Deterministic Dynamic Race Detection Across Program Versions

Sri Varun Poluri  
Indian Institute of Science, Bangalore  
sri.varun@csa.iisc.ernet.in

Murali Krishna Ramanathan  
Indian Institute of Science, Bangalore  
muralikrishna@csa.iisc.ernet.in

Abstract—Dynamic race detectors operate by analyzing execution traces of programs to detect races in multithreaded programs. As the thread interleavings influence these traces, the sets of races detected across multiple runs of the detector can vary. This non-determinism without any change in program source and input can reduce programmer confidence in using the detector. From an organizational perspective, a defect needs to be reported consistently until it is fixed. Non-determinism complicates the work flow and the problem is further exacerbated with modifications to the program.

In this paper, we propose a framework for deterministic dynamic race detection that ensures detection of races until they are fixed, even across program versions. The design attempts to preserve the racy behavior with changes to the program source and input can reduce programmer confidence in using the detector. From an organizational perspective, a defect needs to be reported consistently until it is fixed. Non-determinism complicates the work flow and the problem is further exacerbated with modifications to the program.

We have implemented a framework, named STABLER, and evaluated our ideas by applying popular race detectors (DJIIT+, FastTrack) on different versions of many open-source multithreaded Java programs. Our experimental results show that we are able to detect all the unfixed races consistently across major releases of the program. For both the detectors, the maximum incurred slowdown, with our framework, for record and replay is 1.2x and 2.29x respectively. We also perform user experiments where volunteers fixed a significant number of races. In spite of these changes, our framework is effective in its ability to detect all the unfixed races.

I. INTRODUCTION

Detecting and fixing races in multithreaded applications is vital to prevent non-deterministic errors [24], [36], [39], [15]. Race detectors based on static analysis [8], [14], [42] can be imprecise due to poor abstraction of threads. On the other hand, the use of concrete data to detect races by dynamic race detectors [15], [33], [38], [37] ensures detection with fewer false positives. Dynamic detectors instrument multithreaded programs, execute them on a test input and analyze the generated traces to detect races precisely [15], [33]. However, there is a major hindrance pertaining to practical deployment of dynamic race detectors because multiple invocations of race detectors on the same test input may detect different sets of races. This is because each invocation can correspond to a different thread interleaving and thereby, a different execution trace. Consequently, analysis of dissimilar execution traces can yield different sets of races.

Practical adoption of program analysis tools depend on the reliability of the tool as much as on the quality of the reported defects [27]. If a program is unchanged and each invocation of the detector reports a different set of defects, the programmer will lose confidence in the results generated by the tool [34]. More specifically, when a previously detected race is not reported, it becomes unclear on whether: (a) the race is fixed, or (b) the analyzed execution trace is different compared to the execution trace on which the race is initially reported. Therefore, an unreported race cannot assure the developer that the defect is fixed.

Recording and replaying the thread schedules for a given input can potentially be used to address this problem [13], [11], [12], [4], [21], [25], [26]. For a given test input, when record and replay is employed, the execution trace becomes deterministic as the same thread interleaving is followed across multiple invocations and consequently the detector will report the same set of races. However, a straight-forward application of this solution is infeasible in practice. This is because the defects reported by analysis tools are not necessarily fixed by developers immediately [6]. In other words, when the reported races are in the unfixed state, other modifications to the source code can occur [40]. Therefore, while the schedule corresponding to the initial set of reported races is recorded on one version of the program, the replay may need to happen on a modified version to catch unfixed races. While the unfixed races can also be detected using any other buggy schedule, it may be difficult to manifest if the buggy schedule happens rarely [34]. Instead, it may be more practical to use the hints from the detected run (on the original version) during replay.

The significance of fixing critical race conditions before releasing the software motivate the deployment of a race detector during software development. The practical challenges posed by non-determinism in the process motivate the design of a framework for deterministic dynamic race detection. The key features of the framework should be as follows: (a) a race detected in any version of the program is consistently detected in subsequent versions of the program until it is fixed, (b) function seamlessly even when modifications to the source include addition and deletion of locks and (shared) memory operations, and (c) minimal overhead on top of dynamic race detection.

In this paper, we design a framework for deterministic dynamic race detection across program versions with the above features. When the execution trace associated with a program \( P \) is analyzed using a dynamic race detector, the corre-
Fig. 1: Illustrative example: Blocked schedule.

Fig. 2: Architecture.

 responding schedule is also recorded. We represent a schedule as a sequence of dynamic execution indices [45], which is composed of source line numbers along with other attributes. When the program is unchanged, replaying the schedule and employing the race detector on the execution ensures detection of the same set of races. When the program is modified to \( P' \), we initially perform a mapping of the source line numbers from \( P \) to \( P' \). We achieve this using a tree edit distance algorithm (RTED [32]) that maps the nodes in the abstract syntax trees of the two versions. Subsequently, we transform the recorded schedule on \( P \) using the mapped source line numbers to generate an appropriate schedule on \( P' \), replay \( P' \) with the transformed schedule and perform race detection on the replayed execution. We manipulate the recorded schedules appropriately to address the challenges pertaining to addition (or deletion) of lock and shared memory operations.

We have incorporated our approach as part of a framework, named STABLER, that is built on top of RoadRunner [16]. We perform elaborate validation by performing dynamic race detection using popular detectors, DJIT+ [33] and FastTrack (FT) [15], on multiple Java programs. Without the use of our framework, when race detection is applied on the newer version of the associated benchmark, we observe that up to 15% races are not reported, in spite of repeated executions (10 times). Using our framework, we are able to deterministically detect unresolved races across program versions. When we ran DJIT+ using STABLER, the framework generates 27 (2%) additional schedules to ensure detection of all unresolved races with a maximum record and replay overhead of 1.13 and 1.79 respectively. For FT, the framework generates 21 (%) additional schedules with a maximum record and replay overhead of 1.21 and 2.29 across program versions. We also performed experiments where volunteer programmers are asked to fix a few races detected on the initial version. Using STABLER, we are able to detect all the unresolved races in the modified versions of the benchmarks considered which had versatile code modifications.

II. Motivation

In this section, we motivate the need for a deterministic framework for dynamic race detection across program versions. We analyze the execution of eight Java programs with DJIT+ [33], a popular race detector, by running the programs using pre-existing tests [46] or generating race exposing multithreaded tests. The executions are analyzed by the detector independently and the process is repeated five times on the same version of the program. Ideally, there should not be any change in the set of detected races because neither the code nor the tests are modified. However, we observe that different races are detected across multiple runs and across all the benchmarks.

In Section I, we discussed that modifications to source code can occur even before all the races reported by analysis tools are fixed. This made it imperative that a schedule recorded on one version of the program needs to be replayed on a modified version. We now motivate the challenges associated with replaying a recorded schedule on a modified version.

Figure 1(i) presents an execution schedule on h2-v1.3.151 and the order of operations, which starts with label 1. A race detector reports two races on access to \( \text{pos} : (t_2: 144, t_1: 179) \) and \( (t_1: 160, t_2: 144) \). From Figure 1(ii), we observe that the first race is fixed in v1.3.152. If we transform the schedule naively as shown in Figure 1(ii), the new schedule becomes infeasible as the methods \( \text{setFileLength} \) and \( \text{readFully} \) are synchronized. Any attempt to replay the schedule will result in the execution being blocked. Therefore, the first race on \( \text{pos} (t_1: 160, t_2: 144) \), was previously reported will not be reported. Moreover, randomized execution on the newer version may not necessarily detect the race definitively. Thus, the design of the framework should ensure that schedule obstructing modifications are handled appropriately.

If the obstructing component of the schedule is eliminated from the constraint set during a replay, then the two methods \( \text{readFully} \) and \( \text{setFileLength} \) can execute atomically in some order. If \( \text{readFully} \) executes before \( \text{setFileLength} \), then a happens-before [23] relation is introduced between the release of lock by the former followed by the acquisition by the latter. The introduction of the happens-before relation suppresses the race \( (t_1: 160, t_2: 144) \) from being detected by DJIT+. Therefore, the design of the framework should ensure that schedule modifications do not result in suppression of races. Apart from the aforementioned issues, introduction of additional shared memory accesses can cause the execution path to change leading to nondeterminism.

In this paper, we propose a framework to enable deterministic dynamic race detection. For the different versions of the benchmarks considered in our experiments, STABLER was able to detect all the races deterministically.

III. Design

Figure 2 presents the architecture of our framework, named STABLER. The core components of the framework are Recorder, Mapper, Transformer and Replayer. The
Recorder is responsible for recording the schedule for each run of the program. The Mapper provides a mapping of source line numbers between two program versions. The Transformer transforms the schedule such that it is appropriate for the modified version. Replayer replays the transformed schedule on the modified program using the same test input. If the Replayer fails to replay successfully, it makes appropriate modifications to the schedule so that unfixed races continue to be detected.

A. Preliminaries

We now define terms that are used in the rest of the paper. Dynamic execution index ($e$) is a tuple composed of line numbers $(l_1,l_2,...,l_n)$, which identifies the dynamic execution instance of an instruction uniquely[45].

Execution dependence relation ($e_1 \rightarrow e_2$) is a binary relation between execution indices $e_1$ and $e_2$. It specifies that the instruction at index $e_2$ can be executed only after executing the instruction at $e_1$. Execution dependence relation is transitive i.e., if ($e_1 \rightarrow e_2$) and ($e_2 \rightarrow e_3$) hold, then ($e_1 \rightarrow e_3$) holds.

Synchronization point refers to an execution index of one of the following types of instructions: thread creation, thread exit, lock acquisition and lock release.

Synchronization region [a, b] denotes a region between two synchronization points of a single thread. It is represented by a logical clock [a, b] where a and b represent the start and end logical time of the region respectively.

Synchronization schedule ($S$) is a set of execution dependence relations such that all execution indices correspond to synchronization points.

Memory schedule ($M$) is a set of execution dependence relations such that all execution indices correspond to shared memory accesses.

Race pair ($e_1, e_2$) is a pair of execution indices representing shared memory accesses by different threads, where at least one of the accesses is a write and the accesses are not ordered by a happens-before relation[23].

B. Recorder

We leverage the approach presented in Peregrine [12] for recording the schedule on the initial version. We maintain a hybrid schedule which is an intelligent union of synchronization and memory schedules. Instead of recording the order between all memory accesses, only the racing memory accesses in an execution that contribute to the non-determinism need to be recorded (see [12] for more details). The synchronization points also help prune the racing pairs to minimize the recording overhead.

We describe the approach used to record a schedule of a program execution on a given input in Algorithm 1. Initially $S$ and $M$ are empty and activeThreads is initialized to the main thread under execution. Lines 3–13 show the construction of the synchronization schedule ($S$), detection of set of racing pairs ($D$) and additions of new threads in the execution. If the instruction is a synchronization point, we add an execution dependence between the current execution index($e$) and the index of the previous synchronization point ($e_{prev}$). This set of execution dependence relations among the synchronization points forms the synchronization schedule ($S$).

<table>
<thead>
<tr>
<th>Algorithm 1 Recorder</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Program: $P$, Program Input: $I$</td>
</tr>
<tr>
<td><strong>Output:</strong> Schedule: $\sigma$, Races Detected: $D$</td>
</tr>
<tr>
<td>1: $e_{prev} :=$ execution index of starting instruction</td>
</tr>
<tr>
<td>2: activeThreads := initial set of executable threads</td>
</tr>
<tr>
<td>3: while activeThreads $\neq \emptyset$ do</td>
</tr>
<tr>
<td>4: $t :=$ a random thread from activeThreads</td>
</tr>
<tr>
<td>5: inst := next instruction to be executed by $t$</td>
</tr>
<tr>
<td>6: $e :=$ execution index of $inst$</td>
</tr>
<tr>
<td>7: if inst is a synchronization point then</td>
</tr>
<tr>
<td>8: $S := S \cup (e_{prev} \rightarrow e)$; $e_{prev} := e$;</td>
</tr>
<tr>
<td>9: else if inst is a memory access then</td>
</tr>
<tr>
<td>10: $D := D \cup \text{raceDetector}(inst)$;</td>
</tr>
<tr>
<td>11: else if inst is a thread creation point then</td>
</tr>
<tr>
<td>12: add the new thread to activeThreads</td>
</tr>
<tr>
<td>13: Execute inst;</td>
</tr>
<tr>
<td>14: for all ($e_1, e_2$) $\in D$ do</td>
</tr>
<tr>
<td>15: [$a_1, b_1]$ := region($e_1$); [$a_2, b_2]$ := region($e_2$);</td>
</tr>
<tr>
<td>16: if $a_1 &lt; b_2$ &amp;&amp; $a_2 &lt; b_1$ then $M := M \cup (e_1 \rightarrow e_2)$;</td>
</tr>
<tr>
<td>17: $\sigma := S \cup M$;</td>
</tr>
</tbody>
</table>

If the instruction is a memory access, we use race detectors [33], [15] (represented by the raceDetector auxiliary function) to detect whether the current access races with any other memory access. The output of raceDetector($inst$) is ($e_p$, $e$), where $e_p$ is index of a previously executed memory access that is racing with the current access and $e$ is the index of the current access. If the instruction is a thread creation operation then that particular thread is added to the list of activeThreads.

We construct the memory schedule by iterating over all the pairs in $D$ and specifying an order between any two racing accesses. The auxiliary function region($e$) returns the synchronized regions associated with $e$ and helps identify the encompassing synchronization points. If the regions corresponding to a racing pair do not overlap, then it is not necessary to specify an access order explicitly between the two accesses because the synchronization schedule implicitly orders them. Therefore, the memory schedule ($M$) is updated only when the synchronized regions overlap (lines 14 to 16). The overall hybrid schedule ($\sigma$) for the execution is constructed by combining the synchronization and memory schedules.

In the rest of the paper, we will use the following notation to specify an element ($\tau = X^a_j$) in a schedule ($\sigma$). The possible values of $X$ include $\mathbb{L}$, $\mathbb{R}$, $\mathbb{W}$ and $\mathbb{U}$ referring to the lock, read, write and unlock operations respectively. The subscript $a$ represents the object on which the operation is performed and the superscript corresponds to the index of the operation within a thread. For example, $\mathbb{R}_x^j$ refers to a read performed on object $x$ and is the sixth operation in the thread. $e(t_j, \tau)$ specifies the dynamic execution index corresponding to the element $\tau$ in thread $t_j$, thread($e$) returns the thread that executes the instruction.
Fig. 3: Hybrid schedule

Algorithm 2 Replayer

Input: Schedule: \(\sigma\), Program: \(P'\), Program Input: \(I\)
Output: Races Detected: \(D\)

1: \(\text{waitingThreads} := \emptyset\), \(\text{activeThreads} := T\)
2: \(\text{blockedIndices} := \emptyset\)
3: while \((\text{activeThreads} \cup \text{waitingThreads}) \neq \emptyset\) do
4: \(t := \text{random thread from activeThreads}\)
5: \(\text{inst} := \text{next statement to be executed by} \ t\)
6: \(e := \text{execution index of inst}\)
7: if \(\exists (e_u \rightarrow e) \in \sigma \text{ s.t. } \neg \text{executed}(e_u)\) then
8: \(\text{blockedIndices} := \text{blockedIndices} \cup e\)
9: Move \(t\) to \(\text{waitingThreads}\) and goto 4
10: Execute \(\text{inst}\);
11: if \(\text{inst}\) is a memory access then
12: \(D := D \cup \text{raceDetector}(\text{inst})\)
13: for all \((e \rightarrow e_0) \in \sigma \text{ s.t. } e_0 \in \text{blockedIndices}\) do
14: Remove \(e_0\) from \(\text{blockedIndices}\)
15: Move thread \((e_0)\) to activeThreads

Algorithm 2 shows the working of the Replayer. It maintains two sets of threads: \(\text{waitingThreads}\) contains the threads that are in a waiting state because an instruction needs to be executed by some other thread, and \(\text{activeThreads}\) contains the threads that can execute. Initially, \(\text{activeThreads}\) contains all threads that are available to execute \((T)\) and we select a random thread to execute. If the instruction under consideration is dependent on other instructions which are yet to be executed, we add the associated execution index to the \(\text{blockedIndices}\) set as shown between lines 7 – 9. If the instruction is not blocked, it is executed and race detection is performed (line 10 – 12). Subsequently, the thread identifies all the threads that are waiting for it to execute the current instruction (based on the execution index) and transfers them to \(\text{activeThreads}\) (lines 13 – 15). When a thread completes its execution, it ceases to exist in either \(\text{activeThreads}\) or \(\text{waitingThreads}\). The replay concludes when both sets are empty. As the execution follows the schedule given by \(\sigma\), we detect the same set of races \((D)\) that is previously detected.

We will now describe the replay of the schedule given in Figure 3. Initially, threads \(t_1\) and \(t_2\) are added to \(\text{activeThreads}\). Consider that \(t_2\) is chosen to execute at line 4 of Algorithm 2. When the instruction corresponding to \(W_1^2\) is reached, the Replayer observes that a dependency \(e(t_1, R_2^2) \rightarrow e(t_2, W_2^2)\) exists in \(\sigma\). Therefore, \(e(t_2, W_2^2)\) is moved to \(\text{blockedIndices}\) and \(t_2\) is moved to \(\text{waitingThreads}\). Thread \(t_1\) gets an opportunity to execute and is free to execute all the instructions leading up to \(e(t_1, W_2^1)\). In the interim, when \(e(t_1, R_2^2)\) is executed, it moves \(t_2\) to \(\text{activeThreads}\) because the dependent execution index on which \(t_2\) is waiting has finished execution. Subsequently, \(t_1\) is moved to \(\text{waiting state}\) when it reaches \(e(t_1, W_2^2)\) and \(t_2\) is scheduled. The replay is orchestrated in this manner to ensure loss-less detection.

D. Transformer

Our framework should enable replay of execution on version different to the one on which recording is performed. Algorithm 3 takes a schedule \((\sigma)\) recorded on \(P\) and sources of programs \(P\) and \(P'\) as input. It outputs a schedule \((\sigma')\) that can be used to replay the execution on \(P'\) to ensure detection of unfixed races. This requires appropriate transformation of the recorded schedule. A schedule is composed of execution indices and hence requires transformation of the indices. As source line numbers form the core component of the indices, mapping the line numbers across versions becomes necessary.

Algorithm 3 Transformer

Input: Program \((P)\), Modified Program \((P')\), Schedule \((\sigma)\)
Output: Modified schedule \((\sigma')\)

1: \(\psi' := \text{astGen}(P')\); \(\psi' := \text{astGen}(P')\);
2: \(\delta := \text{rted}(\psi, \psi')\);
3: for each \((e_1 \rightarrow e_2) \in \sigma\) do
4: \(e_1' := \text{update}(e_1, \delta); e_2' := \text{update}(e_2, \delta)\)
5: \(\sigma' := \sigma \cup (e_1' \rightarrow e_2')\)

Applying standard diff tools (e.g., [2], [1]) will fail to map similar instructions as they are syntax agnostic and flag benign changes. Therefore, we compare the abstract syntax trees (ASTs) of the two versions to map the line numbers. We
use ANTLR[31] to generate the ASTs (line 1 of the algorithm). It takes the source code and the grammar corresponding to the language as input and outputs the ASTs. Subsequently, we apply a state-of-the-art tree edit distance algorithm (RTED [32]) to compare the generated ASTs. This outputs $\delta$ at line 2 that specifies the map of line numbers between the two versions.

The algorithm then iterates over all the execution indices present in the schedule $\sigma$ and updates the execution index based on the computed map ($\delta$) to obtain the modified execution indices. Subsequently, we add the dependencies between the modified execution indices to construct $\sigma'$. We illustrate the working of the Transformer using an example.

Figure 4(a) shows two versions of a simple Java program that includes methods foo and bar. A partial execution trace when two threads invoke foo with true and false on the first version is shown in Figure 4(b). The trace also shows execution indices ($e_1, e_2, e_3$). The second column of Figure 4(c) shows the definition of these indices [45]. Observe that the recorded schedule ($\sigma$) cannot be replayed on $P'$. For example, the execution index $\langle 108, 109 \rangle$ corresponds to line 109 in $P'$ instead of the required line (108).

The transformation of the indices is achieved using Algorithm 3. Figure 4(d) shows the ASTs of the two versions. Applying the tree edit distance algorithm helps in discarding benign changes. For example, the invocation of bar in $P$ is moved to a different line in $P'$. However, we observe that there is no difference in the corresponding nodes in the ASTs. On the other hand, when the conditional is discarded from the implementation of bar in $P$, an access to a field $y$ is added in $P'$, rted identifies the changes. Internally, it performs a post-order traversal of the trees and maps the nodes as given in Figure 4(e). When the execution indices need to be transformed, the associated source lines are transformed which is achieved using mapping of the AST nodes. For example, line 109 corresponds to nodes (2, 3, 4). These nodes map to (5, 6, 7) which correspond to line 108 in $P'$. We transform all the execution indices in this manner and the transformed schedule ($\sigma'$) is given in Figure 4.

E. Challenges in replaying

Seamless replay of all transformed schedules is not possible. In Section II, we present a couple of scenarios where the replay becomes infeasible or does not achieve the purpose of detecting unfixed races. We now describe these scenarios in detail and explain the methodology employed to address them.

Initially, STABLER replays using the transformed schedule and checks whether the execution is blocked. If yes, it resolves the block by using Algorithm 4. Once an unblocked schedule is generated and the execution completes, all the races detected during the execution are reported. If execution failed to detect a race pair, STABLER generates additional schedules by considering two cases (a) both accesses execute but the detector failed to detect, (uses Algorithm 5 and Algorithm 6), (b) one of the accesses is not executed (uses Algorithm 7).

Algorithm 4 handleBlocking

Input: Program $P'$, Schedule $\sigma'$, Input $I$
Output: Schedule $\sigma''$

1: while REPLayer ($\sigma', P', I$) is blocked do
2: Identify ($e_1 \rightarrow e_2$) in $\sigma'$ blocking the execution
3: $\sigma' := \sigma' - (e_1 \rightarrow e_2)$
4: Update waitingThreads, blockedIndices suitably
5: for each ($e_1 \rightarrow e_2$) $\in \sigma'$ do
6: if $e_1$ and $e_2$ are guarded by the same lock then
7: $\sigma' := \sigma' - (e_1 \rightarrow e_2)$

Algorithm 4 presents a simple yet effective approach to address problems due to addition of new locks. Whenever the replay is blocked, it identifies the constraint that causes the block and eliminates it from the transformed schedule.
Subsequently, when the replay is complete, it checks all the dependencies in the schedule and if the corresponding indices share the same lock, then the dependency is removed from the schedule (lines 5–7). For the example under consideration, the constraint \( e(t_1, R^2) \rightarrow \sigma' \) is eliminated from \( \sigma' \) initially as it causes the schedule to block. When the replay is complete, it eliminates \( e(t_2, W^3) \rightarrow e(t_1, W^3) \) from the schedule because the indices share lock \( \alpha \) and can result in a block in another schedule (e.g., when \( t_1 \) executes initially).

\[
\begin{align*}
t_1: & \quad L^1 \rightarrow R^2 \rightarrow W^3 \rightarrow U^4 \rightarrow R^6 \\
t_2: & \quad L^0 \rightarrow W^1 \rightarrow U^2 \rightarrow \sigma' \rightarrow \sigma \\
D: & \{ (t_1, W^0_1, t_2, W^0_2), (t_1, W^0_1, t_1, W^0_3) \}
\end{align*}
\]

Fig. 6: Possible schedule after blocking from Fig. 5 is resolved.

2) Suppressed defects: The primary goal of STABLER is to report all previously detected unfixed races, not replay the execution without blocking. When the cross-out dependencies in Figure 5 are removed, a schedule that is possible is given in Figure 6. However, analyzing this execution will not detect the unfixed race given by \( (t_1, W^0_1, t_2, R^2) \) due to the introduction of a happens-before relation \( (U^4_1 \rightarrow L^0_n) \). Eliminating constraints can result in many feasible schedules and not all detected races occur on all schedules obviously.

**Algorithm 5 handleSuppression**

**Input:** Program \( P \), Schedule \( \sigma' \), Input \( I \), Race Pair: \( (e_1, e_2) \)

**Output:** \( \sigma'' \)

1: \( t_1 := \text{thread}(e_1); t_2 := \text{thread}(e_2) \)
2: \( E_1 := \text{synchronization points executed after } e_1 \text{ in } t_1 \)
3: For \( \text{orphan unlocks in } E_1 \), add corresponding locks to \( E_1 \)
4: \( E_2 := \text{synchronization points executed before } e_2 \text{ in } t_2 \)
5: For \( \text{orphan locks in } E_2 \), add corresponding unlocks to \( E_2 \)
6: for each \( e \in E_1 \) do
7: \( \text{if } (e_2 \rightarrow e) \notin \sigma' \text{ then } S_1 := S_1 \cup \{ e \} \)
8: for each \( e \in E_2 \) do
9: \( \text{if } (e \rightarrow e_1) \notin \sigma' \text{ then } S_2 := S_2 \cup \{ e \} \)
10: for each \( e \in S_1 \) and \( e' \in S_2 \)
11: \( \text{if } \text{ob} j(e) = \text{ob} j(e') \text{ then } \sigma'' := \sigma' \cup (e' \rightarrow e) \)

Algorithm 5 presents an approach that handles suppression of races due to eliminated dependencies. Broadly, it identifies the synchronization operations that can introduce a happens-before relation and constructs newer dependencies to prevent this introduction. The process is performed in a scalable manner while ensuring that it does not cause blocked executions.

We now describe the functioning of Algorithm 5 using the undetected race \( e(t_1, W^3_2), e(t_2, R^2_3) \) in Figure 6. When the algorithm is applied, \( E_1 \) is computed as \( \{ e(t_1, U^4_1) \} \) (at line 2) and because it is an orphan unlock, the index of the corresponding lock is added to get \( \{ e(t_1, L^0_n), e(t_1, U^4_1) \} \). Similarly, \( E_2 \) is computed as \( \{ e(t_2, L^0_n), e(t_2, U^4_2) \} \). For the current example, \( S_1 \) and \( S_2 \) are equivalent to \( E_1 \) and \( E_2 \). We consider all the four possible pairs of execution indices and since they correspond to operations on the same object, we add the dependencies \( L^0_n \rightarrow L^0_n, L^0_n \rightarrow U^4_1, U^4_0 \rightarrow L^1_3, \) and \( U^4_3 \rightarrow U^4_0 (e(... \text{ removed for brevity}) \). This can be represented as \( U^3_3 \rightarrow L^1_3 \) after removing the program order dependencies. The resulting schedule is given in Figure 7 and the absence of a happens-before ensures the detection of the suppressed race.

![Fig. 7: Added dependencies to prevent suppressed races.](image)

3) Masked Defects: Modifications to the source can also include addition (or deletion) of shared memory accesses that can hinder the process of detecting previously detected races. Figure 8(a) shows a write access \( W^1_2 \) that is additionally performed by \( t_1 \) compared to the original schedule given in Figure 3. Because there is no constraint on the write access, assume it is executed before \( R^2_3 \) by \( t_2 \). Analyzing the execution will result in the race \( (t_1, W^1_2, t_2, R^2_3) \) not being reported. This is
because the race detectors [15], [33] maintain only the latest accesses by each thread for scalability. Therefore, the access given by $w^1_y$ is masked by $w^2_y$.

**Algorithm 6 handleMasking**

**Input:** Program $P'$, Schedule $\sigma'$, Input $I$, Race Pair: $(e_1, e_2)$

**Output:** $\sigma''$

1. $t_1 := \text{thread}(e_1)$; $t_2 := \text{thread}(e_2)$
2. $E := \text{indices for accesses on obj}_3(e_1)$ happening after $e_1$ in $t_1$
3. for each $e \in E$ do
4. if $\text{op}(e) = \text{op}(e_1)$ \&\& ($e_2 \rightarrow e$) \&\& $\sigma' \notin \sigma''$ then

Algorithm 6 presents our approach to prevent masking by introducing additional constraints on new shared memory operations. If $(e_1, e_2)$ is the race that is not detected, we obtain the object $(o)$ on which $e_1$ is performed and identify all the indices for operations performed by $t_1$ on $o$ after $e_1$ (line 2). For each such index $e$, if the operation type is equivalent to that of $e_1$ (e.g., write) and there is no dependency between $e_2$ and $e$, we introduce a dependency. The addition of a dependency can potentially introduce blocking. In such scenarios, we apply Algorithm 4 where only the existing dependencies are eliminated and the newly added dependencies persist. For each race that is not reported due to masking, an additional schedule is constructed.

Applying the current algorithm, we will add the dependency $\sigma$($e_2$) = ($e_1$, $e_2$) as shown in Figure 8(b). Analyzing the execution will report the race under consideration. Moreover, observe that the race ($t_1$; $w^2_y, t_2; w^2_y$) is also missed due to masking. If we add the dependency $w^2_y \rightarrow w^2_y$, it conflicts with $U^2_x \rightarrow L^3_y$. The latter dependency is relaxed using Algorithm 4 and the newly constructed schedule enables detection of the write-write race.

Fig. 9: Divergent schedule

4) **Divergent Schedules:** The above scenarios correspond to non-detection of races even when the associated races occur. Modifications to the code and subsequent replay can cause the execution path to diverge such that the associated races do not occur. This is because the control flow can be influenced by the added (or deleted) memory accesses. Figure 9(a) contains an additional read access (denoted by $R^*_y$) compared to the hybrid schedule shown in Figure 3. Consider the read to be a conditional ($y \geq 0$) and the values written at $w^2_y$ by $t_1$ and $t_2$ is $-1$ and $1$ respectively. The access can conceivably read the value written by $t_1$ and the access to $R^*_y$ may not happen resulting in the race detector not reporting the unresolved race ($t_2 : w^1_y, t_1 : R^*_y$).

Algorithm 7 presents an approach to handle divergent schedules. Initially, it ensures that all read accesses that existed previously read values written at locations in the original schedule (lines 1 - 4). We consider the read accesses that are introduced in the new version as dangling reads. The reads whose writes from the original version are deleted are also considered dangling. For each such read, we construct a sequence of all possible writes that can influence the read denoted by $P_w[e]$ (lines 5 - 7). Subsequently, we perform a localized search exploring the combination of write-read accesses and synthesizes new schedules until the race is reported or the threshold for the search is reached (lines 8 - 13). Threshold value was set to 100 for our experiments. While the number of schedules is combinatorial, the search space is localized to overlapping synchronized regions. The systematic exploration of the schedules is inspired by CHESS [28].

**Algorithm 7 handleDivergence**

**Input:** Program $P'$, Schedule $\sigma'$, Input $I$, Race Pair: $(e_1, e_2)$

**Output:** $\Sigma$

1. for each $(e', e) \in \sigma' \land \text{op}(e) = \text{read}$ do
2. $E := \text{write indices on obj}_3(e) \text{ in thread}(e')$
3. for all $e'' \in E s.t \exists e \rightarrow e'' \rightarrow e' \notin \sigma' = \sigma''$ then
4. $\sigma'' := \sigma' \cup (e \rightarrow e'')$
5. for each $e \in \text{dangling-reads}(\sigma)$ do
6. if $\exists e \in E s.t (e \rightarrow e') \notin \sigma' \land (e'' \rightarrow e) \notin \sigma' \land \neg \exists e'' \in \Sigma \land \neg \exists (e'' \rightarrow e) \in \Sigma$ then
7. $P_w[e].\text{add}(e')$
8. while $(e_1, e_2)$ not detected or $|\Sigma| \leq \text{search limit}$ do
9. $\sigma'' := \sigma' \cup (e \rightarrow e'')$
10. for each $e \in \text{dangling-reads}(\sigma)$ do
11. $\sigma'' := \sigma'' \cup (e \rightarrow e'') \cup (e \rightarrow e')$
12. $e', e'' \in P_w[e]$ and $e'$ immediately precedes $e''$.
13. $\Sigma := \Sigma \cup \sigma''$

Fig. 9(b) presents the synthesized schedule for our example. Because $R^*_y$ is a dangling read, it identifies the two possible writes and eventually finds that reading the value written by $w^2_y$ at $t_2$ causes the undetected race to be reported.

**Table I:** Experimental results. $\Delta$: tree edit distance, $T_T$: transformation time (in secs).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>$P$</th>
<th>$\Delta$</th>
<th>$T_T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>1.3.151</td>
<td>3713</td>
<td>4.4.152</td>
</tr>
<tr>
<td>ArrayList</td>
<td>1.2</td>
<td>727</td>
<td>1.4.152</td>
</tr>
<tr>
<td>BitSet</td>
<td>1.2</td>
<td>194</td>
<td>1.5.152</td>
</tr>
<tr>
<td>HashMap</td>
<td>1.2</td>
<td>852</td>
<td>1.3</td>
</tr>
<tr>
<td>Hashtable</td>
<td>1.2</td>
<td>3041</td>
<td>1.55</td>
</tr>
<tr>
<td>xpath</td>
<td>2.5.0</td>
<td>2599</td>
<td>2.5.1</td>
</tr>
<tr>
<td>batik</td>
<td>1.1</td>
<td>879</td>
<td>1.5.1</td>
</tr>
<tr>
<td>jigsaw</td>
<td>2.2.5</td>
<td>3794</td>
<td>2.2.6</td>
</tr>
</tbody>
</table>

**IV. Experimental Results**

We have implemented our framework using ROADRUNNER [16] and perform race detection on multithreaded Java programs using two popular dynamic race detectors, DJIT+[33] and FT [15]. We perform the experiments on an Ubuntu-12.04 desktop running on a 3.5 Ghz Intel Core i7 processor with 16GB RAM. We used the following benchmarks for our experiments: h2 is a java-based SQL
Table II: Results specifying test cases, constraints, detected races \( (D) = \) detected races with just mapping \( (\sigma') \) + fixed races + count of races missed due to modifications and subsequently detected \( (Q) \), total runs necessary for detecting all races.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Test cases</th>
<th>Constraints</th>
<th>( D )</th>
<th>( \sigma )</th>
<th>Fixed</th>
<th>( Q )</th>
<th>Schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>200</td>
<td>7128</td>
<td>38</td>
<td>12</td>
<td>41</td>
<td>8</td>
<td>219</td>
</tr>
<tr>
<td>ArrayList</td>
<td>100</td>
<td>82</td>
<td>15</td>
<td>12</td>
<td>2</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>BitSet</td>
<td>100</td>
<td>118</td>
<td>19</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>HashMap</td>
<td>100</td>
<td>80</td>
<td>38</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>HashTable</td>
<td>100</td>
<td>824</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>xalan</td>
<td>200</td>
<td>7706</td>
<td>41</td>
<td>38</td>
<td>0</td>
<td>3</td>
<td>203</td>
</tr>
<tr>
<td>batik</td>
<td>200</td>
<td>2385</td>
<td>38</td>
<td>26</td>
<td>2</td>
<td>10</td>
<td>201</td>
</tr>
<tr>
<td>jigsaw</td>
<td>200</td>
<td>6528</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>203</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Constraints</th>
<th>( D )</th>
<th>( \sigma )</th>
<th>Fixed</th>
<th>( Q )</th>
<th>Schedules</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2</td>
<td>7039</td>
<td>48</td>
<td>3</td>
<td>39</td>
<td>4</td>
<td>211</td>
</tr>
<tr>
<td>ArrayList</td>
<td>90</td>
<td>16</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>BitSet</td>
<td>118</td>
<td>19</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>HashMap</td>
<td>97</td>
<td>39</td>
<td>39</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>HashTable</td>
<td>829</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>xalan</td>
<td>7828</td>
<td>47</td>
<td>43</td>
<td>0</td>
<td>4</td>
<td>204</td>
</tr>
<tr>
<td>batik</td>
<td>2382</td>
<td>38</td>
<td>26</td>
<td>2</td>
<td>10</td>
<td>202</td>
</tr>
<tr>
<td>jigsaw</td>
<td>6498</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>204</td>
</tr>
</tbody>
</table>

A database engine, ArrayList, BitSet, HashMap, HashTable are classes from the JDK, xalan is a document converter, batik is a scalable vector graphic image generator and jigsaw is a web server. The JDK classes are used in the experimental setup of [46] and we use the tests provided by them to execute the programs. For the other benchmarks, we randomly generate multithreaded tests using the methodology employed in [46].

Table I presents the detailed information on the benchmarks including data on the versions and the lines of code. The tree edit distance (\( \Delta \)) to transform the ASTs from one version to another ranges from 34 to 4.6K demonstrating significant changes. Based on manually browsing the code, we observe that the changes include fixes to race conditions using additional synchronizations or deleting memory accesses. The time taken for mapping the lines from one version to the other is also given which is reasonable.

A. Determinism

Table II presents the results regarding the ability of STABLER to deterministically detect all unfixed races. On the initial version, we execute the benchmarks depending on the shown number of test cases and analyze the executions with the two race detectors. For each test execution, a corresponding schedule is generated and logged by the recorder. For example, 200 schedules are generated while running 200 test cases of h2. The number of constraints required for recording is shown and ranges from 80 to 8k approximately for both the detectors. For example, the schedules for h2 has 7128 execution dependency relations when applying DJIT+. The initial set of races (\( D \)) detected using the detectors is also shown. It ranges from 4 to 56 for DJIT+ and ranges from 5 to 48 for FT.

When we simply transform the recorded schedules to \( \sigma' \) using line number mapping and apply the two detectors, we observe that the detected races (under \( \sigma' \)) range from 3 to 38, and 4 to 43 for the two detectors respectively. As expected, not all races are detected with the simple transformation. In h2, we observe that only 7 and 5 races are detected by the two detectors respectively whereas the original version had 56 and 48 races. Subsequently, we applied the algorithms described in Section III-E and detected a few more races (shown under \( Q \)). Across all the benchmarks, for the two race detectors, the races are not detected previously because of blocked schedules (4 and 3), introduction of happens-before relation (17 and 15), masking (1 and 0) and divergent schedules (3 and 4). For the remaining undetected races, we manually verified that the races are indeed fixed in the new version.

The number of schedules to achieve this is also presented. For h2, we require 19 additional schedules beyond the 200 necessary schedules (given 200 test cases) using DJIT+ and require 11 additional schedules using FT. Any of the different cases discussed in Section III may add up to the additional schedules. For example, among the 19 additional schedules generated for h2 using DJIT+, 15 were generated in the process of resolving blocks and four were generated to detect suppressed defects.

B. Overhead

![Fig. 10: Overhead introduced by STABLER. Plain detection time (in secs) shown above bars.](image)

Figure 10 presents the overhead of using STABLER. The maximum incurred slowdown on top of plain race detection is 1.13x and 1.21x for DJIT+ and FT respectively and hence reasonable. The overhead for replay ranges from 1.3x to 1.79x for DJIT+ and 1.2x to 2.29x for FT. The higher overhead is due to two reasons: (a) for every execution index, we need to ensure that the constraints are satisfied, and (b) when a race is not detected, we perform further analysis to modify the schedules appropriately (see Section III-E). Because identifying races on rare schedules can be a difficult task [34], we consider these overheads to be reasonable.

C. User experiments

In Table II, apart from h2, the number of races fixed in the newer version is less. Therefore, we wanted to study the effectiveness of STABLER in the presence of more fixes to races. To achieve this, we conducted user experiments.

308
We now discuss the scenarios where STABLER cannot be employed. Firstly, complex refactorings where mapping the line numbers across versions is imprecise cannot be handled. Secondly, the process of ensuring determinism in certain programs can be hard\cite{43}. If an execution cannot be made deterministic on the same version, our approach will not be applicable. Finally, based on the number/kind of additions (or deletions) leading to divergent schedules, our approach may reach the limit (see Algorithm 7) for searching before reporting the unfixed race.

V. RELATED WORK

To the best of our knowledge, we are the first to propose and address the problem of deterministic race detection across versions. SIMRT \cite{46} addresses the problem of identifying races that are induced due to code changes. In their approach, they focus on efficiently selecting test cases to ensure that the newly introduced data races can be detected in a cost-effective manner. There are a number of techniques that propose an approach for accelerating exploring schedules from one program version to another version \cite{17}, \cite{20}. These approaches consider the issue of test coverage.

Reducing the bugs in multithreaded programs by restricting the number of feasible interleavings has been the focus of recent efforts. Deterministic multithreading (DMT) and stable multithreading (SMT) systems \cite{3}, \cite{5}, \cite{25}, \cite{12} constrain the thread interleavings such that the execution on the same input is deterministic. Unlike these approaches, we address the problem of record and replay on different versions.

Replay techniques are employed to manifest a bug reliably to ease the process of root cause analysis \cite{18}, \cite{30}. CLAP\cite{19} employs an elegant approach of solving the constraints locally to reproduce a concurrency bug that occurred in the field with less overhead. These techniques are targeted towards being able to reproduce a concurrency bug that happen in the field. In contrast, we propose a systematic approach for consistently detecting races before deployment.

There are a number of dynamic analysis techniques to detect concurrency bugs \cite{15}, \cite{37}, \cite{35}. RoadRunner \cite{16} is a dynamic analysis framework which can be used to build these dynamic analysis tools. As discussed earlier, the disadvantage of these tools is that defect detection is dependent on the executed interleaving. We address the problem of non-determinism in the context of race detectors. Our approach can also be extended to lock-set based detectors \cite{37}. Our approach handles issues pertaining to non-determinism in lockset detectors also. Static concurrency bug detectors \cite{44}, \cite{29}, \cite{22} are deterministic but are imprecise \cite{38}.

The approach presented in multi-version software updates \cite{10} runs the two versions in parallel to ensure reliability. Seeker \cite{41} synthesizes method sequences by combining static and dynamic analysis techniques to ensure high coverage. Systematic exploration of the search space using context-bounding \cite{28}, depth bounding \cite{9}, variable and thread bounding \cite{7} attempt to detect bugs in fewer execution runs. We differ from these approaches as our core goal is to ensure detection of already detected races, until they are fixed.

VI. CONCLUSION

We presented the design of a framework, named STABLER, for deterministic dynamic race detection across program versions. We record, transform and replay the schedules intelligently in an attempt to detect all races that are detected previously and are not yet fixed. Modifications to the source including addition and deletion of locks and shared memory accesses are handled by our framework. Our experimental results on many multithreaded Java programs demonstrate the effectiveness of STABLER in achieving the stated goal.