by:

$p(Q_{successful}) = (PG + PN)^{(p*n)} * (PB + PN)^{(n-p*n)}$, we thus obtain Claim 1:

1. If *PG* > *PB*, then it always pays to select all elements with *p=1*.
2. If *PG* < *PB*, then it always pays to select none of the elements using *p=0*.
   - Note, if we choose no element, we can still "win", as there is a significant probability that all the *n* elements are neutral.
3. If PG = PB, assuming n is large, there exists an x such that x/n = PG=PB. Thus, $(PG + PN)^{p*n} * (PB + PN)^{(n-p*n)} = (1-(x/n))^n \dashrightarrow 1/(e^x)$
   - If *PG=PB < 1/n* then it goes to infinite
   - If *PG=PB=1/n* then the probability of success is 1/e ~ ½
   - If *PG=PB>1/n* then goes quickly to zero

For the second model type, where we assume that each element is either good, bad, or neutral, Claim 2 holds as follows: Assuming we have *k* good and *r* bad elements, the probability that Q is successful is given by: $p(Q_{successful}) = p^k * (1-p)^r$

Note, this function obtains its maximum at: $\frac{k}{k+r}$

In the general case where each element has a probability (PB_i, PG_i, and PN_i), the following two distinct alternatives exist: either probabilities for each element are about the same, making it possible to analyze the case using the average probabilities

PB, *PG*, and *PN*. Otherwise, there are some elements with probabilities of PG_i/PB_i that are significantly higher than the rest of the elements. This case is analyzed in a similar manner to the second variant of the model. For the first alternative, Claim 1 holds, while for the second alternative Claim 2 is applicable.

Thus, we reduce the general case to one of the two above claims. If the first alternative holds, the bug could manifest in many cases without applying scheduling noise to the program's execution, eliminating the need to insert such noise. If the second alternative holds, the model predicts that for a given concurrent program containing a concurrent bug, there exists a globally optimal probability for seeding noise in which the bugs are most likely to manifest and that the rate of finding these bugs goes down with an increase or decrease of that probability. Experiments described in a later section show that the probability applying noise on the tested programs tends to behave in a manner similar to that predicted by Claim 2.

4 Methods for efficient search

In this section, we reason about the terms under which concurrent bugs can efficiently be detected. Considering the huge interleavings space and the fact that noise makers have been shown to detect concurrent bugs, one could naturally argue this implies that an overwhelming portion of this space must manifest the bug. This, in turn, would imply that every method that exhaustively scans all possible interleavings is likely to quickly find bugs. However, this is known not to be the case. Using the model, we show that another possibility exists; if there are only a small number of elements that are classified as either good or bad, the proportion of bug manifesting interleavings is still polynomial. However, we can provide an efficient search procedure to find a subset of program events that constitutes a successful query (contains no bad events and all good events) by running the program a small number of times. As explained above, we equate such a successful query with the bug manifesting itself.

We use a search technique that for a given program with n program locations and a deterministic classification (second variant of the model) produces a small group of subsets of program locations ($S_1,\ldots,S_l$ where $l<n$), such that one of them will reveal the bug. Afterwards, we show how this technique can be extended to deal with a probabilistic classification (first and second variant of the model) to produce a small fixed number of subsets of program's locations ($S_1,\ldots,S_l$ where $l<n$) such that one of them will reveal the bug with a high probability. Performing an exhaustive search over all possible queries is not practical as it is exponential to the program size.

We formalize this problem as follows: Assume there are n events l1,…,ln where each event is either good, bad or neutral. You are allowed to select a subset (Si) of event (without knowing their classification) and execute the program. If none of the locations in Si are bad and all good locations are in Si, you are done (the bug has manifested). For a small number of good and bad events (k) , (k<<n), there are O(logk(n)) subsets such that one of them separates the good events from the bad events (e.g., for k=1 there are 2*log(n) such of subsets). This can be generalized to

any value of n and small k (see [10]). Thus, we have a deterministic procedure with a logarithmic number of tests that detects any bug composed from a small number of events good and bad events.

For example, consider the case where *n=16* and *k=1*. We select four subsets S_1, S_2, S_3, S_4 and their complements S_1', S_2', S_3', S_4', such that at least one of them (for any choice of one bad location and one good location) will contain the good location and will not contain the bad location. This scenario is depicted in Figure 4 where S_4' will pass the test as it does not contain the bad location l_3, but does contain the good location l_{11}. This follows from the fact that the sets S_1, S_2, S_3, S_3 and their complements denote all the binary numbers 0...15. Thus, if there exists a choice that does not separate between a bad event and a good event, there is one number whose binary representation appears twice.

In the experimental part of this work, we use a random method to select these subsets instead of a deterministic method. This is done since for a small number of subsets, selecting elements at random (e.g., choosing an element to belong to S_i with probability *0.5*) is as effective as the deterministic selection for the purpose of separating the good and bad elements (see [13]). Random construction is preferable due to:

- The number of good and bad locations is not known a priori. A successful query may be composed of more than one good or bad element. Thus, different number of good and bad elements will require a different deterministic construction.
- Random selection is an effective solution to the variant of the model where elements are classified with a probability. This is since the classification of an event to be good or bad element could change between executions (for example, a neutral event in one execution may become bad in another and vise versa).
- Random seeding of program locations is more easily implemented than deterministic selection.

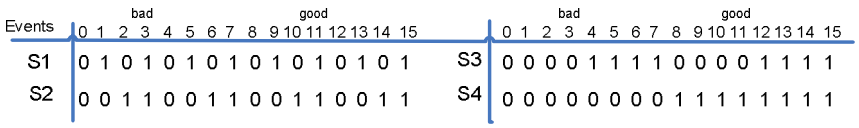

Figure 4

4.1 Using the model to improve existing testing tools

In this section we use the introduced model and search method to improve raceFinder [2], our previously develop noise maker. In addition, we discuss the possibility of using these methods in tools from other testing and verification domains. The model suggests that for a given scheduling heuristic [2] and a given program there is an optimal probability for choosing where to make noise. In addition, we consider the fact that studies of concurrent bug patterns [10] [13] [18], research on data races

[21], and atomicity [12] suggest that concurrent bugs are usually composed of a small number of good and bad events. Thus, we use the model to motivate the creation of new black box heuristics based on a two-level seeding technique. In the first level, each heuristic chooses a subset of program events. This is done by randomly choosing every variable or program location with a given probability. In the second level, scheduling noise is applied only to chosen subset of events.

The first heuristic, referred to as 'random noise', takes advantage of the observation that an optimal probability exists. If we choose the probability randomly, we will eventually hit upon the optimal probability or come close to the optimal probability. As our model is continuous, we can assume that when we hit a near-to-optimal probability, the probability of revealing the bug will be nearly optimal. In addition, if we fix a required distance, *d*, from the optimal probability, then *(1/(2*d))log(1/(2*d))* runs will be required on average case to hit a probability that is close enough to the optimal probability.

In the second heuristic, referred to as 'best noise', we work with a formula based the second variant of the model. We assume the bug occurs when a sequence of concurrent events accesses a specific shared variable in a specific order. We thus assume that there is one good event, while the number of bad events is proportional to the program size (*n)*, and this proportion is determined by a parameter *c*. Thus, at each execution we choose each event with probability: $p = \frac{1}{1+cn}$,the optimal probability given by the model.

Our search method can be extended to work with tools from different testing domains. We motivate this idea by considering a possible use with context-bounded model checkers [20][17]. Such tools search for concurrent bugs by placing a limited number of context switches (*k*) over all possible locations of the program (*n*). When considering this method in light of our introduced model two obvious limitations come to mind: 1) assuming the existence of only good events may lead to a false sense of security by looking at smaller sequences than required, 2) high computational complexity of calculating all possibilities to place the *k* context switches ($O(n^k)$). As shown, there exists a deterministic construction that requires a substantially lower number of interleavings to be scanned. This is done by placing a larger number of context switches (than the minimal number required for the bug to manifest) at locations that are pre-determined by this construction.

Using the probabilistic construction we can motive random selection of where to place context switches for context-bounded model checking. However, we do not claim that our search methods should necessarily be applied as-is with other tools. For example, when working with a model checker it may not be desirable to place many context switches. Thus, a construction that favors smaller subsets having fewer contexts switches may be desirable. This in turn will result in a larger number of tests required. Moreover, applying our search method for verification purposes will require extending the classification to be deterministic (the second variant of the model). It is not clear to us yet, whether and how this could be done.

5 Experimental results

In this section, we aim to validate that the observed behavior of the noise probability acts in a manner similar to that predicted by our model. In addition, we compared the new heuristics with ones used in previous work [2], [7],[23]. In our experiments we used programs taken from the publicly available multi-thread benchmark [9]. Figure 5 presents the names of the test programs, their size measured by lines of code, and their bug type. The programs were tested using on a cluster of Pentium IV 1800 computers using the Java SDK standard edition, version 1.4.1_01 (results are similar to using older versions) using windows operating systems (2000 & XP). In all experiment scheduling noise was applied using the yield() scheduling heuristic [2]. Using small programs in the experiments provided the advantage of making their behavior relatively easy to analyze. The larger programs used demonstrate the scalability of our approach. The Crawler algorithm is a real life race condition bug embedded in an IBM product [7]. The execution time of the different program ranges from a few seconds to a few minutes. Appling scheduling noise does not significantly prolong the execution time of the programs, except for the Shop program.

Program name	**# LOC**	**Bug type**
Account	160	Weak reality
Max	220	Weak reality
Shop	320	Sleep + weak
Piper	280	Wait not in loop
Manager	520	Weak reality

Program	**# LOC**	**Bug type**
Crawler	1200	Guarded if
Raytracer	1600	Weak reality
XtangoAnimator	2088	Deadlock
Dictionary	706	Weak reality

Figure 5

5.1 Noise probability behavior

In two following experiments, we examined how the bug manifestation rate correlates with the different probabilities for choosing a subset of events to seed with scheduling noise. For each test program, we measured the number of times a concurrent bug manifested itself throughout 1000 runs for each chosen probability. We conducted two experiments by choosing two types for grouping events: 1) the set of events that access a single shared variable (variables), 2) the set of events that are related to a memory access at a given program location (program locations). To obtain a subset we chose each element with probability *p* going over all elements (either variables or program locations). In both graphs, the X-axis represents the probability *p* of choosing either a program locations (Figure 6a) a shared variables (Figure 6b) and the Y-axis represents the number of times the bug manifested. Results are shown for a subset of the programs.

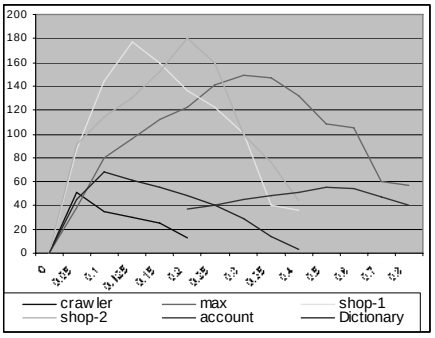

Figure 6a - program locations

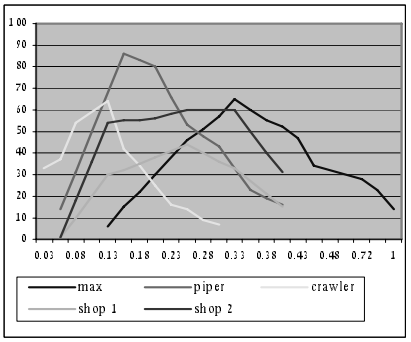

Figure 6b – shared variables

5.2 Comparing different heuristics

In this experiment, we compared the following heuristics: 1) white noise, 2) the new black box heuristics and 3) focusing on a variable related to the concurrent bug (referred to as contention noise [2]). These heuristics were tested on a subset of the test programs described in Figure 5. Scheduling noise was applied using the yield() scheduling heuristic. In order to evaluate the strength of the different heuristics, each program was run 500 times and we counted the number of times a bug has manifested. In the comparative graph (Figure 7a), the Y-axis represents the total number of bugs found when each program was run and the X-axis summarizes the performance of each heuristic. The heuristics "Random" and "Best" were implemented are as described in Section 3.2 (where "best" is executed with the *c* parameter chosen to be as high as possible). Figure 7b illustrates the effect that parameter c has on manifesting the bug, using the 'best noise' heuristic. The Y-axis represents the total number of bugs found when each program was run and the X-axis represents different values chose for the parameter *c*.

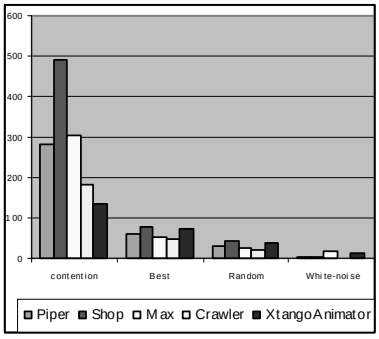

Figure 7a – different heuristics

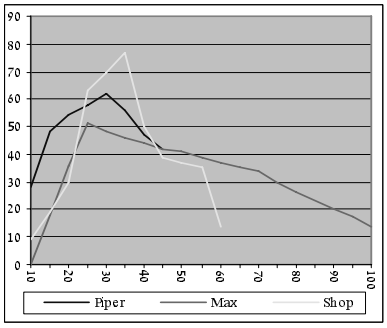

Figure 7b – parameter C

5.3 Analyzing the results

We hypothesized that since bad locations exist, bug finding performance should not monotonically increase with the more locations selected, as it would with only good and neutrals. Thus, an optimal probability should exist and the number of bugs found will go down with an increase or decrease of probability. The results obtained by the both experiments as well as other experience gathered over the last three years [2][9] validate our predictions. The results demonstrate that a global optimal noise probability exists, and choosing other probabilities causes the bug to manifest less frequently. The frequency with which the bug manifests depends on the proximity of the chosen probability to the optimal probability. This behavior is similar to the one predicted by the second variant of the model (an event is either good, bad or neutral). Thus, this implies that some elements have significantly higher probabilities of being either more good or bad than the rest of the elements.

Furthermore, we hypothesized that the number of good locations is small and independent of the program size, and that the number of bad locations may increase with program size (as illustrated by the examples in the introduction). Thus, we predicted that the optimal probability, in accordance with our model's formula, will decrease with the size of the program. The general trend in our experiments pointed to this behavior when we observed the optimal probability and the program's size. We saw that choosing events at the granularity of program locations rather than at the variable level yields better results as more bugs were found using the optimal probability.

When comparing the results from the different heuristics, we observed that the "random" heuristic greatly increases the number of bugs found in comparison to white noise. When the 'best noise' heuristic was used with the correct value for the *c* parameter, it outperformed the random heuristic and outperformed white noise by almost an order of magnitude. As expected, focusing the noise on the variable related to the concurrent bug provides better results than the rest of the heuristics, as it is based on having a priori knowledge about the program. Thus, the new black box heuristics can be applied when contention information or other information about the program is not available or incorrect. In addition, we experimented with different values of *c* when using the 'best noise' heuristic and observed that the best value of c is close to ⅓. However, we believe that in order to achieve conclusive findings regarding this parameter we a larger experiment should be conducted.

6 Conclusions and future work

This paper explores the theory and practice of deciding where in the program to induce thread switches. Contrary to previous research in the area, we use a black box approach and develop a theory for good, neutral, or bad program locations for thread switches. We also demonstrate that there are testing scenarios where using a uniform probability for choosing the location to be seeded results in a very low probability of revealing bugs. This, we motivate the use of a two-level scheme of first selecting a

subset of the program locations and then performing context switches. We present experiments that validate our theory and show how it can be used to improve our testing tool.

This work presents the following novel contributions. We explore the terms in which efficient search for real-life concurrent bugs could be carried out. The presented methods, or similar alternatives, are likely to work only for bug patterns containing few good and bad locations. Our extensive experience with analyzing bugs, both in hardware and in software, shows us that most bugs fall into this category. We reason about the behavior of concurrent bugs to for a deeper insight of the work done so far. Furthermore, understanding the interactions between program elements that lead to concurrent bugs allows motivating the development of new techniques. For example, the insights presented in this work have already motivated formulating noise-making as a search problem [8] to allow the use of AI-based search methods to enhance noise-making process.

In future work we would like to use information about the program gathered at run-time to learn the classification of program locations. We believe that this will prove most helpful for debugging purposes as initial results have already indicated. Our developed model is the first step towards defining a new model composed of long scheduling elements (as opposed to context switch which are "short" scheduling elements). We believe this will further allow reducing the complexity of the search process and could also be applicable to the formal methods domain. Finally, we intend to extend this work to model realm of sequential programs.

7. Acknowledgments

We would like to thank Alan Hartman, Ronen Shaltiel, Oded Lachish and Tao Xie for their comments relating to this work. We thank the Caesarea Edmond Benjamin the Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science, at the University of Haifa for partially supporting this research.

References

1. C. Artho, A. Biere, and K. Havelund: High-level Data Races. In: Journal on Software Testing, Verification and Reliability (STVR), 13(4).
2. Y. Ben-Asher, Y. Eytani, and E. Farchi. "Heuristics for Finding Concurrent Bugs." In Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.
3. S. Cheer-Sun Yang and L. Pollock. "All-du-path coverage for parallel programs". In International Symposium on Software Testing and Analysis, 1998.
4. J. D. Choi and H. Srinivasan. "Deterministic replay of Java multithreaded applications." In Symposium on Parallel and Distributed Tools, 1998.
5. J. C. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. "Bandera: Extracting finite-state models from Java source code." In International Conference on Software Engineering (ICSE), 2000.

6. Dinning and E. Schonberg. “Detecting Anomalies in Programs with Critical Sections.” In Conference Proceedings on ACM/ONR Workshop on Parallel and Distributed Debugging, 1991.
7. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. “Multithreaded Java Program Test Generation.” IBM Systems Journal, Vol. 41, No. 1, 2002.
8. Y. Eytani. "Concurrent Java Test Generation as a Search Problem". In *Workshop on Runtime Verification* (RV), *2005.*
9. Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. "Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs". Concurrency and Computation: Practice & Experience, to appear..
10. Erdös P., Ko C., and Rado R. “Intersection theorems for systems of finite sets.” In Quart. J. Math. Oxford 12 (1961), 313-318.
11. E. Farchi, Y. Nir, and S. Ur. “Concurrent Bug Patterns and How to Test Them.” In Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.
12. C. Flanagan and S.N. Freund. "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs." In Symposium on Principles of Programming Languages (POPL), 2004.
13. Godbole A. P., Skipper D. E., and Sunley R. A. “t-Covering arrays: Upper bounds and Poisson approximations.” In Combinatorics, Probability, and Computing 5 (1996) 105-117.
14. Hallal, H., Alikacem, E., Tunney, P., Boroday, S. and Petrenko, A. "Antipattern-based Detection of Deficiencies in Java Multithreaded Software" In Conference on Quality Software (QSIC2004), 2004.
15. 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.
16. D. Hovemeyer and W. Pugh. “Finding Bugs is Easy.” in *SIGPLAN Notices (Proceedings of Onward! at OOPSLA)*, 2004.
17. I. Rabinovitz, O. Grumberg. ”Bounded Model Checking of Concurrent Programs.” In Conference on Computer Aided Verification (CAV'05), 2005.
18. B. Long and P.A. Strooper. “A Classification of Concurrency Failures in Java Components.” Workshop on Parallel and Distributed Systems: Testing and Debugging, 2003.
19. R. Netzer and B. Miller. “Detecting Data Races in Parallel Program Executions.” In Advances in Languages and Compilers for Parallel Computing Workshop,, 1990.
20. S. Qadeer and J. Rehof. “Context-bounded model checking of concurrent software.” In Symposium on Tools and Algorithms for the Construction and Analysis of Systems, 2005.
21. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. “Eraser: a dynamic data race detector for multithreaded programs.” ACM Transactions on Computer Systems (TOCS), 15(4): 391–411, 1997.
22. V. Schuppan, M. Baur, A. Biere. “JVM Independent Replay in Java.” In Workshop on Runtime Verification (RV), 2004. Volume 113, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.
23. S. D. Stoller. "Testing Concurrent Java Programs Using Randomized Scheduling." In Workshop on Runtime Verification (RV), Volume 70(4), Electronic Notes in Theoretical Computer Science. Elsevier, 2002.
24. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. “Model checking programs. Automated Software Engineering Journal”, 10(2), 2003.