Adaptive Randomized Scheduling for Concurrency Bug Detection

Zan Wang*, Dongdi Zhang*, Shuang Liu*, Jun Sun† and Yingquan Zhao*
*College of Intelligence and Computing, Tianjin University, China
†School of Information Systems, Singapore Management University, Singapore
{wangzan, zhangdongdi, shuang.liu, zhaoyingquan}@tju.edu.cn, junsun@smu.edu.sg

***Abstract*—Multi-threaded programs often exhibit erroneous behaviours due to unintended interactions among threads. Those bugs are often difficult to find because they typically manifest under very specific thread schedules. The traditional randomized algorithms increase the probability of exploring infrequent interleavings using randomized scheduling and improve the chances of detecting concurrency defects. However, they may generate many redundant trials, especially for those hard-to-detect defects, and thus their performance is often not stable. In this work, we propose an adaptive randomized scheduling algorithm (ARS), which adaptively explores the search space and detects concurrency bugs more efficiently with less efforts. We compare ARS with random searching and the state-of-the-art maximal causality reduction method on 27 concurrent Java programs. The evaluation results show that ARS shows a more stable performance in terms of effectiveness in detecting multi-threaded bugs. Particularly, ARS shows a good potential in detecting hard-to-expose bugs.**

***Index Terms*—concurrency bugs, bug detection, concurrency bug pattern, adaptive random testing**

I. Introduction

With the development of multi-core processors, multi-threaded programming is more and more widely adopted nowadays. Although the efficiency brought by multi-threading is appreciated, it may also introduce concurrency bugs. Concurrency bugs are bugs which may only manifest with certain scheduling, i.e., they are heisenbugs which may only be observed if we execute the same program with the same input multiple times. Concurrency bugs are notoriously difficult to detect due to the non-determinism nature.

To expose multi-threaded bugs swiftly from the astronomically large number of possible schedules, researchers have proposed many concurrency bug detectors, which can be roughly classified into three categories, i.e., static detectors, dynamic detectors and hybrid detectors. Static detectors [1] analyse the concurrent programs by leveraging program structures such as control-flow graphs, which are built without executing the program. Static detectors explore an over-approximated state space and thus are able to detect the concurrency bugs in obscure code paths which can be difficult to reach with concrete executions. However, static detectors tend to report many false positives, which require manual effort to identify actual bugs. Contrary to static techniques, dynamic detectors [2]–[6] expose concurrency bugs, by monitoring memory accesses and synchronization operations at runtime. Dynamic detectors have a precise knowledge about the runtime behavior of the program and thus avoids the problem of false positives. However they can only observe a small subset of program states and leave many bugs undetected. To alleviate the limitations of static and dynamic techniques, researchers proposed to combine static and dynamic analysis to leverage the strength of both. Hybrid detectors [7]–[9] obtain program state information through static analysis to aid the dynamic analysis. As a result, fewer thread schedules are explored, which make the detectors more efficient.

One representative hybrid detector [7] is to combine model checking with static analysis. To solve the state space explosion problem, state space reduction methods, such as partial order reduction (POR) [10], have been explored. Previous researches [11] have shown that POR can reduce the redundant schedules to some extent. However, the reduction effectiveness of POR is limited by happens-before relations. To alleviate this problem, Huang [12] adopted maximal causality reduction (MCR) for model checking. It overcomes the happens-before problems by exploring the maximal causality between schedules to achieve the maximal reduction. However MCR requires constraint solving. Contrary to reducing the redundant schedules, the schedule bounding [13] techniques reduce the state space effectively by limiting the number of context switches between different threads. But the schedule bounding technique may miss bugs which only manifest with more-than-bound of number of context switches.

One of the most widely adopted dynamic detectors, the simple randomized detector [2], [3], [14] is proposed based on the observation that systems usually follow similar schedules rather than random schedules. Therefore, researchers attempt to control thread scheduling [15] by picking a random thread to execute at every synchronization. However, the random scheduling samples all possible thread interleavings randomly without considering the differences (in terms of likelihood of leading to bugs between different schedules).

To solve these problems, we propose a novel algorithm called *Adaptive Randomized Scheduling Algorithm* (ARS) to improve the performance of existing randomized detectors. ARS is inspired by adaptive random testing [16]. The assumption of ARS is that the more a trace deviates from the

* Shuang Liu is the corresponding author

978-1-7281-4646-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICECCS.2019.00021

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

passed/successful traces, the more likely that it leads to a bug. Unlike the simple randomized algorithm, ARS picks N farthest traces from the known passed trace at each scheduling point. N is a threshold set to control the "width" of the search space at each step. In this way, we guide the search to favour the traces that are likely to lead to bugs (i.e., the farthest to passed traces). ARS is designed to leverage the advantages of both random execution and guided exploration. Compared to dynamic detecting techniques, such as random testing, ARS is more effective, especially in detecting hard-to-expose bugs. Compared to static detecting techniques such as MCR, ARS is more efficient.

One central question in our approach is how to define the distance between traces. In this work, we propose to measure the distance by contrasting the memory-access patterns [17] in two traces. The reason is that it has been shown that memory-access patterns are often correlated to the root cause of multi-threaded bugs [2], [5]. Our work focuses on detecting non-deadlock defects in multi-threaded programs, which are reported to count for more than 65% [18] of all concurrency defects.

Our contribution is summarized as follows:

- We propose a novel concurrency bug detection algorithm called ARS, which adopts adaptive search guided by distance metrics to efficiently explore the state space of concurrent programs, with the purpose of exposing bugs efficiently.
- We propose to use memory-access patterns to measure the distance between traces, and subsequently to guide the searching procedure effectively.
- We implement ARS based on Java Pathfinder (JPF), our tool is made available online [19].
- We conduct experiments on 27 concurrent programs, to compare ARS with state-of-the-art concurrency bug detection algorithms, such as MCR. Results show that ARS is more stable on the effectiveness of detecting concurrency bugs. ARS also shows a good potential in detecting hard-to-expose bugs.

The rest of this paper is organized as follows. The preliminary is introduced in Section II. Section III uses a motivating example to illustrate the main idea of ARS. The details of ARS algorithm as well as our implementation are described in section IV. In Section V, we report our experiment setting and results. In Section VI, we discuss the threats to validity. Section VII discusses the related work and Section VIII concludes the paper.

II. Preliminary

A. Memory-Access Patterns

It is well known that concurrency bugs are difficult to reproduce due to unexpected schedules, and the randomness in scheduling brings difficulties in detecting and locating concurrency bugs. In recent research, memory-access patterns [17], [20] are proposed to capture the reasons of concurrency defects. It has been reported that memory-access patterns are often correlated to bugs [20], [21]. Park et al., abstract non-deadlock and univariate concurrency bug defects into 8 different memory-access patterns in Falcon [20]. UNICORN [17] further expands the patterns to 17 kinds to handle multivariate concurrent defects.

TABLE I
THE GENERIC MEMORY-ACCESS PATTERNS [17]

ID	Memory-Access Pattern
1	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\})$
2	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset)$
3	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\})$
4	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \{x\}, \emptyset)$
5	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \{x\}, \emptyset)$
6	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset), (t_a, s_k, \emptyset, \{x\})$
7	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \emptyset, \{x\})$
8	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \emptyset, \{x\})$
9	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_b, s_k, \emptyset, \{y\}), (t_a, s_l, \emptyset, \{y\})$
10	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{y\}), (t_b, s_k, \emptyset, \{x\}), (t_a, s_l, \emptyset, \{y\})$
11	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{y\}), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \emptyset, \{x\})$
12	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset), (t_b, s_k, \{y\}, \emptyset), (t_a, s_l, \emptyset, \{y\})$
13	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_b, s_k, \{x\}, \emptyset), (t_a, s_l, \emptyset, \{y\})$
14	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_b, s_k, \emptyset, \{y\}), (t_a, s_l, \{y\}, \emptyset)$
15	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{y\}), (t_b, s_k, \emptyset, \{x\}), (t_a, s_l, \{y\}, \emptyset)$
16	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{y\}), (t_a, s_k, \{y\}, \emptyset), (t_b, s_l, \emptyset, \{x\})$
17	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \{x\}, \emptyset)$

In this work, we adopt the set of 17 memory-access patterns defined in [17], which are shown in Table I. The second column of the table shows the memory-access patterns. Each memory-access pattern is a sequence of at most four steps in the test execution, which concerns only with two threads and at most two variables. Each step is a tuple of (t, s, R, W), where t is a thread id, s is a bytecode instruction generated by a statement in the program, R is a set of variables being read and W is a set of variables being written. For example, the memory-access pattern with id of 1 in Table I means that a read access of x by thread t_a in statement s_i is followed by a write access of x by thread t_b in statement s_j.

B. Adaptive Random Testing

Adaptive Random Testing (ART) [16], [22], [23] describes a family of algorithms of generating random test cases for sequential programs. The idea is based on a fact that the test cases which can cause the bugs are unevenly distributed in the input domain. Because of this skewed distribution, bugs may densely distributed in certain parts of the input domain. Based on this, there are assumptions that evenly distributed test cases are more likely to expose failures, i.e., with fewer test cases, than ordinary testing (where the distributions of test cases are not explicitly considered), and the efficiency might be higher if the distribution is taken into consideration. Chen uses F-measure [16] to realize ART, which represents the expected number of test cases required to detect the first failure, as the effectiveness metric.

Fixed Size Candidate Set Algorithm (FSCS) [16] is one of the adaptive random testing algorithms. FSCS maintains

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

```
1. public class Main {
2.   public static int account;
3.   public static class Customer implements Runnable {
4.     private int cost;
5.     public Customer(int cost) {
6.       this.cost = cost;
7.     }
8.     public void run() {
9.       int temp = account + cost;
10.      account = temp;
11.    }
12.  }
13.  public static void main(String[] args) {
14.    account = 0;
15.    Thread t1 = new Thread(new Customer(10));
16.    Thread t2 = new Thread(new Customer(10));
17.    t1.start();
18.    t2.start();
19.    assert(account == 20) : "wrong!";
20.  }
21. }
```

Fig. 1. The Motivating Example

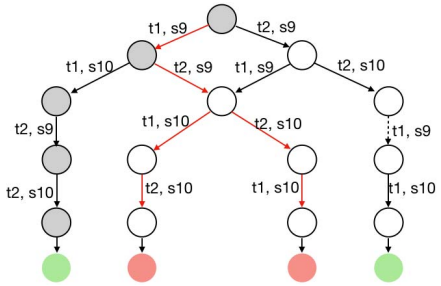

Fig. 2. State Diagram of the Motivating Example

two sets of test cases. One set is the set of all executed test cases, which is denoted by $E = \{e_1, e_2, ..., e_n\}$. The other set is the *candidate set*, denoted by $C = \{c_1, c_2, ..., c_k\}$. The *candidate set* contains k randomly generated inputs and k is fixed throughout the testing process. A candidate will be selected as the next test case if it has the largest distance to its nearest neighbour in E. The basic idea of FSCS is that test cases should be as evenly distributed over the entire input space as possible in order to achieve a small F-measure value.

III. A Motivating Example

In this section, we illustrate our Adaptive Randomized Scheduling (ARS) algorithm with a simple motivating example, shown in Figure 1.

This program has one shared variable $account$, which has an initial value of 0 (line 14). The local variable $temp$ is added to make sure that all of the statements in this example are atomic. The main thread creates two threads in this example, each of which executes a piece of code accessing the shared variable $account$. When statement 9 and line 10 of one thread are interleaved with statement 9 or 10 by the other thread, the atomicity of the line 9 and 10 is violated, which leads to an assertion failure (line 19). This is a typical data race.

In order to explain the entire iterative search process of ARS, we draw the state diagram of the example program (Figure 1) in Figure 2. In Figure 2, nodes represent the states of the program (which include all the shared variable evaluations and the program counter for each thread of the program). The labels on the edges represent the transitions caused by the corresponding thread and statement. A transition is triggered when an instruction is executed. The target state is decided based on the evaluation of the variables as well as the next instruction of each thread to be triggered.

ARS starts with executing a randomly generated schedule until completion. In this example, we assume that the first trace found is trace the nodes of which are highlighted in grey in Figure 2. Note that in Figure 1, one statement can be executed by multiple threads, we use $tr_{s_j}^{t_i}$ to represent a transition which is the result of statement s_j executed by thread t_i. We represent a trace with the sequence of transition labels. For example the trace highlighted in grey in Figure 2 is represented as $C_0 = \langle tr_{s_9}^{t_1}, tr_{s_{10}}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2} \rangle$. We collect the memory-access information of trace C_0 based on the patterns shown in Table I. Following the notation used in [17], [20], we add an extra rule when constructing patterns, i.e., a pattern should not step over write operations on the same shared variable, since the write operation will overwrite the value of shared variable. The patterns of C_0 can be expressed as a set PS_0, which is shown as follows:

$$PS_0 = \{[(t_1, s_{10}, \{\}, \{account\}), (t_2, s_9, \{account\}, \{\})], \\ [(t_1, s_{10}, \{\}, \{account\}), (t_2, s_{10}, \{\}, \{account\})]\} \quad (1)$$

PS_0 contains 2 patterns, each of length 2. Recall the definition of memory-access patterns in Section II-A, the pattern $[(t_1, s_{10}, \{\}, \{account\}), (t_2, s_9, \{account\}, \{\})]$ means that thread 1 writes the shared variable $account$ at statement 10 and then thread 2 reads the value of $account$ at statement 9.

ARS then selectively and adaptively explores the nodes in the search space in a way which is controlled by distance between traces as well as the pool size N, which we will describe in detail in section IV-B. At each node, ARS calculates the pattern set PS' of every sequence of transitions which have been executed, and then compares the numbers of different elements of set PS_0 and PS' (distance). The "distance" is defined as the number of different patterns between PS' and PS_0 (i.e., $PS' \setminus PS_0$), and the details of the definition will be discussed in section IV-B. At each state, the algorithm selects N farthest nodes (from the current passing traces) and continues searching from the selected nodes. We set the N to be 2 in this example (and discuss the selection of N in detail in Section V). ARS continues exploring adaptively until it finds an error or the program terminates.

In the example of Figure 2, suppose we have explored the passed trace C_0 and the current passed trace set is $C_p = \{C_0\}$. In the first iteration, there are two sub-traces, i.e., $\langle tr_{s_9}^{t_1} \rangle$ and $\langle tr_{s_9}^{t_2} \rangle$. Since $\langle tr_{s_9}^{t_1} \rangle$ and $\langle tr_{s_9}^{t_2} \rangle$ do not match any patterns in Table I, the pattern set of both are $\{\}$. Therefore the distance between $\langle tr_{s_9}^{t_1} \rangle$ and C_p, $\langle tr_{s_9}^{t_2} \rangle$ and C_p are both 0. ARS records both of the sub-traces in the $queue$ and proceeds to the second iteration. In the second iteration, there are 4 sub-traces, $queue = \{\langle tr_{s_9}^{t_1}, tr_{s_{10}}^{t_1} \rangle, \langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2} \rangle, \langle tr_{s_9}^{t_2}, tr_{s_9}^{t_1} \rangle, \langle tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2} \rangle\}$. These 4 sub-traces also do not match any patterns, and their $distances$ from C_p are all 0. Since only 2 sub-traces can be

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

kept in $queue$ based on N, the algorithm randomly selects 2 sub-traces (from the pool of 6 traces, i.e., the existing 2 traces in the queue and the 4 newly generated traces), suppose they are $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}\rangle$ and $\langle tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}\rangle$. ARS then adds them in $queue$ before the next iteration starts.

In the third iteration, we get 3 sub-traces, $queue = \{\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_1}\rangle, \langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}\rangle, \langle tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}, tr_{s_9}^{t_1}\rangle\}$. Trace $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_1}\rangle$ has the patterns $PS_1 = [(t_2, s_9, \{account\}, \{\}), (t_1, s_{10}, \{\}, \{account\})]$. Trace $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}\rangle$ has the patterns $PS_2 = [(t_1, s_9, \{account\}, \{\}), (t_2, s_{10}, \{\}, \{account\})]$. Trace $\langle tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}, tr_{s_9}^{t_1}\rangle$ has the patterns $PS_3 = [(t_2, s_{10}, \{\}, \{account\}), (t_1, s_9, \{account\}, \{\})]$. Note that pattern $[(t_1, s_{10}, \{\}, \{account\}), (t_2, s_9, \{account\}, \{\})]$ and $[(t_2, s_{10}, \{\}, \{account\}), (t_1, s_9, \{account\}, \{\})]$ are equivalent. The distances of $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_1}\rangle$, $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}\rangle$ and $\langle tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}, tr_{s_9}^{t_1}\rangle$ with C_p are 2, 2, and 1, respectively.

$\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_1}\rangle$ and $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}\rangle$ have bigger distances to C_p so we add them in the queue and expand continuously. In the next iteration, the program terminates and we get 2 traces $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_1}, tr_{s_{10}}^{t_2}\rangle$ and $\langle tr_{s_9}^{t_1}, tr_{s_9}^{t_2}, tr_{s_{10}}^{t_2}, tr_{s_{10}}^{t_1}\rangle$ (highlighted in red in Figure 2), both of which lead to errors. The search completes in this example since an error occurs. If no error occurs in this iteration, the passed traces will be added to the passed trace set C_p and used to calculate the distance in the following iterations.

It takes ARS 2 complete execution runs to find the first bug in the example program with $N = 2$, and some partial traces (marked in dotted transitions in Figure 2) are terminated by the algorithm. This simple example shows how our ARS algorithm works. Note that in the motivation example, only one passed trace is explored before the bug is detected. In reality, there may be more than one passed traces explored before a bug is detected, in such cases, subtraces which are maximally different from all of the passed traces are selected. More experiments are conducted to evaluate the effectiveness of ARS in section V.

IV. Our approach

In this section, we describe the Adaptive Randomized Scheduling (ARS) algorithm in detail. The ARS algorithm combines the random testing and adaptive random testing to control thread scheduling adaptively. The aim of ARS is to detect concurrency bugs more quickly with less efforts.

A. Definition of Distance between Traces

Before introducing the overall algorithm of ARS, we first describe the distance metric used in ARS. There are existing approaches [16], [24]–[26] which use the Euclidean distance to measure distances between test cases. However these approaches do not consider the effect of different schedulings, and thus cannot find concurrent bugs effectively. ARS considers the scheduling information of threads by adopting memory-access patterns [17]. The results of UNICORN [17] show that memory-access patterns are often correlated to concurrency bugs. Therefore, in ARS, we are motivated to define the distance metrics based on memory-access patterns. The intuition is that the patterns of a failure-inducing trace should be different (and thus has a large distance) from the patterns of a successful trace.

Definition 4.1 (Distance between traces):

$$dist(trace_a, trace_b) \doteq |P_{trace_a} \setminus P_{trace_b}|$$

In the definition, $trace_a$ and $trace_b$ are two traces, and their pattern sets are P_{trace_a} and P_{trace_b}, respectively. The distance between the two traces is defined as the difference of their corresponding pattern sets. According to the definition, the more different the patterns of two traces, the larger the distance value.

Let $Tr = \{tr_1, tr_2, ..., tr_n\}$ be a set of traces. We can formally define the distance between one trace and a trace set as follows.

Definition 4.2 (Distance between a trace and a trace set):

$$dist(trace, Tr) \doteq \min\{dist(trace, tr_i) | tr_i \in Tr\}$$

The distance of $trace$ and a trace set Tr is defined as the minimum of distances between $trace$ and each element Tr_i in Tr. The definition guarantees that if the distance between a trace $trace$ and a trace set Tr is large, then every trace in the trace set Tr has a large distance from $trace$. For the simplicity of notation, we use the short format $\min_i dist(trace, tr_i)$ to represent the equation $\min\{dist(trace, tr_i) | tr_i \in Tr\}$ hereafter.

Let $C = \{c_1, c_2, ..., c_m\}$ be the set of existing passed traces, and $Q = \{q_1, q_2, ..., q_n\}$ be the set of current sub-traces, We give the formal definition of the distance between the two sets as follows.

Definition 4.3 (Distance between two trace sets):

$$dist(Q, C) \doteq \max\{dist(q_i, C) | q_i \in Q\}$$
$$\doteq \max_i \min_j |P_{q_i} \setminus P_{c_j}|$$

Intuitively, Definition 4.3 tries to find a trace in Q, which has the maximum distance with all the traces in the passed trace set C.

In this work, we propose an effectiveness metric, called P-measure. It is defined as the number of executions required to detect the first concurrency bug. More specifically, P-measure means how many traces the algorithm needs to explore in order to find the first defect. P-measure is formally defined in Definition 4.4.

Definition 4.4 (P-measure):

$$P = |C| + 1$$

In Definition 4.4, $|C|$ is the number of passed traces that ARS has explored when the first defect is detected. Intuitively speaking, the P-measure reflects the effectiveness of a testing strategy in terms of the number of complete runs required to expose the bug. In practice, when a failure is detected, the testing is normally stopped and debugging will start.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1: Adaptive Randomized Scheduling Algorithm

```
1  function heuristicSearch(P, N):
2      T_passed ← ∅;
3      trace ← execute P randomly;
4      if trace is failed then
5          return {trace}, T_passed;
6      T_passed ← T_passed ∪ {trace};
7      T_failed ← ∅;
8      while T_failed = ∅ do
9          T_failed, Passed ←oneIteration(T_passed, N);
10         T_passed ← T_passed ∪ Passed;
11     return T_failed, T_passed;
12 function oneIteration(T_passed, N):
13     SubT ← execute P for one step;
14     T ← ∅;
15     Queue ← empty queue;
16     enQueue(Queue, SubT);
17     while Queue ≠ ∅ do
18         sort Queue by distance desc;
19         Queue ← the first N elements of Queue;
20         subtrace ← Queue.dequeue();
21         SubT ← execute subtrace for one step;
22         T' ← all of the completed traces in SubT;
23         T ← T ∪ T';
24         enQueue(Queue, SubT \ T);
25     T_passed ← all of the passed traces in T;
26     T_failed ← all of the failed traces in T;
27     return T_failed, T_passed;
28 function distance(trace, T_passed):
29     Set_distance ← ∅;
30     for t in T_passed do
31         distance ← |patterns(trace) \ patterns(t)|;
32         Set_distance ← Set_distance ∪ {distance};
33     return min(Set_distance);
```

The testing phase would normally be resumed only after the detected defect is fixed. Therefore, the P-measure is more intuitive for debugging tasks in practice than measurements which focus on the number of bugs that can be found. Our work targets finding the first bug as soon as possible. However, ARS is able to detect multiple bugs if it continues with the detection process.

B. The Adaptive Randomized Scheduling Algorithm

The ARS algorithm is shown in Algorithm 1. ARS assumes that the inputs of the tested program are already given, and does not change during the execution of algorithm. The task of ARS is to find the buggy schedules for the given inputs.

There are three functions in Algorithm 1. The *heuristicSearch* function takes a program (P) and a given threshold N (which will be described in detail shortly) as inputs. *heuristicSearch* calls function *oneIteration* in every iteration. The *oneIteration* function takes the set T_{passed}, which contains all of the traces the algorithm has already explored, and the threshold N as inputs. The *distance* function calculates the distance between each sub-trace $trace$ and passed trace set T_{passed}, as defined in Definition 4.2.

The *heuristicSearch* function controls the work flow of the ARS algorithm. It first executes the program P (on a given set of inputs) randomly (line 3). The algorithm stops and returns the failed trace if we encounter a failed trace during executions (line $4-5$). Otherwise, a passed trace $trace$ is found and then is added to the passed trace set T_{passed} (line 6). *heuristicSearch* starts the iteration step (by calling the *oneIteration* function) until a failed trace is found (line $8-10$).

The *oneIteration* function from line 12 to line 27 conducts a search, which collects one or more traces. *oneIteration* takes the set of passed traces T_{passed} and explores the state space of program P until it finds one or more completed traces T. The passed traces in completed trace set T are labeled $Passed$ and added to T_{passed} (line 25). The failed trace, if encountered, are added to T_{failed}. *oneIteration* first executes P for one step (line 13), which explores all transitions available in the initial state of P and records them in a sub-trace set $SubT$. A sub-trace is a sequential list of states, the last state of a sub-trace is not required to be a final state. Then $SubT$ is added into a priority queue $Queue$ (line 16). In the while loop from line 17 to line 24, ARS first sorts the queue based on the distance information of each sub-trace (calculated based on function distance from line $28-33$) and only keeps the first N sub-traces that has the largest distances (line $18-19$). Then ARS dequeues a sub-trace $subtrace$ from the $Queue$ (line 20) and continues exploring $subtrace$ one step forward (line 21) by running P following the path recorded in $subtrace$. The exploration stops when the final state of a $subtrace$ is reached. The result of exploration is a set of sub-traces $SubT$. Then all (completed) sub-traces in $SubT$, i.e., traces that reach the final state, are added to T (line $22-23$), the others are added to $Queue$ (line 24). The while loop continues until the $Queue$ becomes empty. The set T contains all passed traces T_{passed} and failed traces T_{failed} explored in the current iteration when ARS finishes.

The *distance* function from line 28 to line 33 is an implementation of Definition 4.2, which is used to calculate the distance between a trace $trace$ and a set of passed traces T_{passed}. This function is not called explicitly in the algorithm, but rather is called when $Queue$ is sorted at line 18. Before calculating the distance, it first matches the patterns of $trace$ (represented by patterns($trace$)) and the patterns of every element s_i in T_{passed} (represented by patterns(s_i)) based on the generic memory-access patterns shown in Table I. Then the distance between $trace$ and each element s_i in T_{passed} can be calculated based on Definition 4.1 (line 31). All of the distances are added in $Set_{distance}$ (line 32). Finally the minimum number of $Set_{distance}$, i.e., the distance between $trace$ and T_{passed}, is returned.

A strict scheduling technique is required for this algorithm. For example, at line 21, the tested program P needs to be executed strictly following the order of $subtrace$ first, and then extends one more state to produce several new sub-traces in $SubT$. This means the implementation of ARS algorithm should have the ability of reproducing a certain trace or recording and restoring program execution state. The detailed implementation is described in the next section.

C. Implementation

Our implementation is in Java and leverages on Java Pathfinder (JPF) [27], a model checker developed by NASA with the aim of verifying Java programs. JPF is capable

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

of model-checking concurrent programs and it has build-in state space reduction techniques, like partial order reduction. JPF provides excellent supports for concurrent programs. It provides a build-in scheduler which allows exploring different scheduling combinations, and also supports user-defined scheduling policies. JPF can identify states in the program from where executions can diverge, then explore them systematically. That is to say, JPF explores all execution paths of a program. Typical variances supported are different scheduling traces or random values, meanwhile, JPF allows users to introduce their own type of inputs or state machine events. JPF records the complete execution history, including fine grained bytecode instructions, of a trace leading to defects.

In the implementation of ARS, the most important task is to control the scheduling of threads. With JPF, we can control thread scheduling and get the runtime information of the Java program. In fact, JPF traverses all the states of a program in a top-down manner. At run time, JPF generates scheduling points for possible thread switching positions. At each scheduling point, it generates one option for each target that can be switched. By default, it chooses the first one each time. In our implementation, we choose the option which has the largest distance to the previously explored traces.

V. Empirical Evaluations

A. Experiment Setup

The experiments are conducted on a computer with a 2-core, 3.20GHz Intel i5 CPU and 8GB physical memory. The operating system is Windows 10 and the java version is JDK 1.8. The heap space used to run the experiments is set to 2GB.

We evaluate the ARS algorithm on a variety of multi-threaded Java programs collected from recent related publications [12]. These programs are widely adopted as benchmarks by research work on concurrency defect detection, location and fixing.[1] The details of the benchmark programs are shown in column $1-4$ of Table II. The first column is the program name. The second column reports the total lines of code, column 3 shows the number of threads in each program and column 4 is the type of concurrency defects exist in the program. Each program has at least one known error that causes runtime exceptions under certain thread scheduling.

We ran each of the benchmark programs 100 times with ARS and the simple randomized algorithm (SR), which is an implementation of randomized scheduling method [15]. We also compare ARS with maximal causality reduction (MCR) [12], which is the state-of-the-art for stateless model checking and can be used to find concurrent bugs. In [12], it has been shown that MCR is much more effective than alternative approaches like advanced partial order reduction and approaches based on bounded context switch. Since MCR utilizes constraint solving, which is time-consuming, we ran each program with MCR for 10 times. In each run, the program stops when the first error is found, and the number of executions is recorded.

[1]The programs we choose as our benchmark are constrained to programs which JPF can support.

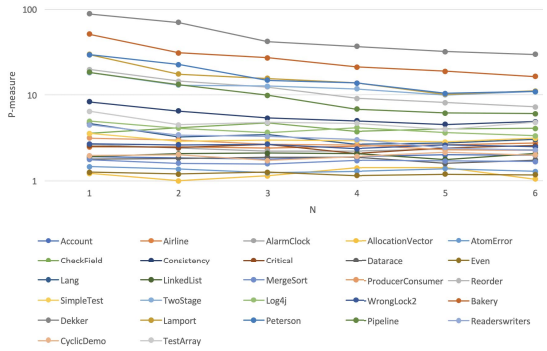

Fig. 3. Evaluation of P-measure with different N values

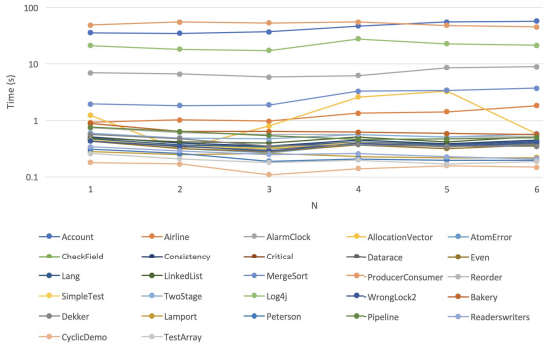

Fig. 4. Evaluation of time with different N values

B. Research Questions

By conducting the experiment evaluations, we aim to answer the following three research questions.

RQ1: How does threshold N impact the defects detection effectiveness and efficiency of ARS? In our approach, the threshold N controls the searching breadth of ARS algorithm. Theoretically, the larger N is, the slower the search is for each run, but the higher probability of finding bugs in each run. We want to explore what is a good N that can balance the two aspects, i.e., to find the first bug with fewer runs and less time. To answer this research question, we set the value of N from 1 to 6 and report the average P-measure as well as execution time for different values of N.

RQ2: How effective is ARS in detecting concurrent defects? The effectiveness of ARS in detecting concurrent defects is the key concern of our approach. For this research question, we measure if ARS can detect the concurrency bugs in the benchmark programs effectively, i.e., whether the concurrency bugs can always be detected in an acceptable number of iterations. We also compare ARS with the simple random search (SR) algorithm as well as the maximal causality reduction (MCR) algorithm with the same experiment settings.

RQ3: How efficient is ARS in detecting concurrent defects? As the third research question, we want to measure whether ARS is able to detect concurrency bugs efficiently, i.e., always detect bugs in an acceptable amount of time. We also compare ARS with SR and MCR with the same experiment settings.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

TABLE II
RESULTS COMPARED WITH SR AND MCR

Program	LoC	#Thrd	Type	# Avg P-measure							# Avg Time (s)		
				ARS	SR	MCR	ARS vs. SR		ARS vs. MCR		ARS	SR	MCR
							p	δ	p	δ			
Account	373	10	atomicity	2.00	2.00	9.2	1.000	0.00	**0.001**	**-0.84**	55.63	**3.76**	1660.60
Airline	136	20	order	2.60	18.10	8.0	**0.001**	**-1.00**	**0.000**	**-1.00**	**1.43**	5.92	187.53
AlarmClock	351	8	race	2.34	3.58	-	**0.003**	**-0.87**	**0.000**	**-1.00**	8.66	**1.49**	-
AllocationVector	348	2	atomicity	1.41	1.02	4.0	0.796	-0.08	**0.001**	**-0.80**	3.31	**0.87**	1.38
AtomError	67	2	race	1.38	1.44	1.0	0.895	0.06	**0.001**	**1.00**	0.35	0.43	**0.15**
CheckField	44	2	atomicity	4.01	3.72	3.0	0.671	0.19	**0.001**	**0.80**	**0.39**	1.14	0.69
Consisitency	35	3	atomicity	4.51	5.51	2.0	0.391	-0.33	**0.000**	**1.00**	**0.39**	1.67	0.49
Critical	70	2	race	2.39	3.35	2.0	**0.023**	**-0.68**	**0.012**	**0.60**	0.35	1.03	**0.26**
Datarace	118	2	atomicity	1.59	2.10	7.0	**0.019**	**-0.71**	**0.000**	**-1.00**	**0.34**	0.65	16.24
Even	63	4	order	1.19	1.23	2.0	0.774	-0.14	**0.000**	**-1.00**	**0.32**	0.38	1.86
Lang	2444	2	atomicity	2.73	3.60	8.0	**0.026**	**-0.66**	**0.000**	**-1.00**	**0.37**	1.10	3.26
LinkedList	299	2	atomicity	1.76	1.68	6.9	0.895	0.07	**0.000**	**-1.00**	**0.42**	0.54	11.28
MergeSort	399	2	race	1.70	1.67	2.0	0.671	0.22	**0.002**	**-0.70**	3.44	**1.11**	426.21
ProducerConsumer	194	7	atomicity	2.30	3.39	-	**0.007**	**-0.81**	**0.000**	**-1.00**	48.58	**4.20**	-
Reorder	92	4	atomicity	8.16	58.77	4.0	**0.001**	**-1.00**	**0.000**	**1.00**	**0.51**	22.19	0.74
SimpleTest	47	2	atomicity	2.83	5.32	4.9	**0.001**	**-0.95**	**0.000**	**-1.00**	**0.35**	1.63	0.64
TwoStage	140	2	race	9.99	23.65	2.0	**0.001**	**-0.96**	**0.000**	**1.00**	**0.51**	7.33	0.75
Log4j	18799	4	order	3.63	7.15	38.0	**0.001**	**-1.00**	**0.000**	**-1.00**	22.92	**7.15**	24.79
WrongLock2	36	5	race	2.60	2.93	-	0.671	-0.20	**0.000**	**-1.00**	**0.35**	0.90	-
Bakery	111	2	atomicity	18.93	47.93	-	**0.001**	**-0.96**	**0.000**	**-1.00**	**0.59**	6.88	-
Dekker	100	2	atomicity	32.05	89.74	-	**0.000**	**-1.00**	**0.000**	**-1.00**	**0.35**	9.05	-
Lamport	150	2	atomicity	10.18	29.64	-	**0.000**	**-1.00**	**0.000**	**-1.00**	**0.22**	2.64	-
Peterson	77	2	atomicity	10.38	36.58	8.00	**0.000**	**-1.00**	**0.000**	**1.00**	**0.20**	3.73	4.75
Pipeline	119	7	race	6.14	24.81	-	**0.000**	**-1.00**	**0.000**	**-1.00**	**0.47**	2.47	-
Readerswriters	598	3	race	2.42	6.07	10.00	**0.000**	**-1.00**	**0.000**	**-1.00**	**0.23**	0.76	66.02
CyclicDemo	54	4	atomicity	2.14	2.17	8.0	0.852	-0.09	**0.000**	**-1.00**	**0.16**	0.20	5.53
TestArray	46	2	atomicity	3.95	7.14	-	**0.003**	**-0.88**	**0.000**	**-1.00**	**0.17**	0.77	-

C. Answer to Research Questions

RQ1: How does threshold N impact the defects detection effectiveness and efficiency of ARS?

To evaluate the effect N has on the effectiveness and efficiency of detecting defects, we record the average of P-measure as well as execution time for each program with different values of $N \in \{1, 2, 3, 4, 5, 6\}$. Figure 3 and Figure 4 summarize the results of the experiments. Results of P-measure on different N values are shown in Figure 3 and the results of time are shown in Figure 4. From the results of Figure 3, we can observe the threshold N has some impact on the defect detection efficiency of ARS. For some programs (e.g., AlarmClock and Bakery), there is an obvious trend of P-measure decreasing with the increase of N. However, for some other programs, such as Account and MergeSort, there is no obvious trend. N can be viewed as controlling the searching width, therefore, the results indicate that there are certain traces which have larger distances from the successful traces and thus they are always selected by ARS to explore. This reflects that memory-access pattern can act as a good distance metric to guide the state space exploration.

The effect of N on the execution time is shown in Figure 4. Intuitively, the time value is affected by multiple factors. A larger N number means more options for searching in each iteration. Therefore bigger N is likely to cause the decrease of P-measure values. On the other hand, the larger N is, the more options need to be maintained for expanding, which is time consuming. We can observe that generally speaking, execution time does not show a growing trend with the increase of N. For some programs (e.g., Account and MergeSort), there are many branches (sub-traces) which have the same distance with existing passed traces, and ARS behaves similarly with the breadth-first search. As a result, ARS explores more branches with the increase of N, and the execution time increases with the increase of N. For some other programs (e.g., Lang and SimpleTest), there exists one branch that has a larger distance (with existing passed traces) than other branches, and ARS behaves similar to depth-first search. Therefore, the exploration is guided to follow that branch (which has the largest distance value) and the buffer queue length N does not have much impact in this case. As a result, the time required does not show an increasing trend with the value of N.

With a comprehensive consideration on the results of P-measure values and the execution time, we decide to set N to 5, which gives us a good execution time, as well as a small P-measure value (which can also be interpreted as stability of effectiveness in detecting bugs, especially in those hard-to-detect bugs) in detecting defects. In all of the following evaluations, the value of N is set to 5.

RQ2: How effective is ARS in detecting concurrent defects?

Table II shows the comparative experiments' results between ARS, SR and MCR. All the results are obtained with N set to 5 for ARS. We conduct the comparisons in two aspects, i.e., the average P-measure and the average time used. The values for ARS and SR are obtained with an average of 100 executions and MCR an average of 10 executions (as MCR

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

takes long time to finish one execution). Columns $5-11$ show the average P-measure results and columns $12-14$ show the average execution time used to detect the first bug.

In our evaluation, we use P-measure to measure the effectiveness of bug detection. Recall that P-measure is defined as the number of passed traces explored when the bug is detected. It can also be interpreted as the number of execution runs used. P-measure can be used to indicate the stability of effectiveness of the defect detection method. The intuition behind this metric is that we want to evaluate whether a bug detection method can consistently expose bugs in various different programs, especially when the bug is hard to detect. To check whether the results of ARS have significant differences comparing to the other two methods, we adopt hypothesis testing to test the significance of results between ARS, SR and MCR in terms of P-measure.

We conduct Wilconxon signed-rank test [28] on the results of the experiment, and we use Bonferroni correction [29] to correct P-value. Wilconxon signed-rank test is a non-parametric hypothesis testing method, which is used to test the null hypothesis that the median difference (of paired data) is equal to zero. The confidence is set to 0.05, a p-value smaller than 0.05 means rejecting the null hypothesis, i.e., the median difference is equal to 0. If the value is greater than 0.05, then the null hypothesis is accepted, which means these two datasets have no difference. We also use Cliff's delta [30] (colunm 9 and 11 in Table II), which is an effect size measurement to compliment hypothesis testing and provide the magnitude of differences between observations from different datasets. The value interval of δ is $[-1, 1]$, and the absolute value of δ represents the significance of differences. The value of δ greater than 0.474 indicates higher effectiveness, i.e., larger magnitude on the significance. smaller δ values indicates lower effectiveness.

Columns $2-4$ in Table II provide the average P-measure of ARS, SR and MCR, respectively. ARS and SR manage to detect the defects in all programs in a few iterations, whereas there are three cases that MCR fails to find the defects in the given time limit (1 hour). We also analyse the significance of the results which are shown in columns $5-8$. The p values in column 5 and 7 are the results of Wilconxon signed-rank test. The p-value smaller than 0.05 (highlighted in bold font) indicates a significant difference between the P-measure of different methods. The results of Cliff's Delta is shown is column 9 and 11, which can measure the magnitude of difference according to the effectiveness levels. We highlight the value of δ is the effectiveness level is "large", i.e., greater than 0.474. In particular, δ is highlighted in green if ARS is significantly better, and in red if it is the opposite.

The Wilconxon signed-rank test results show that ARS has significantly better performance than SR (on P-measure) on 17 programs (as highlighted in bold in the 5th column) and there is no significant difference on the other 10 programs. As for Cliff's Delta there are 17 programs that ARS performs better with effective level of large, 1 program that ARS performs better with effective level of medium, 1 program that ARS performs better with effective level of small, 2 programs that SR performs better with effective level of small, and the effectiveness level is negligible for other 6 programs. From the results we can conclude that ARS performs significantly better than SR in most of the benchmark programs, and ARS is more stable in exposing bugs than SR for those programs that has hard-to-detect bugs, such as Reorder, where ARS uses only one seventh iterations of that of SR.

In general, ARS also shows better performance in P-measure compared to MCR. In our experiment, the performances of ARS and MCR have significant difference on all 27 programs in Wilconxon signed-rank test, and all programs have effectiveness level of large in Cliff's Delta. There are 20 out of 27 cases where ARS outperforms MCR in P-measure, and MCR performs better on the other 7 programs. MCR explores traces by finding new values of shared variables in a (relatively) fixed order. For these 7 programs, MCR performs better because bugs expose when exploring the first few new values. From the experiment results, we can observe that, ARS is effective in detecting concurrency bugs, and it performs more consistently than SR and MCR.

To further zoom into the programs that ARS performs well, we provide some hints on the characters of programs that ARS shows good performance. By further analyzing the tested programs we can conclude that: (1) ARS is good at detecting the bugs which has rare or hard-to-trigger conditions. For instance the Reorder, Bakery and Dekker program in our benchmark. Comparing to all of possible traces in their state space, the traces that can trigger the bugs are rare. This kind of bugs are difficult to expose using randomized scheduling since the buggy traces are not evenly distributed in the space. However, ARS is good at exposing this kind of bug guided by our distance metric. (2) ARS is good at programs which have traces distinguishable by memory-access patterns. There are some programs in our benchmark such as Airline and log4j that falls in this category. For this kind of programs, the traces that can expose bugs contain bug-leading memory-access patterns which are missing in traces that do not expose. This is consistent with the main idea of ARS, which is to guide the search by the distance based on memory-access patterns.

RQ3: How efficient is ARS in detecting concurrent defects?

Columns $8-11$ in Table II report the comparison results on time. We can observe that ARS performs better than SR in 21 programs and worse in 6 programs. ARS has longer execution time on Account, ProducerConsumer and Log4j. These programs access shared variables frequently, which causes significant overhead for monitoring. Moreover, the buggy traces occur frequently in the search space, i.e., the buggy scheduling is easy to produce, and SR can detect them easily through random sampling. Therefore, for those 3 programs, the monitoring overhead of ARS dominates the total execution time. ARS performs better in 25 programs and comparable or worse in 2 programs compared to MCR. Note that ARS manages to detect bugs of 20 programs in less than 1 second. Comparatively, MCR takes a longer time

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

(2 up to 500 times longer than ARS) to detect bugs in general, and especially for some programs (e.g., 1,660 seconds for Account, 187 seconds for Airline and 426 seconds for MergeSort). MCR also fails to detect bugs of some programs (e.g., AlarmClock, ProducerConsumer and WrongLock2) in the time limit (1 hours) we set. From the table we can conclude that ARS is relatively more effective and stable in detecting concurrency bugs, while both SR and MCR show long execution times in some of the programs and the variance in execution time is large.

From the experiment results, we can conclude that ARS is in general more effective, efficient and stable than both SR and MCR in detecting concurrency bugs.

VI. Threats to Validity

In this section, we discuss the threats to validity of our work and provide potential mitigation plans.

Bias of benchmarks A key external threat to validity is that it is hard to obtain benchmarks that are representative of the multi-threaded Java programs in practice. This is a classical issue with concurrency program evaluation. In our work, we collect benchmarks from existing work on concurrency bug detection and fixing, with the hope that our evaluation results and comparison with other works are fair.

Limitations of framework In our approach, we utilize Java Path Finder (JPF) [27] for scheduling control. Due to the limitations of JPF, some large-scale programs can not be executed. The current implementation is a proof-of-concept implementation and JPF is chosen due to its flexible scheduling control interface. We can potentially implement our approach in a light-weight framework such as Soot [31] and ASM [32].

Comparison with Existing Approaches In our evaluation, the simple randomized algorithm (SR) and maximal causality reduction (MCR) are chosen as the baseline comparison algorithms for concurrency bug detection on Java. We do not conduct a thorough comparison with all existing concurrency bug detection algorithms [10], [13], [14]. However, SR and MCR are two most representative algorithms for concurrency bug detection on Java. MCR has been compared with various existing concurrency bug detection methods [10], [13] and shown a better performance. Therefore, we only pick SR and MCR to compare with.

Internal threats There are also some internal threats to validity, i.e., threats from our algorithm and parameter selection, that could affect the evaluation results. For example, the threshold N may affect the performance of ARS. To mitigate such effects, we conduct an experiment with the threshold N (Figure 3 and Figure 4) and choose the N value with the best performance. The experimental results may also be affected by the initial success trace. According to the design of ARS algorithm, the greater number of different patterns, the more likely the trace is chosen. ARS expands the set of passed traces at the end of each iteration by involving the newly observed passed trace, and this may partially mitigate the threats.

VII. Related Work

Concurrency bug detection There have been a large number of works on concurrency bug detection [1], [4], [8], [33]–[36]. Researches on concurrency bug detection usually concentrate on three problems, i.e., (1) how to improve the efficiency of the detection; (2) how to improve the effectiveness of the detection; and (3) how to reduce false positives. The happens-before [33], lockset algorithms [1], [35] and many of the extensions [4], [8] have been proposed for concurrency bug detection. Happens-before analysis and lockset algorithm are widely used to detect bugs of concurrent programs. RaceChecker [33] is a data race detector. It uses happens-before relation to prune the infeasible races before potential races are required to be verified. The lockset algorithm is then being refined to reduce overhead, eliminate false positives and solve some other problems [1], [35]. Some methods [37]–[39] combine lockset algorithm with happens-before to detect multithreaded bugs. RaceTrack [37] uses happens-before technique to analyze threads access to shared memory, and lockset is used when a corresponding location is found to be accessed concurrently by multiple threads. However, since static analysis methods don't use runtime information of concurrent programs, lockset and happen-before analysis methods are difficult to reduce false positive comparing to dynamic methods. Many innovative approaches, such as training [40] and interleaving testing [3], [5], [41], have been proposed to address the false positive problem in concurrency bug detection.

Random testing (or fuzz testing) is widely used to test sequential programs. Random testing is in general efficient in finding common bugs. In concurrent bug detection, random testing is always used to control thread scheduling. PCT [15] is a randomized scheduling method, it uses a disciplined schedule-randomization technique to provide efficient probabilistic guarantees of finding bugs. Other method [2], [3], [14] do some optimization on the basis of the randomized testing algorithm. Although random testing has been quite successful in finding bugs, it suffers from the problem of redundant exploration, i.e., the same (non-buggy) trace may be executed multiple times, which makes it hard to find bugs that hide in vague schedules. ARS is aimed to solve this problem, it introduces the concept of adaptive on simple random testing. There won't be a trace that executed repeatedly and it can detect defect more efficiently.

Adaptive random testing. ART is a slight modification to the random test strategy, using the information of test case selected before to guide the test cases to be selected later. So that the test cases are distributed as evenly as possible throughout the input space. The aim of ART is to improve the capability of detecting errors significantly. Recently, a lot of ART algorithms have been proposed. These algorithms can be divided into three categories : (1) distance-based ART methods such as *Fixed Size Candidate Set* (FSCS) [16]; (2) segmentation-based ART algorithms such as *Dynamic Partitioning* (DP) [22]; (3) Exclusion-based ART methods such as

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.

Restricted Random Testing (RRT) [23]. There are also some optimization algorithm using mirror partitioning [24], genetic algorithm [26], [42], etc. However, all the proposed methods are all used in sequential programs. ARS is inspired by the idea of adaptive random testing. To the best of our knowledge, this is the first time that the adaptiveness is introduced into concurrency bug detection.

VIII. Conclusion

In this work, we propose an adaptive random scheduling algorithm inspired by the idea of FSCS [16]. We empirically demonstrate the efficiency and effectiveness of the algorithm in concurrency bugs detection. The evaluation results also show that our algorithm performs more stable in terms of the effectiveness in concurrent bug detection than state-of-the-art methods such as simple randomized algorithm and maximal causality reduction algorithms.

References

[1] D. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 237–252.

[2] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 135–145.

[3] K. Sen, "Race directed random testing of concurrent programs," *ACM Sigplan Notices*, vol. 43, no. 6, pp. 11–21, 2008.

[4] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng, "Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 160–174.

[5] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 1. ACM, 2009, pp. 25–36.

[6] M. Eslamimehr, M. Lesani, and G. Edwards, "Efficient detection and validation of atomicity violations in concurrent programs," *Journal of Systems and Software*, vol. 137, pp. 618–635, 2018.

[7] G. Brat and W. Visser, "Combining static analysis and model checking for software analysis," in *ase*. IEEE, 2001, p. 262.

[8] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *Acm Sigplan Notices*, vol. 38, no. 10. ACM, 2003, pp. 167–178.

[9] Z. Wu, K. Lu, X. Wang, and X. Zhou, "Collaborative technique for concurrency bug detection," *International Journal of Parallel Programming*, vol. 43, no. 2, pp. 260–285, 2015.

[10] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *ACM Sigplan Notices*, vol. 40, no. 1. ACM, 2005, pp. 110–121.

[11] E. Noonan, E. Mercer, and N. Rungta, "Vector-clock based partial order reduction for jpf," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[12] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 165–174.

[13] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 446–455.

[14] K. Sen, "Effective random testing of concurrent programs," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 323–332.

[15] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 167–178.

[16] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Annual Asian Computing Science Conference*. Springer, 2004, pp. 320–329.

[17] S. Park, R. Vuduc, and M. J. Harrold, "A unified approach for localizing non-deadlock concurrency bugs," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012, pp. 51–60.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 329–339, 2008.

[19] "Ars," https://github.com/ars2019/ars, 2019.

[20] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 245–254.

[21] S. Lu, S. Park, and Y. Zhou, "Detecting concurrency bugs from the perspectives of synchronization intentions," *IEEE Transactions on Parallel & Distributed Systems*, no. 6, pp. 1060–1072, 2011.

[22] T. Y. Chen, R. Merkel, P. Wong, and G. Eddy, "Adaptive random testing through dynamic partitioning," in *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*. IEEE, 2004, pp. 79–86.

[23] K. P. Chan, T. Y. Chen, F.-C. Kuo, and D. Towey, "A revisit of adaptive random testing by restriction," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. IEEE, 2004, pp. 78–85.

[24] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, pp. 1001–1010, 2004.

[25] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 04, pp. 553–584, 2006.

[26] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Transactions on Reliability*, vol. 58, no. 4, pp. 619–633, 2009.

[27] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[28] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[29] Y. Benjamini and D. Yekutieli, "The control of the false discovery rate in multiple testing under dependency," *Annals of statistics*, pp. 1165–1188, 2001.

[30] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[31] "Soot," https://github.com/Sable/soot, 2018.

[32] "Asm," https://asm.ow2.io/, 2018.

[33] K. Lu, Z. Wu, X. Wang, C. Chen, and X. Zhou, "Racechecker: efficient identification of harmful data races," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 78–85.

[34] R. Wang, Z. Ding, N. Gui, and Y. Liu, "Detecting bugs of concurrent programs with program invariants," *IEEE Transactions on Reliability*, vol. 66, no. 2, pp. 425–439, 2017.

[35] T. Elmas, S. Qadeer, and S. Tasiran, "Precise race detection and efficient model checking using locksets," Microsoft Tech Report, Tech. Rep., 2006.

[36] A. Habib, "Finding concurrency bugs using graph-based anomaly detection in big code," in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*. ACM, 2016, pp. 55–56.

[37] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 221–234.

[38] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware java runtime," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 245–255.

[39] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[40] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: detecting atomicity violations via access interleaving invariants," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 37–48.

[41] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, 2008, pp. 267–280.

[42] T. Y. Chen, F.-C. Kuo, and H. Liu, "Distributing test cases more evenly in adaptive random testing," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2146–2162, 2008.

Authorized licensed use limited to: TIANJIN UNIVERSITY. Downloaded on September 23,2023 at 03:04:57 UTC from IEEE Xplore. Restrictions apply.