HARS: Heuristic-Enhanced Adaptive Randomized Scheduling for Concurrency Testing

Yanzhou Mu
College of Intelligence and Computing
Tianjin University
Tianjin, China
2019218009@tju.edu.cn

Zan Wang
College of Intelligence and Computing
Tianjin University
Tianjin, China
wangzan@tju.edu.cn

Shuang Liu*
College of Intelligence and Computing
Tianjin University
Tianjin, China
shuang.liu@tju.edu.cn

Jun Sun
School of Information Systems
Singapore Management University
Singapore City, Singapore
junsun@smu.edu.sg

Junjie Chen
College of Intelligence and Computing
Tianjin University
Tianjin, China
junjiechen@tju.edu.cn

Xiang Chen
School of Information Science and Technology
Nantong University
Nantong, China
xchencs@ntu.edu.cn

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00033

***Abstract*—Concurrency programs often induce buggy results due to the unexpected interaction among threads. The detection of these concurrency bugs costs a lot because they usually appear under a specific execution trace. How to virtually explore different thread schedules to detect concurrency bugs efficiently is an important research topic. Many techniques have been proposed, including lightweight techniques like adaptive randomized scheduling (ARS) and heavyweight techniques like maximal causality reduction (MCR). Compared to heavyweight techniques, ARS is efficient in exploring different schedulings and achieves state-of-the-art performance. However, it will lead to explore large numbers of redundant thread schedulings, which will reduce the efficiency. Moreover, it suffers from the "cold start" issue, when little information is available to guide the distance calculation at the beginning of the exploration. In this work, we propose a Heuristic-Enhanced Adaptive Randomized Scheduling (HARS) algorithm, which improves ARS to detect concurrency bugs guided with novel distance metrics and heuristics obtained from existing research findings. Compared with the adaptive randomized scheduling method, it can more effectively distinguish the traces that may contain concurrency bugs and avoid redundant schedules, thus exploring diverse thread schedules effectively. We conduct an evaluation on 45 concurrency Java programs. The evaluation results show that our algorithm performs more stably in terms of effectiveness and efficiency in detecting concurrency bugs. Notably, HARS detects hard-to-expose bugs more effectively, where the buggy traces are rare or the bug triggering conditions are tricky.**

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

I. Introduction

Benefiting from the rapid development of multi-core processor technology, concurrency programming has been widely promoted. While bringing exciting efficiency improvements, it also introduces concurrency bugs, which are notoriously hard to detect [1], [2]. Unlike general bugs, concurrency bugs only appear on specific buggy test inputs under certain thread schedules, and can only be reproduced when executing the target program multiple times with the same test input.

To detect concurrency bugs from the enormous thread schedules at an acceptable cost, many approaches have been proposed, ranging from exhaustive exploration approaches [3] to randomized scheduling approaches [4]–[6]. Musuvathi and Qadeer [7] proposed an iterative context bounding (ICB) approach to bounding context switch numbers in model checking to explore all possible states of the bounded context switches exhaustively. Partial Order Reduction (POR) [8] aims to reduce redundant thread schedules. However, it is limited by approximating the happens-before relations. To alleviate this problem, Huang [3] proposed maximal causality reduction (MCR) to reduce the search space of all possible interleavings. However, MCR suffers from scalability issues since it uses the maximum causal model to partition the state space into equivalent classes, which requires generating lots of constraints after executing the program to ensure read operation can get different write operation values as much as possible [9].

Randomized approaches [4]–[6], [10], [11], which detect concurrency programs randomly or with heuristic guidance, on the contrary, are shown to be efficient and widely adopted. Existing Approaches such as RACEFUZZER [6] and ATOMFUZZER [5] aim at detecting specific kinds of concurrency bugs, i.e., race conditions and atomicity violations through heuristics that guide the exploration of the corresponding scheduling. Hong et al. [10] focused on the synchronization blocks of concurrency programs. By dynamically establishing a thread model, they analyze the synchronization pairs that have not been covered and control the program schedule to execute them. Moreover, These approaches suffer from the priority inversion problem and may also explore a lot of redundant schedules that are not related to concurrency bugs.

Recently, Wang et al. propose an adaptive randomized scheduling approach [11]. The main idea of the adaptive randomized scheduling approach is to measure the diversity

* Shuang Liu is the corresponding author

of thread schedules via a distance metric and adaptively guide the detecting process. In this way, it avoids the exhaustive exploration of the whole search space and yet can explore a diverse set of thread schedules, which allows it to expose hard-to-detect concurrency bugs. However, this approach may encounter the "cold start" problem, which happens at the beginning of the detecting phase. When no or few memory-access patterns [12], [13] are observed. It tends to behave like a random search and focuses on the traces without bugs, thus greatly reducing the efficiency. Therefore, It will be helpful to improve this approach by adopting effective information to locate the traces that are likely to expose bugs in the early stage of detection.

In this work, we introduce Heuristic-Enhanced Adaptive Randomized Scheduling (HARS) algorithm. To effectively address the "cold start" problem, which exists in the execution of ARS, we introduce three heuristics i.e., Thread Switch Frequency (TSF), Write Operation Proportion (WOP), and Shared Variable Count (SVC), which are effective in detecting the traces that may expose bugs earlier. The heuristics are inspired by the memory-access pattern, which has been shown to be closely related to the significant features of concurrency bugs [12], [13]. Those heuristics are proposed to increase the probability of triggering new memory-access patterns and enable more effective distance metric guidance. Moreover, we also conduct pruning to avoid exploring redundant schedules. We conduct a comprehensive experiment with a benchmark of 45 multi-threaded Java programs to evaluate the effectiveness of HARS. To be specific, we make the following contributions:

- We propose the Heuristic-Enhanced Adaptive Randomized Scheduling (HARS) algorithm, which introduces three heuristics to eliminate the "cold start" problem and prunes redundant traces. It is publicly accessible [14].
- We build a comprehensive concurrency detection benchmark containing 45 multi-thread Java programs, with consideration of the complexity of concurrency bugs in reality. The benchmark contains various types of concurrency bugs, including deadlock, data race, etc.
- We evaluate with 45 programs, and the results show the effectiveness of HARS when compared to the simple randomized algorithm (SR) [15], thread-scheduling approach (TSA) [10], the maximal causality reduction (MCR) approach [3], and the adaptive randomzied scheduling (ARS) approach [11]. HARS can improve the distinguishable degree by 9.40% compared to ARS, which means that HARS solves the "cold start" problem well. HARS can further reduce the P-measure of 15.24% compared to ARS, and 72.58%, 79.64%, 58.02% compared with SR, MCR, and TSA, respectively (Note that MCR fails to detect bugs on 14 programs and TSA fails to detect bugs on 31 programs within the given time limit. Therefore, the average value is calculated on the programs that MCR and TSA can detect bugs). HARS is also significantly better than SR, TSA, and MCR in execution time and has only a small overhead compared to ARS.

II. PRELIMINARY

Memory-access patterns [12], [13] are proposed to capture the root cause of concurrency bugs. Park et al. [12] summarized three thread conflict interleaving and five unserializable thread interleaving memory-access patterns in Falcon. These patterns only focused on a single variable. Furthermore, they extended their research on multi-variable thread interleaving in Unicorn [13] and increased the original 8 memory-access patterns to 17 (shown in Table I). The second column introduces the content of the memory-access pattern. Each element is a tuple of (thread, statement, READ, WRITE). The thread represents the related thread id and the statement is the bytecode instruction that is most relevant to the shared variable and created by the source code in the program. READ and WRITE indicate a series of shared variables to be read or written. For example, the tuple $(t_a, s_i, x, \emptyset)$ means the shared variable x is read by thread t_a according to the bytecode instruction created by the ith statement in the source code.

TABLE I: The generic memory-access patterns [13]

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	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \{x\}, \emptyset)$

III. A MOTIVATING EXAMPLE

In this section, we introduce how the Heuristic-Enhanced Adaptive Randomized Scheduling (HARS) method works through a motivating example, which simulates the book management as shown in Fig. 1. There is one shared variable $store$ (it represents the currently available book amount), which is initialized with 100 (line 8). Besides, there are two threads (lines 9-13, 14-18), which simulate storing 100 books (lines 4-5) twice from different customers. Please note that the statement 4 and 5 involve read and write operation on the shared variable $store$, respectively. Therefore, if the executing order about these two statements of thread t_1 and t_2 is staggered, the program will get the wrong result.

To better introduce the detection process of HARS on the motivating example, We create the state space of the program, as shown in Fig. 2. The nodes represent the program state: program counters, variable values, and thread information. The edges are the transformation from one state to another, which

```
s1.public class Main{
s2.public static int store;
s3.public static void operation(int amount){
s4.      int newamount=store+amount;
s5.      store=newamount;
s6.   }
s7.      public static void main(String[] args) {
s8.            store=100;
s9.            new Thread(new Runnable() {
s10.               @Override
s11.               public void run(){
s12.                   operation(100);
s13.               }}).start()
s14.            new Thread(new Runnable() {
s15.               @Override
s16.               public void run(){
s17.               operation(100);
s18.               }}).start()
s19.            assert(store==300) : "wrong!";
s20.   }
s21.}
```

Fig. 1: A motivating concurrency program

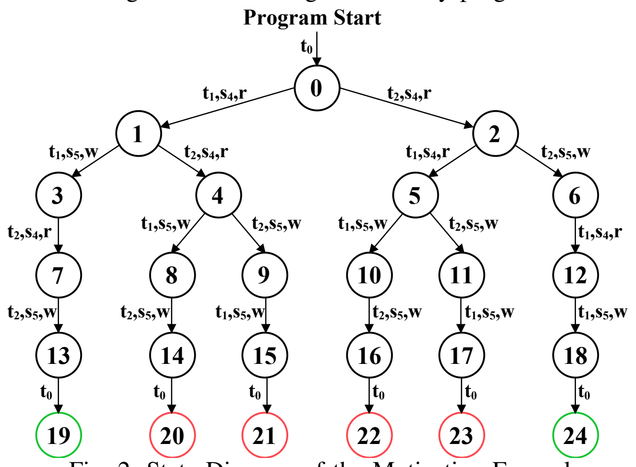


Fig. 2: State Diagram of the Motivating Example

is caused by executing bytecode instructions related to the read/write operation on the shared variable. t_0 represents the main thread (Since it does not involve any read/write operation on the shared variable *store*, it is omitted in the following introduction). For instance, one edge $\langle t_i, s_j, r\rangle$ represents a transition which is the result of statement s_j related to one read operation executed by thread t_i. In addition, We adopt $\langle tr_{s_j}^{t_i}\rangle$ to represent the node which is the result of the thread t_i executing the statement s_j when introducing the workflow.

HARS starts by randomly executing a completed trace. Suppose the first completed trace is $C_0 = \langle tr_{s_4}^{t_1}, tr_{s_5}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle$. So the set C_p which stores the passed traces is $\{\langle tr_{s_4}^{t_1}, tr_{s_5}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle\}$ now. Then we extract the memory-access pattern of trace C_0 based on Table I. The extracted patterns are those with ID 2 and 3 in Table I. Then the patterns of C_0 are added into the set of patterns PS_0:

$$PS_0 = \{[(t_1, s_5, \{\}, \{store\}), (t_2, s_4, \{store\}, \{\})], \\ [(t_1, s_5, \{\}, \{store\}), (t_2, s_5, \{\}, \{store\})]\} \quad (1)$$

Before the first iteration, we set the threshold N to 2 to control the length of the queue that stores the explored subtraces (A subtrace is an uncompleted trace of unfinished states). Then HARS explores two new subtraces, i.e., $\langle tr_{s_4}^{t_1}\rangle$ and $\langle tr_{s_4}^{t_2}\rangle$. Since there is no memory-access pattern in $\langle tr_{s_4}^{t_1}\rangle$ and $\langle tr_{s_4}^{t_2}\rangle$, both the memory-access pattern distance between $\langle tr_{s_4}^{t_1}\rangle$ and C_p, $\langle tr_{s_4}^{t_2}\rangle$ and C_p are 0 (Recall that the distance [11] is defined as the different patterns between PS_0 and the set PS' which belongs to new subtraces, i.e., $PS_0 \setminus PS'$). The three heuristics, i.e., TSF, WOP and SVC (They will be introduced in Section IV-A), provide the values of $\langle 0, 0\rangle$, $\langle 0, 0\rangle$ and $\langle 1, 1\rangle$ respectively. For instance, suppose the order of current queue after sorting according to the SVC heuristic is $\{\langle tr_{s_4}^{t_1}\rangle, \langle tr_{s_4}^{t_2}\rangle\}$. $\langle 1, 1\rangle$ means that the SVC heuristic value of the first element ($\langle tr_{s_4}^{t_1}\rangle$) and the second element ($\langle tr_{s_4}^{t_2}\rangle$) of the queue are both 1 since they both only involve one shared variable. The distinguishable degree (It is also introduced in Section IV-A) of the three heuristics are all 1. The length does not exceed the threshold N. Therefore, HARS randomly selects a subtrace to execute and suppose $\langle tr_{s_4}^{t_1}\rangle$ is selected. HARS executes $\langle tr_{s_4}^{t_1}\rangle$ for one step and obtains two new subtraces, i.e., $\langle tr_{s_4}^{t_1}, tr_{s_5}^{t_1}\rangle$ and $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}\rangle$. The queue is $\{\langle tr_{s_4}^{t_1}, tr_{s_5}^{t_1}\rangle, \langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}\rangle, \langle tr_{s_4}^{t_2}\rangle\}$ when the first iteration finishes.

In the second iteration, HARS first prunes the subtrace $\langle tr_{s_4}^{t_1}, tr_{s_5}^{t_1}\rangle$ at the beginning since it is the prefix of the trace $\langle tr_{s_4}^{t_1}, tr_{s_5}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle$, which is meaningless to explore since there are no new nodes under it. ARS and other randomized methods can not avoid exploring such meaningless subtraces due to the randomness. The 2 subtraces left in the queue, i.e., $\{\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}\rangle, \langle tr_{s_4}^{t_2}\rangle\}$, contain no pattern, and the distance between them and C_p are 0. Therefore, we adopt the heuristics to assist in selecting a subtrace. The three heuristics (i.e., TSF, WOP and SVC) provide the values of $\langle 1, 0\rangle$, $\langle 0, 0\rangle$ and $\langle 1, 1\rangle$ respectively. The distinguishable degree of the three heuristics are 2, 0 and 0, respectively. Therefore, TSF is adopted to select the next subtrace, i.e., $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}\rangle$, to expand from. Note that there is no sufficient information to guide ARS to select a subtrace from the queue according to the pattern distance metric and thus ARS may suffer from the "cold start" problem. HARS utilizes the three heuristics to guide the subtrace selection and largely eliminates this phenomenon. The current queue is $\{\langle tr_{s_4}^{t_2}\rangle\}$ after $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}\rangle$ dequeues. The queue length does not exceed the threshold N. HARS executes one step from $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}\rangle$ and obtains two new subtraces $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}\rangle$, $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle$, which makes the queue to be $\{\langle tr_{s_4}^{t_2}\rangle, \langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}\rangle, \langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle\}$ when the second iteration ends.

In the third iteration, we observe that the subtrace $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}\rangle$ matches the pattern $PS_1 = [(t_2, s_4, \{store\}, \{\}), (t_1, s_5, \{\}, \{store\})]$. The subtrace $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle$ matches the patterns PS_2=$[(t_1, s_4, \{store\}, \{\}), (t_2, s_5, \{\}, \{store\})]$. Subtrace $\langle tr_{s_4}^{t_2}\rangle$ does not match any pattern. The distances of the three subtraces with C_p are 2, 2 and 0, respectively. $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}\rangle$ and $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle$ have equal distances to C_p. To select the more "suspicious" subtrace, HARS further calculates the heuristic metrics. The three heuristics, i.e., TSF, WOP and SVC, provide the values of $\langle 2, 1\rangle$, $\langle 1, 1\rangle$ and $\langle 1, 1\rangle$ respectively. The distinguishable degree of the three heuristics are 1.5,1,1 respectively. HARS adopts TSF and selects $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}\rangle$ to expand for the next iteration. Although

Algorithm 1: The HARS Algorithm

```
Input  : $P$: Program to be tested; $N$: the limit length of $Q$;
Output: $T_{failed}$: the set of detected error traces; $T_{passed}$: the set of passed
        traces after executing search
$T_{passed} \leftarrow \emptyset$;
$trace \leftarrow$ execute P randomly;
if $trace$ is failed then
    return $\{trace\}, T_{passed}$;
$T_{passed} \leftarrow T_{passed} \cup \{trace\}$;
$T_{failed} \leftarrow \emptyset$;
while $T_{failed} = \emptyset$ do
    $T_{failed}, Passed \leftarrow$h-Search ($T_{passed}, P, N$) ;
    $T_{passed} \leftarrow T_{passed} \cup Passed$;
return $T_{failed}, T_{passed}$;
```

Algorithm 2: Heuristic Search (h-Search)

```
Input  : $T_{passed}$: the set of current passed traces; $P$: Program to be tested;
         $N$: the limit length of $Q$;
Output: $T_{failed}$: the set of detected error traces; $T_{passed}$: the set of passed
         traces after executing search
$SubT \leftarrow$ execute P for one step;
$T \leftarrow \emptyset$;
$Q \leftarrow$ empty queue;
enQueue($Q$, $SubT$);
while $Q \neq \emptyset$ do
    Prune $Q$;
    Random shuffle $Q$;
    Sort $Q$ by pattern distance desc;
    if $Q[0].desc$ != $Q[1].desc$ then
        $subtrace \leftarrow Q$.dequeue();
    else
        s $\leftarrow 1 + \sum_{i=1}^{Q.len-1} Q[0].desc == Q[i].desc$;
        $Q_s \leftarrow$ first s elements in $Q$;
        $Q_s^h \leftarrow$ sort $Q_s$ by heuristic h, $h \in \{TFS, WOP, SVC\}$;
        $D^h \leftarrow$ Degree ($Q_s^h[0]$, $Q_s^h[1]$), $h \in \{TFS, WOP, SVC\}$;
        $r \leftarrow max(D^{TSF}, D^{WOP}, D^{SVC})$;
        if $r < 0$ then
            $subtrace \leftarrow Q_s^h[0]$ for a random h ;
        else
            $subtrace \leftarrow Q_s^h[0]$ for h with the largest r;
        remove $Q_s^h[0]$ from $Q$;
    $Q \leftarrow$ the first $N$ elements of $Q$;
    $SubT \leftarrow$ execute $subtrace$ for one step;
    $T' \leftarrow$ all of the completed traces in $SubT$;
    $T \leftarrow T \cup T'$;
    if failed traces in $T$ then
        break;
    enQueue($Q$, $SubT \setminus T$);
$T_{passed} \leftarrow$ all of the passed traces in $T$;
$T_{failed} \leftarrow$ all of the failed traces in $T$;
return $T_{failed}, T_{passed}$;
```

there is now sufficient information for ARS to match the memory-access pattern, ARS fails to select a more suspicious subtrace through distance metric. However, the heuristics allow HARS to make a further decision. The current queue is $\{\langle tr_{s_4}^{t_2}\rangle, \langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_2}\rangle\}$. HARS executes one step and obtains one new trace $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}, tr_{s_5}^{t_2}\rangle$, which fails at the end of execution(marked with red in Fig. 2). Since HARS detects a failed trace, it will terminate and record the failed trace $\langle tr_{s_4}^{t_1}, tr_{s_4}^{t_2}, tr_{s_5}^{t_1}, tr_{s_5}^{t_2}\rangle$ into a new set T_{failed}. Finally, T_{failed} and C_p will be returned.

HARS explores two completed traces to detect the bug. It will be more complicated before the detection finishes in practice. Therefore, the new subtraces encountered during the exploration will be greatly different from the completed traces. The main idea of HARS aims to utilize the heuristics related to concurrency bugs to assist memory-access patterns to adequately explore the unknown state space without repetition. In addition, HARS also removes the meaningless fully explored subtraces by pruning, which improves efficiency.

IV. Our approach

In this section, we will introduce the details of the proposed heuristics, the distinguishable degree which is calculated based on the heuristics and the workflow of the Heuristic-Enhanced Adaptive Randomized Scheduling (HARS) algorithm.

A. Heuristics

We propose three heuristics to assist the memory-access pattern distance metric, thus enabling the detection more effectively. We observe that there are three main factors, i.e., thread switching, shared variable, and read/write operations, in a memory-access pattern. Therefore, we design heuristics from those aspects. The heuristics are also proposed with consideration of existing findings on recent research [16]–[18], as well as our observation through experiments. The heuristics will help HARS focus on the distinct traces without repeating random search locally, which is caused by the "cold start" problem. (e.g. In general, programs with more threads are more branching and the TSF heuristic allows HARS to capture different behaviors by "encouraging" more thread switching).

(1) Thread Switch Frequency (TSF). Frequent thread switching is likely to interrupt the original program logic and introduce more memory-access patterns. So it may introduce data races according to our experiments, which is reported to cause the most serious concurrency bugs and result in huge losses [16]. Therefore, we record the thread switch frequency of the visited subtraces as one of the heuristics.

(2) Write Operation Proportion (WOP). Previous research [18] shows that some memory-access patterns trigger more bugs, and the most bug-triggering patterns, e.g., "pattern 3" in Table I which is the concurrency write-write on a shared variable, contains more write operations. We use the write operation proportion, which is defined as the proportion of write operations in total read/write operations as one heuristic.

(3) Shared Variable Count (SVC). The number of shared variables involved in current trace is considered as the third heuristic. We can observe from Table I that more shared variables may match more memory-access patterns. Based on the findings of our previous study [18], memory-access pattern coverage is positively correlated with the bug-revealing capability. Moreover, the number of shared variables is also adopted by other existing works as important features in concurrency bug prediction [17].

Note that we do not have any preference for the heuristics. We would rather record the corresponding meta-information, i.e., the thread switching frequency, the number of read/write operations, and the number of shared variables, and calculate the metrics of all three heuristics. We then select the most effective heuristic in each iteration. The effectiveness of a heuristic is measured by two rules: (1) whether the heuristic can distinguish the visited subtraces; and (2) to what extent the heuristic can distinguish the subtraces, i.e., the distinguishable degree. If a heuristic provides a ranking where the values of the subtraces ranked 1st and 2nd are different, i.e., it meets the condition (1). Then we consider the heuristic that can distinguish the subtraces in the queue. If more than one heuristic can provide rankings that can distinguish subtraces, we further check the **distinguishable degree**, which is measured by $(1 + V_1^h)/(1 + V_2^h)$, where V_1^h and V_2^h are the value of the 1st-ranking subtrace and the value of the 2nd-ranking subtrace by using the heuristic h, respectively(Adding 1 to both numerator and denominator is to avoid the divide-by-zero problem). If more than one heuristic provides the same distinguishable degree, we randomly pick one.

B. Heuristic-Enhanced ARS

The HARS algorithm is shown in Algorithm 1 (it introduces the main method of HARS) and 2 (it specifically introduces how HARS explores state space and detect bugs). The task of HARS is to detect the buggy schedules in the state space of the target program. Algorithm 1 takes the target program P and the threshold N as input parameters. It starts by randomly executing P to get one completed trace (line 2). HARS will finish and return the trace if it detects a failed trace (lines 3-4). Otherwise, the passed trace $trace$ will be added into the passed trace set T_{passed} (line 5). Algorithm 1 starts the searching step by calling method *h-search* as shown in Algorithm 2 until one or more failed traces are found (lines 7-9). The *h-Seach* method conducts a search, which collects one or more traces and will return T_{passed} and T_{failed} (line 10) finally.

Algorithm 2 takes T_{passed}, which consists of the completed traces, the target program P and the threshold N as input parameters. Firstly, *h-Search* executes P for one step (line 1) to collect all available initial transitions of P and adds them into the subtrace set $SubT$. Then all the elements of $SubT$ are recorded into the new priority queue Q (line 4). From line 5 to 28, HARS first prunes the fully explored subtraces (those subtraces that have been fully explored) in Q in order to reduce the search state space (line 6). A subtrace is fully explored when all of its descendant subtraces are explored. After reaching the final state, HARS updates the status of whether its ancestor subtraces are fully explored. According to the experimental results, the prune operation will contribute a lot in those programs with short thread scheduling and simple branches (such as *SimpleTest*), while it can be invalid in those programs that are easy to expose bugs or have complex depth and breadth (such as *Log4j*). In both two types of programs, prune operation does not occupy a huge time overhead. However, maintaining prune operation can effectively reduce the search space on more complex concurrency programs and avoid exploring redundant traces. Before formal sorting, HARS first randomly shuffles Q to introduce randomness on subtraces with the same distances (line 7). This step can introduce more diversity to Q and avoid searching into the local area since most of the subtraces are consistent with the distance metric at the beginning. Then it sorts Q based on memory-access pattern distance metric in descending order(line 8).

The main differences between HARS and ARS are from line 9 to 21. More specifically, HARS also adopts memory-access pattern distance metric to select more suspicious subtraces like ARS. However, HARS adopts three finer-grained heuristics to further select the subtraces for next execution (lines 11-21), which is enhanced based on ARS. If the ranking based on memory-access pattern distance can distinguish the first-ranked subtrace and the second-ranked subtrace in Q, HARS will select the first element of Q and assign it to $subtrace$ (lines 9-10). Otherwise, HARS collects all the subtraces with the same distance metric as the first-ranked subtraces, and assigns those subtraces (suppose there are s such subtraces) to a temporary queue Q_s (lines 12-13). Then we sort Q_s with the three heuristics we proposed in Section IV-A (line 14). Then we will calculate the distinguishable degree (defined in Section IV-A) on the sorting results of the three heuristics. The input of method $Degree$ is the heuristic value of the 1st-ranked and 2nd-ranked subtrace based on one heuristic, and the output is the distinguishable degree of the corresponding ranking. The distinguishable degree of these three heuristics will be recorded into D^h ($h \in \{TFS, WOP, SVC\}$) (line 15). Method max selects the heuristic, which has a maximum distinguishable degree (line 16). If the method max is unable to find one max value from $D^{TSF}, D^{WOP}, D^{SVC}$ (i.e., If there are more than one maximum distinguishable degree among the three heuristic ranking results), it will return a negative value, and HARS will randomly choose one maximal ranking result from Q_s^h and assign $Q_s^h[0]$ to $subtrace$ (lines 17-18). Otherwise, HARS will select the heuristic h with the maximal distinguishable degree and assign $Q_s^h[0]$ to $subtrace$ (lines 19-20). Then we remove $Q_s^h[0]$ from Q since it has been selected for the next execution (line 21). HARS will only keep the first N subtraces with the largest distances (line 22). Then HARS continues exploring $subtrace$ one step forward (line 23) by running P following the path recorded in $subtrace$. The exploration stops when the final state of $subtrace$ is reached. The result of exploration is a set of subtraces $SubT$. Then all completed subtraces in $SubT$, i.e., traces that reach the final state, are added to T (lines 24-25). If there are failed traces in T, HARS will terminate and exit the while loop (line $26-27$). Otherwise, the others which have not reach the final state in $SubT$ are added to Q (line 28) and continue exploring. The while loop continues until the Q becomes empty or HARS detects bugs. The set T contains all passed and failed traces explored when HARS finishes. After HARS finishes its search, it will record all the passed traces found into T_{passed} (line 29), all the failed traces found into T_{failed} (line 30), and return them to the main function *HARS* (line 31).

V. Empirical Evaluations

A. Experiment Setup

We evaluate HARS on 45 concurrency programs collected from existing work [3] [19] [20] [16]. There are 40 pro-

TABLE II: Results of P-measure and execution time

Program	LOC	#Thrd	Type	#Avg P-measure					#Avg Time (s)				
				ARS	HARS	SR	MCR	TSA	ARS	HARS	SR	MCR	TSA
Account	373	10	atomicity	1.550	**1.390**	1.810	9.200	N.A.	9.360	6.688	**2.063**	1660.598	N.A.
Airline	136	20	order	27.050	27.240	20.780	**8.000**	N.A.	**6.263**	7.460	6.499	187.526	N.A.
Alarmclock	351	8	race	2.840	**2.830**	4.220	N.A.	N.A.	2.347	3.536	**1.574**	N.A.	N.A.
Allocationvector	348	2	atomicity	1.130	**1.000**	1.010	4.000	1.900	0.192	**0.096**	0.231	1.377	56.677
Atmoerror	67	2	race	1.010	1.030	1.040	**1.000**	N.A.	0.160	0.280	**0.106**	0.150	N.A.
Bakery	111	2	atomicity	25.070	**13.780**	54.780	N.A.	85.600	1.834	**1.439**	8.056	N.A.	2.548
Boundedbuffer	136	3	deadlock	1.000	1.000	1.000	1.000	N.A.	0.097	**0.080**	0.203	0.359	N.A.
Checkfield	44	2	atomicity	5.270	4.170	4.370	**3.000**	N.A.	0.169	**0.144**	0.490	0.688	N.A.
Consisitency	35	3	atomicity	2.880	2.810	6.350	**2.000**	48.900	0.252	**0.140**	0.831	0.487	1.329
Critical	70	2	race	2.480	2.230	3.320	**2.000**	N.A.	0.220	**0.112**	0.435	0.262	N.A.
Cyclicdemo	54	4	atomicity	1.780	1.720	2.190	8.000	**1.000**	0.179	**0.141**	0.211	5.530	2.352
Datarace	118	2	atomicity	1.970	1.560	2.200	7.000	**1.100**	**0.144**	0.407	0.448	16.235	0.845
Dekker	100	2	atomicity	40.400	**36.720**	105.560	N.A.	64.100	**0.663**	0.947	11.404	N.A.	2.067
Even	63	4	order	1.010	**1.000**	1.020	2.000	N.A.	0.203	**0.084**	0.108	1.857	N.A.
Java.library.arraylist	1971	2	race	1.280	**1.230**	1.290	2.000	N.A.	0.120	**0.092**	0.209	0.891	N.A.
Java.library.hashset	3989	2	order	20.610	**11.440**	39.430	149.400	N.A.	**5.097**	13.523	9.715	44.326	N.A.
Java.library.linkedlist	2556	2	race	**12.070**	12.580	255.170	458.100	N.A.	**2.640**	5.823	56.095	154.185	N.A.
Java.library.treeset	4274	2	order	2.790	2.760	2.690	**1.000**	N.A.	0.247	0.271	**0.184**	0.216	N.A.
Java.library.vector	4675	2	order	2.550	**2.310**	2.440	N.A.	N.A.	0.246	**0.222**	0.582	N.A.	N.A.
Lamport	150	2	atomicity	12.100	**11.830**	36.020	N.A.	88.500	0.371	**0.284**	3.534	N.A.	2.593
Lang	2444	2	atomicity	2.680	2.730	4.260	8.000	**1.000**	0.183	**0.145**	0.562	3.264	0.433
Linkedlist	299	2	atomicity	2.850	**1.960**	2.130	6.900	5.800	0.525	0.412	**0.314**	11.277	2.939
Log4j	18799	4	order	4.760	**3.640**	7.190	38.000	N.A.	19.554	15.118	**4.638**	24.786	N.A.
Mergesort	399	2	race	1.550	**1.530**	1.770	2.000	N.A.	1.851	2.016	**1.396**	426.213	N.A.
Mix0	50	2	race	1.750	1.610	1.830	N.A.	**1.000**	**0.113**	0.115	0.292	N.A.	0.430
Nestedmonitors	140	3	deadlock	1.000	1.000	1.000	1.000	N.A.	0.169	**0.120**	0.252	0.134	N.A.
Peterson	77	2	atomicity	13.390	12.130	32.800	**8.000**	N.A.	0.420	**0.331**	3.885	4.753	N.A.
Pingpong	115	4	atomicity	2.540	2.310	N.A.	2.000	**1.000**	0.663	0.616	N.A.	0.642	**0.476**
Pipeline	119	7	race	2.580	**2.490**	25.540	N.A.	N.A.	**0.381**	0.700	4.060	N.A.	N.A.
Producerconsumer	194	7	atomicity	2.540	**2.150**	3.430	N.A.	N.A.	16.151	15.612	**2.584**	N.A.	N.A.
Rax	76	2	deadlock	**1.060**	1.090	1.140	2.000	N.A.	0.102	**0.085**	0.212	0.653	N.A.
Readerswriters	598	3	race	2.270	**1.980**	7.210	10.000	N.A.	**0.312**	0.716	1.288	66.023	N.A.
Reorder	92	4	atomicity	11.260	9.070	74.480	**4.000**	N.A.	0.716	**0.698**	9.626	0.738	N.A.
Reorder2	71	6	atomicity	2.880	**2.780**	4.360	3.000	N.A.	0.248	**0.201**	0.461	1.088	N.A.
Sharedobject	48	3	atomicity	2.920	2.900	3.280	N.A.	**2.300**	**0.146**	0.173	0.422	N.A.	0.847
Simpleexample	29	3	atomicity	3.820	**3.630**	8.520	N.A.	63.200	0.230	**0.121**	0.825	N.A.	3.457
Simpletest	47	2	atomicity	6.090	**4.260**	6.440	4.900	N.A.	0.222	**0.202**	0.608	0.636	N.A.
Store	46	2	race	1.570	1.510	**1.390**	2.000	N.A.	0.194	**0.111**	0.180	0.924	N.A.
Stringbuffer	416	3	atomicity	3.950	2.950	3.470	**2.000**	N.A.	0.294	**0.145**	0.421	0.406	N.A.
Test	34	3	race	3.380	**2.900**	5.710	N.A.	N.A.	0.139	**0.102**	0.516	N.A.	N.A.
Test2	34	3	order	1.490	**1.440**	1.940	N.A.	N.A.	0.114	**0.090**	0.180	N.A.	N.A.
Testarray	46	2	atomicity	4.460	**2.890**	8.970	N.A.	N.A.	0.279	**0.232**	0.909	N.A.	N.A.
Twostage	140	2	race	10.680	9.340	36.640	**2.000**	N.A.	**0.390**	0.437	4.134	0.751	N.A.
Wronglock	69	2	race	3.750	3.100	4.400	**2.000**	N.A.	0.175	**0.130**	0.410	0.289	N.A.
Wronglock2	36	5	race	1.340	1.210	1.360	N.A.	**1.000**	**0.113**	0.130	0.160	N.A.	0.430
Average	-	-	-	5.853	**4.961**	18.090	24.371	11.819	**1.655**	1.790	3.212	84.428	2.497

The first 4 columns show the name, the number of code lines, the number of threads, and the type of concurrency bugs in the corresponding program; Columns 5-9 show the value of P-measure of each method; Columns 10-14 show the average execution time of each method.

grams adopted from real-world concurrency scenarios, and each program contains concurrency bugs about data race, order violation, atomicity violation, or deadlock. The other 5 programs are from the package `Java.util.Collection` . The programs do not handle concurrency aspects properly and may lead to program crashes or memory corruption [5], [21], [22]. We generate test cases to expose the potential bugs in them. The detailed information of the 45 programs is shown in the first 4 columns of Table II. We compare our approach with the following four baseline approaches: (1) the simple randomized algorithm (SR) [15], which is a simple, yet effective and classical randomized algorithm for concurrency program testing; Note that we adopt the SR method, which is integrated into the JPF [23] framework in the default way. It starts scheduling from the initial state of the program and selects the subsequent feasible scheduling states based on the random strategy until the termination state is reached, which is similar to performing a depth-first search. If the program contains deadlocks, It will report them directly. (2) the heuristic search approach based on synchronization-pair coverage measure (TSA) [10], which is the state-of-the-art randomized approach; (3) the maximal causality reduction (MCR) approach [3], which is the state-of-the-art stateless model checker (exhaustive testing) for detecting concurrency bugs; (4) the adaptive randomized scheduling (ARS) approach, which customizes adaptive randomized testing for concurrency program testing. Similar to ARS, the threshold N of HARS also controls the search length. Theoretically, larger N results in a higher probability of finding bugs but also decreases the search efficiency. ARS sets N to 5 as the best parameter recommendation. We also empirically analyze with different N values (from 1 to 6) according to the executing time and P-measure. The detailed results are shown on our project website [14]. To make HARS achieve a balance between effectiveness and efficiency, we set the threshold N to 4 based on the results. For all the results reported in this section, we run each benchmark program multiple times and report the average value to alleviate the effect of randomness. For the experiments related to ARS, HARS, and SR approaches, we run each program 100 times and report the average results. Since the constraint solving adopted in MCR and the static analysis techniques adopted in TSA are both time-consuming, we run ran each program 10 times and reported the average results for MCR and TSA. In each run, the program stops when the first bug is found, the number of executions and the execution time will be recorded. The experiments are conducted on a computer with a 4-core, 2.80GHz Intel i5 CPU and 8GB physical memory. The operating system is Windows 10, and the Java version is JDK 1.8. The heap space is 2GB.

B. Research Questions

RQ1: What does HARS solve the "cold start" problem existing in ARS? Previous research [11] has been proved that ARS can effectively detect concurrency bugs, but it also suffers the "cold start" problem. HARS adopts three heuristics related to the feature of the memory-access pattern to address it. Therefore, we aim to investigate whether HARS can effectively solve the "cold start" problem.

RQ2: How effective is HARS in detecting concurrency bugs compared with existing methods? Whether HARS can effectively detect the concurrency bugs contained in the target program is our key concern. Therefore, we measured whether it can always detect concurrency bugs in an acceptable number of iterations to evaluate its performance. We also compare HARS with state-of-the-art methods.

RQ3: How efficient is HARS in detecting concurrency bugs compared with existing methods? In addition to the effectiveness of detecting concurrency bugs, the efficiency of HARS is also our main concern. In the current research problem, we hope to explore whether HARS can detect concurrency bugs in a shorter time than existing methods.

RQ4: Does HARS achieve high MAP coverage? We aim to evaluate the bug-revealing effectiveness of HARS, i.e., can HARS find as many bugs as possible in the limited resources. The previous research [18] has shown that MAP coverage is positively correlated with the bug-revealing effectiveness of test executions. Since there is only one bug in most of the programs, and manually injecting bugs may introduce construction threats. Therefore, we evaluate how fast HARS can achieve high MAP coverage, which indirectly indicates the bug-revealing effectiveness of HARS.

C. Result Analysis

RQ1: What does HARS solve the "cold start" problem existing in ARS?

Design. ARS mainly focuses on adopting the pattern distance metric to guide detection while HARS tries to adopt heuristics to further enhance it. To measure the "cold start" problem and the effectiveness of HARS in solving the problem, we use the distinguishable degree which is introduced in section IV-A to calculate the contribution of the pattern distance metric and heuristics. Note that the more suspicious the selected subtrace is, the greater the distinguishable degree is. When it equals 1, it indicates that the sorting based on pattern distance metric or heuristics can not effectively select valuable subtraces. When the length of the queue is 1, the distinguishable degree is recorded as 0. We recorded the first 50 sorting processes which represent the early stage of the detection and repeated them 50 times on all benchmark programs.

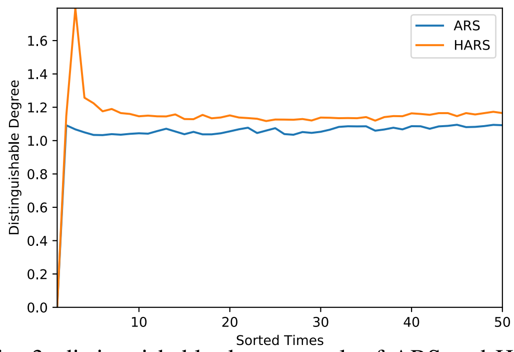


Fig. 3: distinguishable degree result of ARS and HARS

Result. Fig. 3 shows the changes of the distinguishable degree of ARS and HARS with the depth of exploration (the increase of sorting times). We utilize the average result of the 50 iterations on each program, and then compute the average result of all programs. Note that excluding two deadlock programs (i.e. *Boundedbuffer* and *Nestedmonitors*) and four programs (i.e. *Allocationvector*, *Mix0*, *Atmoerror*, and *Test2*) whose bugs can be detected in the first 50 sorting processes, so we collect the results on the left 39 programs. We find that the distinguishable degree of pattern distance metric gradually increases. However, it is maintained in the range of 1.0 - 1.1 during the first 50 sorting processes and the total average value is 1.042, which shows that the "cold start" problem exists at the beginning of the ARS detection process for different programs. Meanwhile, we observed that the distinguishable degree curve of HARS increased first and then decreased and converged. The range of HARS is 1.2-1.6 and the total average value is 1.140, which is better than ARS. Since the program information accumulated at the very beginning is very uneven, a heuristic will dominate sorting in the first few iterations, and its distinguishable degree will be much higher than others. As further exploration is done, the pattern distance metric will also be effective and the difference between heuristics will gradually diminish, so the distinguishable degree will decline and converge. However, the heuristics can greatly help HARS maintain an advantage over ARS.

In some programs with rare buggy traces such as *Dekker*, ARS can efficiently locate branches far from the correct traces according to the memory-access pattern related to the bug. Therefore, the pattern distance metric is effective in such cases. Although the "cold start" phenomenon is not obvious in *Dekker*, the heuristic (i.e. the TSF and WOP heuristic) can help HARS focus on branches that contain valuable memory-access patterns faster, which also shows the benefits of heuristics. However, in programs with many branches such as *Account*, ARS will go back to the ancestral subtraces contained in the queue and explore old search spaces induced by the "cold start" problem. This phenomenon makes ARS focus on searching the shallow area of the state space randomly. In contrast, HARS can avoid this according to heuristics. Program *Account* starts 8 threads, TSF heuristic can effectively distinguish the difference between different subtraces and continue exploration without returning to those ancestral subtraces. Therefore, HARS can avoid the "cold start" problem and detect the buggy trace faster than ARS.

RQ2: How effective is HARS in detecting concurrency bugs compared with existing methods?

Design. We evaluate HARS on 45 benchmark programs and report results in Table II. In our evaluation, we use P-measure (the amount of traces used to detect the first bug.) to measure the effectiveness. The calculation method of P-measure is shown in Definition 2. In Definition 2, $\|T_{passed}\|$ represents the amount of all passed traces detected. The design insight of P-measure is to evaluate whether the method can stably detect bugs in different types of concurrency programs.

$$P = \|T_{passed}\| + 1 \qquad (2)$$

To evaluate the difference between HARS and the existing methods from the statistic view, we also adopt statistical test methods to analyze the results in terms of P-measure. We first conduct Wilcoxon signed-rank test [24] which rectifies the p-value by adopting Bonferroni correction [25]. Based on the above results, we further analyze the p-values by utilizing Win/Tie/Loss analysis as recommended by other previous studies [26], where "Win" means that the results of HARS are significantly better than ARS (resp. SR, TSA, and MCR) at a confidence level of 95%, "Tie" means that there is no statistically significant difference, and "Lose" means ARS (resp. SR, TSA, and MCR) are significantly better than HARS at a confidence level of 95%. In addition, Cliff's δ [27] is also adopted to measure the results between HARS and other methods, which is a non-parametric effect size measurement. δ measures how often the values in one method are larger than that in the second method. The value interval of δ is [-1, 1], where -1 (resp. 1) denotes that all values in one method are smaller (resp. larger) than those of the other method, and 0 denotes that the values in the two methods are completely overlapping. To give a clear comparison of the effect size results, we count the number of projects for each magnitude (N/S/M/L) following Romano and Kromrey's suggestion [28].

Finally, we conduct the Scott-Knott ESD test [29] to rank all methods in terms of P-measure. The Scott-Knott ESD test is widely adopted to analyze whether some methods outperform others and can generate a global ranking for these methods. It guarantees that there is no significant performance difference between methods in the same group, and there is a significant performance difference between methods in different groups.

Result. The P-measure results are shown in column 5 to column 9 in Table II. The best performance results are highlighted in bold. ARS and HARS manage to detect the bugs in all programs in a few iterations, whereas SR fails on the program `Pingpong`, TSA only can detect bugs successfully on 14 programs, and MCR fails to find the bugs on 14 programs in the given time limit (1 hour each). We can observe that HARS performs the best in 22 cases, which outperforms all the other approaches. There are only 2 cases that ARS performs the best, 1 case that SR performs the best, 11 cases that MCR performs the best, and 7 cases that TSA performs the best. These results suggest that HARS are more stable in exposing bugs than other baseline methods. By carefully analyzing the programs that HARS outperforms the other methods, we can find that those program traces, e.g., `Readerswriters`, produce a large number of memory-access patterns in the subtraces. This result suggests the effectiveness of the memory-access pattern distance metric. HARS performs significantly better than ARS on `Simpletest`. Since this program does not contain many patterns, the bug is closely related to the improper interleaving of write operations in the two threads. The TSF heuristic helps to effectively select the most suspicious subtrace to detect the bug. Other examples include `Bakery` and `Dekker`, which adopt the heuristic TSF and WOP to guide HARS to select subtraces more effectively. HARS also shows effectiveness in avoiding duplicate traces. HARS can prune 45.83% of the traces in the `Simpletest` program, which significantly improves the detection effectiveness.

The results of the Wilcoxon signed-rank test and Cliff's δ are shown in Table III. From column 2 to 5, we can observe that there are 9 programs where HARS has significantly better results, as indicated by the symbol Win, than ARS, and the number is 27, 30, and 36 for SR, MCR, and TSA. Note that the bugs in `Boundedbuffer` and `Nestedmonitors` are deadlocks. ARS, HARS, SR, and MCR can detect the deadlock through one schedule. Therefore, there is no difference between the 4 methods in terms of P-measure. TSA can not detect deadlock because it takes too long to analyze and execute the scheduling. Besides, we also find that MCR performs better on those programs, where the bugs only expose in specific evaluations of shared variables, such as `Reorder`, `TwoStage`. However, MCR can not achieve satisfactory performance on the 5 programs of the Java.util.Collection package. For example, When testing `Java.library.ArrayList`, it is time-consuming to perform code instruments and solve the constraints of the parent classes (e.g., Abstractcollection) and the implemented interfaces (e.g., List), which are relevant to the bugs. This increases the detection cost of MCR. TSA shows good performance in detecting bugs that are caused by incorrectly using synchronous protection mechanisms. For example, in `Cyclicdemo`, due to the wrong use of the "Synchronized" keyword, the shared variables are still not correctly protected and cause an exception during execution. Since the core idea of TSA is to give high priority to covering the schedules related to rich synchronized code blocks, it can detect this kind of bug more efficiently. HARS, on the other hand, shows significantly better performance on 20+ programs than all the other baseline methods. HARS also shows good performance for bugs that are hard to expose and only related to rare schedules, for instance, `Dekker` and `bakery`, which have less than 2% of buggy traces in the search space.

The effect size results about Cliff's δ are shown in Table III.

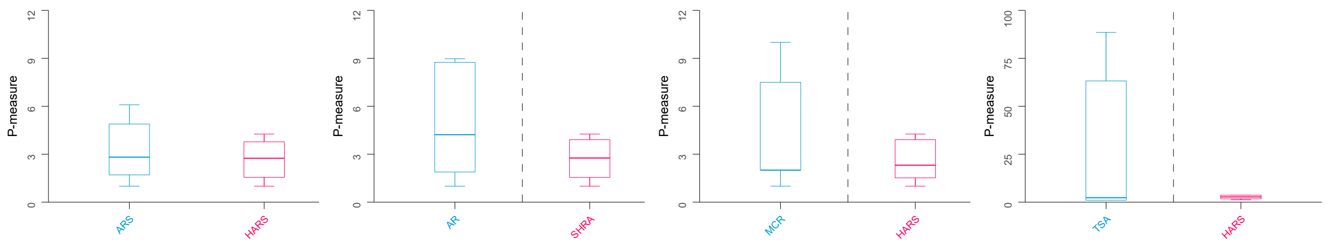
TABLE III: Results of statistical test analysis

Program	ARS	SR	MCR	TSA
Account	0.112 (Tie)/0.07 (N)	0.005 (Win)/-0.17 (S)	0.006 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Airline	0.871 (Tie)/-0.00 (N)	0.120 (Tie)/0.05 (N)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Alarmclock	0.954 (Tie)/-0.00 (N)	0.005 (Win)/-0.17 (S)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Allocationvector	1.000 (Tie)/0.01 (N)	1.000 (Tie)/-0.01 (N)	0.002 (Win)/-1.00 (L)	0.031 (Win)/-0.60 (L)
Atmoerror	0.424 (Tie)/-0.02 (N)	0.777 (Tie)/-0.01 (N)	0.149 (Tie)/0.30 (S)	0.000 (Win)/-1.0 (L)
Bakery	0.000 (Win)/0.34 (M)	0.000 (Win)/-0.62 (L)	0.000 (Win)/-1.0 (L)	0.014 (Win)/-0.60 (L)
Boundedbuffer	N.A./0.00 (N)	N.A./0.00 (N)	N.A./0.00 (N)	0.000 (Win)/-1.0 (L)
Checkfield	0.131 (Tie)/0.09 (N)	0.992 (Tie)/-0.07 (N)	0.032 (Loss)/0.70 (L)	0.000 (Win)/-1.0 (L)
Consisitency	0.975 (Tie)/-0.05 (N)	0.000 (Win)/-0.36 (M)	0.006 (Loss)/1.00 (L)	0.053 (Tie)/-0.46 (M)
Critical	0.214 (Tie)/0.14 (N)	0.002 (Win)/-0.20 (S)	0.126 (Tie)/0.20 (S)	0.000 (Win)/-1.0 (L)
Cyclicdemo	0.856 (Tie)/-0.03 (N)	0.055 (Tie)/-0.15 (S)	0.006 (Win)/-1.00 (L)	0.006 (Loss)/1.00 (L)
Datarace	0.005 (Win)/0.22 (S)	0.001 (Win)/-0.24 (S)	0.006 (Win)/-1.00 (L)	0.012 (Loss)/0.80 (L)
Dekker	0.669 (Tie)/0.04 (N)	0.000 (Win)/-0.51 (L)	0.000 (Win)/-1.0 (L)	0.262 (Tie)/-0.24 (S)
Even	1.000 (Tie)/0.01 (N)	0.346 (Tie)/-0.02 (N)	0.002 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Java.library.arraylist	0.450 (Tie)/0.03 (N)	0.403 (Tie)/0.00 (N)	0.006 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Java.library.hashset	0.000 (Win)/0.49 (L)	0.000 (Win)/-0.63 (L)	0.006 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Java.library.linkedlist	0.512 (Tie)/-0.09 (N)	0.000 (Win)/-0.92 (L)	0.006 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Java.library.treeset	0.826 (Tie)/0.02 (N)	0.619 (Tie)/-0.07 (N)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Java.library.vector	0.819 (Tie)/-0.01 (N)	0.557 (Tie)/-0.06 (N)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Lamport	0.787 (Tie)/-0.03 (N)	0.000 (Win)/-0.47 (M)	0.000 (Win)/-1.0 (L)	0.014 (Win)/-0.76 (L)
Lang	0.780 (Tie)/-0.01 (N)	0.000 (Win)/-0.21 (S)	0.006 (Win)/-1.00 (L)	0.006 (Loss)/1.00 (L)
Linkedlist	0.010 (Win)/0.19 (S)	0.176 (Tie)/-0.17 (S)	0.006 (Win)/-1.00 (L)	0.013 (Win)/-0.76 (L)
Log4j	0.009 (Win)/0.22 (S)	0.000 (Win)/-0.31 (S)	0.006 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Mergesort	1.000 (Tie)/0.01 (N)	0.129 (Tie)/-0.05 (N)	0.009 (Win)/-0.90 (L)	0.000 (Win)/-1.0 (L)
Mix0	0.127 (Tie)/0.12 (N)	0.155 (Tie)/-0.10 (N)	0.000 (Win)/-1.0 (L)	0.006 (Loss)/1.00 (L)
Nestedmonitors	N.A./0.00 (N)	N.A./0.00 (N)	N.A./0.00 (N)	0.000 (Win)/-1.0 (L)
Peterson	0.319 (Tie)/0.13 (N)	0.000 (Win)/-0.41 (M)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Pingpong	0.067 (Tie)/0.13 (N)	0.000 (Win)/-1.00 (L)	0.006 (Loss)/1.00 (L)	0.006 (Loss)/1.00 (L)
Pipeline	0.480 (Tie)/0.08 (N)	0.000 (Win)/-0.91 (L)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Producerconsumer	0.243 (Tie)/0.00 (N)	0.000 (Win)/-0.26 (S)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Rax	0.458 (Tie)/-0.03 (N)	0.322 (Tie)/-0.03 (N)	0.005 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Readerswriters	0.076 (Tie)/0.13 (N)	0.000 (Win)/-0.70 (L)	0.006 (Win)/-1.00 (L)	0.000 (Win)/-1.0 (L)
Reorder	0.018 (Win)/0.16 (S)	0.000 (Win)/-0.77 (L)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Reorder2	0.702 (Tie)/0.07 (N)	0.002 (Win)/-0.27 (S)	0.286 (Tie)/-0.16 (S)	0.000 (Win)/-1.0 (L)
Sharedobject	0.626 (Tie)/0.05 (N)	0.090 (Tie)/-0.17 (S)	0.000 (Win)/-1.0 (L)	0.125 (Tie)/0.54 (L)
Simpleexample	0.562 (Tie)/0.00 (N)	0.000 (Win)/-0.40 (M)	0.000 (Win)/-1.0 (L)	0.014 (Win)/-0.60 (L)
Simpletest	0.012 (Win)/0.15 (N)	0.000 (Win)/-0.27 (S)	0.032 (Win)/-0.72 (L)	0.000 (Win)/-1.0 (L)
Store	0.540 (Tie)/0.04 (N)	0.967 (Tie)/-0.08 (N)	0.028 (Win)/-0.60 (L)	0.000 (Win)/-1.0 (L)
Stringbuffer	0.012 (Win)/0.17 (S)	0.151 (Tie)/-0.07 (N)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Test	0.072 (Tie)/0.14 (N)	0.000 (Win)/-0.34 (M)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Test2	0.487 (Tie)/0.06 (N)	0.001 (Win)/-0.16 (S)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Testarray	0.005 (Win)/0.18 (S)	0.000 (Win)/-0.50 (L)	0.000 (Win)/-1.0 (L)	0.000 (Win)/-1.0 (L)
Twostage	0.252 (Tie)/0.07 (N)	0.000 (Win)/-0.59 (L)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Wronglock	0.910 (Tie)/-0.03 (N)	0.008 (Win)/-0.12 (N)	0.006 (Loss)/1.00 (L)	0.000 (Win)/-1.0 (L)
Wronglock2	0.096 (Tie)/0.03 (N)	0.039 (Win)/-0.10 (N)	0.000 (Win)/-1.0 (L)	0.014 (Loss)/0.80 (L)

Column 1 is the name of the program, Column 2-5 show the statistical test results of HARS with ARS, SR, MCR, and TSA. The format of each element in columns 2-5 is "*p*-value of Wilcoxon test (W/T/L)/value of Cliff's δ (N/S/M/L)".

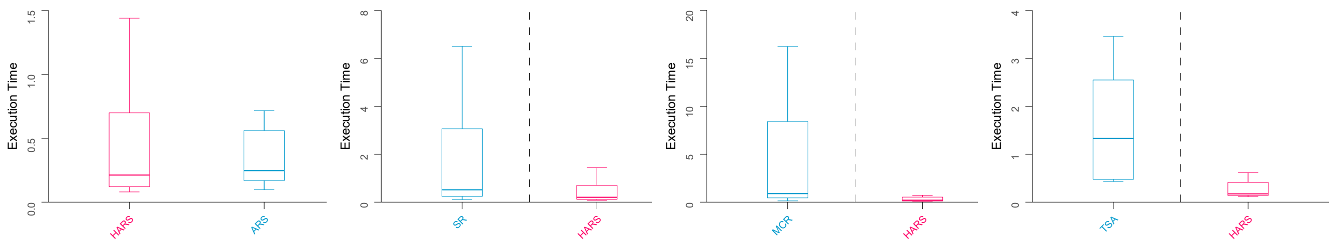
Among all of the programs that HARS is significantly better than ARS, there are 2 programs with an effective level of large or medium, and the number is 15, 30, and 36 for SR, MCR, and TSA, respectively. We can find that the distance metric can effectively help locate the buggy traces, as the information accumulates with the detection process. Therefore, HARS is not significantly better than ARS in most programs. As shown in Table II and Table III, HARS outperforms ARS on 38 programs but only significantly exceeded ARS on 9 programs. However, when the pattern distance metric can not provide active guidance for detection due to the lack of memory-access patterns in target programs, e.g., *Reredoer* and *Testarray*, HARS can still effectively detect bugs based on the designed heuristics. In particular, *Java.library.hashset* involves the operation of two threads to add and remove elements in the HashSet data structure in staggered order. Most of the memory-access patterns are very similar, while the buggy memory-access pattern information is very rare. Therefore, ARS wastes time searching for the correct traces which have those patterns. However, HARS can efficiently locate those buggy traces with the guidance of the heuristics. Based on the above analysis results, we can conclude that **Compared to SR, MCR and TSA, (1) HARS performs more stable on P-measure, which means it can effectively reduce the scheduling times spent on bug detection; (2) HARS achieves good performance for programs whose traces containing large amounts of diverse memory-access patterns; (3) HARS performs well on the programs where the buggy traces are sparsely distributed. Compared to ARS, HARS mainly improves the performance in the following two aspects: (1) HARS adopts heuristics to more effectively select traces that contain bugs without significantly increasing the time overhead than ARS. (2) HARS can effectively avoid exploring repeated traces.**

The results of the Scott-Knott ESD test on P-measure are shown in Fig. 4. Please note that the smaller the P-measure is, the better the performance of detection methods is. Therefore, the method in Fig. 4 should be as right as possible. Recall



(a) ARS (b) SR (c) MCR (d) TSA

Fig. 4: Scott-Knott ESD test results about P-measure with other baseline methods



(a) ARS (b) SR (c) MCR (d) TSA

Fig. 5: Scott-Knott ESD test results about executing time with other baseline methods

that SR, MCR, and TSA can run on 44, 31, and 14 programs in the benchmark, respectively. Thus we divided HARS into several groups with other baseline methods and perform Scott-Knott ESD test with them, respectively. **From Fig. 4 we can conclude that (1) HARS is significantly better than SR, TSA and MCR in terms of P-measure. (2) Although there is no statistically significant difference between HARS and ARS, it still maintains a weak advantage over ARS.**

RQ3: How efficient is HARS in detecting concurrency bugs compared with existing methods?

Design. We measure the time used for each method to execute the benchmark programs. In addition, we also use the Scott-Knott ESD test [29] to analyze the execution time result, and the results are shown in Fig. 5.

Result. Columns 9-12 in Table II report the execution time result. ARS, HARS, SR, MCR, and TSA take an average of 1.655, 1.790, 3.212, 84.428, and 2.497 seconds to detect bugs. HARS takes slightly more time than ARS due to the overhead of recording execution information and auxiliary heuristics sorting. However, this time overhead is not significant, as is shown in Fig. 5a. SR, TSA, and MCR take more time than HARS. SR uses less time than HARS on only 10 programs while HARS performs better on 35 programs. After an in-depth investigation, we find that many threads frequently access shared variables in programs such as `Account`, `Producerconsumer`, `Airline`, and `Log4j`. Therefore, HARS takes more time due to the overhead caused by maintaining the collected information. Moreover, the buggy traces occur frequently in the search space of `Atmoerror`, `Java.library.treeset`, `Mergesort`. Then SR can detect them easily through random sampling. Therefore, the monitoring overhead of HARS occupies the total execution time for those programs. Note that HARS or ARS only needs to detect bugs with one execution without entering the

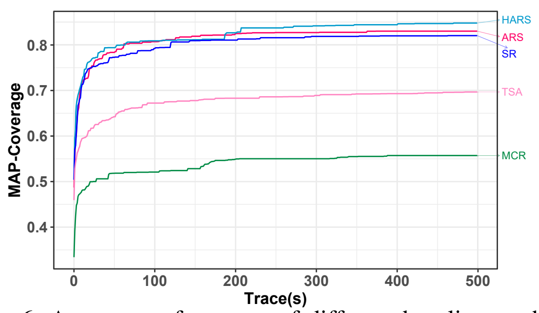


Fig. 6: Average performance of different baseline methods on MAP coverage

detection stage based on distance or heuristic sorting on some programs (such as *Allocationvector*, *Boundedbuffer*). Such programs are too simple or contain deadlock, and thus the performance of HARS or ARS is equal to SR. By increasing the executions, the randomness will decrease and the gap will be smaller. The detection time of MCR is about 500 times that of HARS, especially in some programs such as `Account` (1,661s), `Airline` (188s), and `MergeSort` (426s). Besides, since the time limit is 1 hour, MCR also can not detect bugs in other programs (e.g., `AlarmClock`, `Dekker`, `and WrongLock2`) within the limit. TSA detects fewer bugs and the time spent on detecting bugs is usually 5 times more than that of HARS, such as `LinkedList` and `Lamport`. Similar to P-measure results, the better method in Fig. 5 is also as right as possible. HARS is always ranked in the top group when compared with SR, MCR and TSA. **We can observe from Fig. 5 that HARS does not spend significantly more time than ARS and is significantly better than other approaches on execution time.**

RQ4: Does HARS achieve high MAP coverage?

Design. Existing work [18] has shown that MAP coverage is positively correlated with the bug revealing effectiveness.

Therefore, we adopt MAP coverage to indirectly indicates the bug-revealing effectiveness of HARS. Since most of the benchmark programs contain a small number of threads and a few read/write operations on shared variables, the pattern amount is not large. We run SR on the benchmark programs for two hours, collect all patterns that appeared in the traces, and used the number as an estimation of the total number of patterns. We then use the 5 methods to obtain 500 traces for each program and record the number of distinct patterns. We then plot the number of traces with MAP coverage [18].

Result. Fig. 6 shows the achieved MAP coverage of all methods on 28 benchmark programs. (Note that TSA can not complete the experiment in the given time limit on 9 programs, and MCR fails to finish the experiment on 8 programs) After running 500 traces on the 28 benchmark programs, HARS, ARS, SR, TSA and MCR reaches the MAP coverage of 84.81%, 83.03%, 82.05%, 69.68% and 55.72%, respectively. HARS achieves higher MAP coverage than ARS, SR, TSA, and MCR at the end. MCR achieves the lowest MAP coverage. We can also observe that HARS always achieves higher MAP coverage faster than the other methods. SR achieves high coverage faster than ARS at the beginning due to the randomness but can not exceed HARS, which also shows that the heuristics adopted can effectively solve the "cold start" problem of ARS in the early stage of operation. The goal of TSA is to achieve higher coverage of code pairs and synchronization pairs in concurrency programs. When it reaches the saturation synchronization coverage, the execution of TSA will fall into repeated scheduling, and the randomness can be substantially alleviated, thus making it difficult to detect new patterns. **The results show that HARS can achieve high MAP coverage faster than existing methods, which indirectly shows that HARS is effective in detecting concurrency bugs based on the conclusion in the previous study [18].**

VI. Threats to Validity

Threats to constructive validity. It mainly comes from the baseline methods. We select the adaptive randomized scheduling (ARS) method, the simple randomized (SR) algorithm, the thread-scheduling approach (TSA), and the maximal causality reduction (MCR) method. These four baseline methods can effectively represent the latest research in the area. ARS and SR are implemented on the JPF framework and TSA is implemented on the CalFuzzer framework. MCR is implemented on the code framework written by their authors.

Threats to internal validity. The internal threats are from parameter selection of HARS, which will affect the final results such as parameter N which decides the size of the queue. To relieve these factors, We tried multiple groups of parameters to select the most suitable parameter setting as the basis for experimental evaluation.

Threats to external validity. A key critical threat is the benchmarks. We miss some large-scale programs due to the limitations of JPF [23]. However, even for large-scale programs, the concurrency interleaving may also involve a small amount of code. Therefore, our benchmark can represent typical concurrency bugs to some extent. In addition, the adoption of third-party tools and libraries will also introduce some external threats. For instance, JPF is chosen due to its flexible scheduling control interface and is widely used and tested, which mitigates the threats to some extent.

VII. Related Work

Researchers have conducted lots of work about concurrency program detection [30]–[36]. The most related research is about exploring bug-inducing interleavings of concurrency executions. Two kinds of approaches, i.e., exhaustively exploration approaches [20], [37] and randomized approaches [4], [6], [38], are actively researched in this area.

Exhaustive Exploration Approaches. Symbolic execution [39] and model checking [37], [40]–[43] techniques are actively employed to exhaustively explore all possible interleavings. Happens-before relation analysis [30], the lockset algorithm [32], [33], and their extended versions [34], [35] are already adopted to detect concurrency bugs. The lockset algorithm is often improved to reduce detection cost and false positives [32], [33]. Some research work [44]–[46] adopts the lockset algorithm and happens-before relations both when detecting concurrency bugs. RaceChecker [30] prunes the redundant races and achieves the memory-access information based on the analysis of happens-before relation. In addition, it also uses a lockset when several threads access the corresponding shared memory simultaneously. In previous stateless model checking techniques, such as VeriSoft [47] and CHESS [48] compute specific scheduling to control the search of the state space, which reduces the redundant interleavings. MCR [3] minimizes the number of explored executions for stateless model checking concurrency programs based on the foundation of the maximal causal model and tries to overcome the limitations of happens-before relations. Maple [49] provides an online technique to predict untested interleavings that can potentially be exposed for a given test input by actively controlling the thread schedule.

Randomized Approaches. Random or fuzz testing is a common technique in sequential programs testing and achieves high efficiency in detecting general bugs. However, random testing tends to be adopted to control program scheduling in concurrency bug detection. PCT [4] adopts a strict schedule-stochastic technique to ensure appropriate probabilistic guarantees of detecting bugs. Heuristics search conducts some optimizations based on the randomized testing algorithm and guides more efficient explorations [6], [38]. TSA [10] builds a Java concurrency program testing method based on Calfuzzer. It analyzes the synchronization code of the target program, including the acquisition of shared variables, thread-related operations, etc. Empirical results show that the criteria should be covered in the dynamic execution to control the approaches to execute the program and cover more new synchronization pairs as much as possible. Our research relates to the randomized approaches since we adopt other strategies instead of random policies to control thread scheduling. Unlike other randomized approaches, we design heuristics related to the concurrency bugs to make up for the shortcomings of ARS method and add the pruning operation to improve the performance.

VIII. Conclusion

In this work, we propose the Heuristic-Enhanced Adaptive Randomized Scheduling (HARS) algorithm which adopts three effective heuristics related to concurrency bugs to improve the detection performance. We conduct empirical experiments on a large set of benchmarks and perform statistical analysis on the results to make reliable and statistically sound conclusions. We thoroughly analyze the results and provide useful insights into which HARS are practical. Compared with the previous adaptive randomized scheduling method ARS, HARS can locate the state space region containing bugs faster; Compared with randomized detection methods and exhaustive search methods, HARS shows better effectiveness and efficiency.

Acknowledgment

This work was partially supported by the National Natural Science Foundation of China under Grant Numbers 61872263, U1836214, 61802275, Intelligent Manufacturing Special Fund of Tianjin under Grant Number 20201180. Shuang Liu is the corresponding author.

References

[1] J. Huang. Debugging concurrent software: Advances and challenges. *Journal of Computer Science and Technology*, 31:861–868, 2016.
[2] F. A. Bianchi, A. Margara, and M. Pezzè. A survey of recent trends in testing concurrent software systems. *IEEE Transactions on Software Engineering*, 44:747–783, 2018.
[3] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI'15*, 2015.
[4] S. Burckhardt, Pravesh Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS XV*, 2010.
[5] C. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *SIGSOFT '08*, 2008.
[6] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI '08*, 2008.
[7] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07*, 2007.
[8] E. Noonan, E. Mercer, and Neha Rungta. Vector-clock based partial order reduction for jpf. *ACM SIGSOFT Softw. Eng. Notes*, 39:1–5, 2014.
[9] M. Kokologiannakis, A. Raad, and V. Vafeiadis. Effective lock handling in stateless model checking. *Proceedings of the ACM on Programming Languages*, 3:1 – 26, 2019.
[10] S. Hong, J. Ahn, S. Park, and M. Kim. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *ISSTA'12*, 2012.
[11] Z. Wang, D. Zhang, S. Liu, and J. Sun. Adaptive randomized scheduling for concurrency bug detection. In *ICECCS'19*, pages 124–133, 2019.
[12] S. Park, R. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *ICSE'10*, 2010.
[13] S. Park, R. Vuduc, and M. J. Harrold. A unified approach for localizing non-deadlock concurrency bugs. In *ICSE'12*, pages 51–60, 2012.
[14] Hars. https://github.com/HARS2020/HARS, 2020.
[15] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *PADTAD '06*, 2006.
[16] Y. Jiang J. Gao, X. Yang, H. Liu, W. Ying, and W. Sun. Managing concurrent testing of data race with comrade. In *ISSTA'18*, 2018.
[17] T. Yu, W. Wen, X. Han, and J. H. Hayes. Conpredictor: Concurrency defect prediction in real-world applications. *IEEE Transactions on Software Engineering*, 45:558–575, 2019.
[18] Z. Wang, Y. Zhao, S. Liu, J. Sun, X. Chen, and H. Lin. Map-coverage: A novel coverage criterion for testing thread-safe classes. In *ASE'15*, pages 722–734, 2019.
[19] F. Chen, T. Serbanuta, and G. Rosu. jpredictor: a predictive runtime analysis tool for java. In *ICSE '08*, 2008.
[20] S. Huang and J. Huang. Maximal causality reduction for tso and pso. In *OOPSLA'16*, 2016.
[21] C. Hammer, J. T. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE'08*, pages 231–240, 2008.
[22] L. Wang and S. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP '06*, 2006.
[23] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2:366–381, 2000.
[24] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1:196–202, 1945.
[25] Y. Benjamini. The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, 29:1165–1188, 2001.
[26] E. Kocaguneli, T. Menzies, Jacky W. Keung, David R. Cok, and Raymond J. Madachy. Active learning and effort estimation: Finding the essential content of software effort estimation data. *IEEE Transactions on Software Engineering*, pages 1040–1053, 2013.
[27] N. Cliff. Ordinal methods for behavioral data analysis. 1996.
[28] R. Jeanine and K. Jeffrey. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? 2006.
[29] E. Jelihovschi, José Cláudio Faria, and I. Allaman. Scottknott: A package for performing the scott-knott clustering algorithm in r. *Trends in Applied and Computational Mathematics*, 15:003–017, 2014.
[30] K. Lu, Z. Wu, X. Wang, C. Chen, and X. Zhou. Racechecker: Efficient identification of harmful data races. *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 78–85, 2015.
[31] Z. Ding, R. Wang, J. Hu, and Y. Liu. Detecting bugs of concurrent programs with program invariants. *IEEE Transactions on Reliability*, 66:425–439, 2017.
[32] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03*, 2003.
[33] Tayfun Elmas, S. Qadeer, and S. Tasiran. Precise race detection and efficient model checking using locksets. *Microsoft Tech Report*, 2006.
[34] 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 *OOPSLA'10*, 2010.
[35] R. O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPoPP '03*, 2003.
[36] A. Habib. Finding concurrency bugs using graph-based anomaly detection in big code. In *OOPSLA'16*, 2016.
[37] Brian Norris and Brian Demsky. Cdschecker: checking concurrent data structures written with c/c++ atomics. In *OOPSLA '13*, 2013.
[38] Katherine E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: effective unit testing for concurrency libraries. In *PPoPP '10*, 2010.
[39] K. Sen and Gul A. Agha. Automated systematic testing of open distributed programs. In *FASE'06*, 2006.
[40] S. Lauterburg, Mi. Dotta, and D. Marinov. A framework for state-space exploration of java-based actor programs. In *ASE'09*, 2009.
[41] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE '10*, 2010.
[42] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing c++ concurrency. In *POPL '11*, 2011.
[43] J. Dingel and H. Liang. Automating comprehensive safety analysis of concurrent programs using verisoft and txl. In *SIGSOFT '04*, 2004.
[44] Yuan Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05*, 2005.
[45] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07*, 2007.
[46] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs. *Concurrency and Computation: Practice and Experience*, 19, 2007.
[47] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97*, 1997.
[48] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI'08*, 2008.
[49] J. Yu, S. Narayanasamy, and C. Pereira. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA '12*, 2012.