Reduce Dependence for Sound Concurrency Bug Prediction

Shihao Zhu*†, Yuqi Guo*†, Yan Cai*† , Bin Liang‖, Long Zhang‡§, Rui Chen¶, Tingting Yu¶
* *Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China*
†*University of Chinese Academy of Sciences, China* ‖*Renmin University of China, China*
¶*Beijing Sunwise Information Technology Ltd., China* ‡*National Intelligent Voice Innovation Center, Anhui, China*
§*Inspur Cloud Information Technology Co., Ltd., Shandong, China*
{zhush,guoyq,yancai}@ios.ac.cn, liangb@rcu.edu.cn, zlong@ios.ac.cn, {chenrui, yutingting}@sunwiseinfo.com

***Abstract*—Recently, dynamic concurrency bug predictions have kept making notable progress in improving concurrency coverage while ensuring soundness. Most of them rely solely on dynamic information in traces and overlook the static semantics of the program when predicting bugs. To ensure soundness, they assume that any (memory) read can fully affect subsequent program execution via control-flow and data-flow. However, the assumption over-approximates constraints among (memory) writes and reads and hence limits reordering space over thread interleaving, ultimately leading to false negatives. From program semantics, only a subset of reads actually affect their subsequent executions. Therefore, by refining dependencies between reads and subsequent executions based on static program semantics, one can refine the assumption and eliminate unnecessary constraints. This can bring a chance to explore more thread interleaving space and uncover more concurrency bugs. However, refining dependencies can compromise soundness and bring heavy overhead.**

To tackle these challenges, this paper introduces the concept of Necessary Consistent Read Event (NRE) and a hybrid analysis algorithm. NRE refines dependencies between reads and their subsequent events and is used to identify necessary constraints where a read probably affects the execution of its subsequent events. Next, we design an efficient and accurate hybrid analysis algorithm to calculate NREs for each event in the trace. The hybrid analysis algorithm maps events to program SSA instructions and simulates executions based on the original trace. NRE and the algorithm can enhance the capabilities of existing concurrency bug prediction methods at a low cost, regardless of the type of concurrency bug they target. In this paper, we focused on data race and developed NRE and the algorithm as a prototype tool RECONP. We conducted a set of comparative experiments on MySQL with M2 and SEQCHECK. The results show that RECONP can detect 46.9% and 22.4% more data races than M2 and SEQCHECK, respectively. And the hybrid algorithm only accounts for 34% of the total time cost.

***Index Terms*—concurrency bug, prediction, sound, dependence.**

I. INTRODUCTION

Multi-threaded programs can take advantage of hardware, making them crucial in large-scale applications. Nevertheless, the intricacies of multi-threaded program logic present formidable challenges to programming, often leading to the inadvertent introduction of concurrency bugs and posing a significant threat to the quality of concurrent software [1]. Thus, detecting concurrency bugs has become a crucial task in ensuring the quality of concurrent software.

Except for synchronizations, the execution of each thread in a multi-threaded program is independent. This brings that the interleaving orders of threads remain entirely unpredictable, making the detection of concurrency bugs a challenging task due to state explosion [2], [3]. Over decades, researchers have proposed numerous concurrency bug detection methods [4]–[26]. In recent years, dynamic methods that predict concurrency bugs based on existing executions have gained attention for their notable effectiveness, efficiency, accuracy, and scalability. These methods are based on partial order [14]–[20] or constraint solvers [22]–[25], with representative ones being M2, SEQCHECK, TOCCRACE, and DIRK.

Prediction methods predict concurrency bugs by inferring potential executions from existing execution traces. Given a trace, they first identify an event sequence ρ that may directly trigger a concurrency bug or indirectly reveal one. Then they check whether ρ is feasible by inferring potential executions, of which ρ is a sub-sequence. Most of these methods solely depend on information in traces, neglecting the static program semantics. This oversight hinders their ability to comprehend the relations between events, which can impact the feasibility and behavior of subsequent events and thereby influence the feasibility of the inferred executions. For example, the value read by a read event might affect a branch event, which in turn affects whether subsequent events can occur. Consequently, to ensure soundness, i.e., no false positives, these methods have to make an over-approximated assumption: **any (memory) read can fully affect subsequent program execution via control-flow and data-flow**. This assumption introduces over-approximated constraints: the value read by any read event must remain consistent with the original trace, except in specific cases [18]. The imposition of more constraints leads to the exploration of fewer thread interleavings, and as a result, the assumption results in many false negatives.

Notwithstanding the above, it is essential to note that not all read events have an impact on the subsequent events. If we can refine the dependencies between events by considering program static semantics, we can break the aforementioned assumption and eliminate many unnecessary constraints. This

Corresponding author: Yan Cai.

would enable us to explore more thread interleavings and uncover more bugs. However, static analysis is usually unsound and introducing static information into concurrency bug prediction can result in extremely high time overhead (TOCCRACE has 10x the time cost compared to SEQCHECK). Therefore, this task faces two challenges, i.e., (1) **soundness**: how to refine dependencies between events and to distinguish necessary constraints from unnecessary ones without compromising soundness and (2) **efficiency**: how to design algorithms to utilize static information to accurately analyze dependencies between events without introducing much time cost.

In this paper, we propose a novel concept named Necessary Consistent Read Event (NRE) and a hybrid analysis algorithm to address the two challenges, respectively. NRE refines the dependencies between events based on the def-use relations [27] between instructions involved in the execution of the original trace. Given a trace σ and an event e, NRE identifies the read events that should get the same value as those in σ to ensure that e occurs and the behavior of e remains unchanged in inferred executions. The read-from relations derived from this and the synchronization relations between threads constitute the necessary constraints, while other constraints are unnecessary. The hybrid analysis maps events to SSA instructions and simulates the execution of the original trace. During simulation, the dependencies between events are maintained according to the def-use relations, and NREs for each event are computed. The algorithm employs partial simulation to balance efficiency and accuracy.

NRE and the hybrid algorithm can be easily integrated into existing sound prediction methods to detect more concurrency bugs, regardless of the type of concurrency bug they target. For simplicity, this paper focuses on data race detection for detailed discussion and experimental evaluation.

We applied NRE and the hybrid analysis algorithm to a recent work M2, implemented RECONP, and conducted a set of comparative experiments on MySQL. We compared RECONP with M2 and another data race detector SEQCHECK. In the experiment, there are 256 traces collected, with sizes ranging from 62 MB to 39 GB. In effectiveness, RECONP detected 46.9% more races than M2 and 22.4% more than SEQCHECK on average. In efficiency, the hybrid analysis only accounts for 34% of the overall running time; and this cost can drop to approximately 9.4% or even lower if NRE and the hybrid analysis algorithm are applied to TOCCRACE or the constraint-solver-based methods like DIRK [25]. Besides, the memory overhead is 81.6% and 74.5%, respectively, which is also reasonable.

In summary, this paper makes the following contributions:

- A novel concept Necessary Consistent Read Event is proposed to refine the dependencies between events. It can be used to eliminate unnecessary constraints in concurrency bug prediction while maintaining soundness.
- A hybrid analysis algorithm has been developed to analyze dependencies between events and calculate NRE, balancing the accuracy and efficiency.
- We applied NRE and the hybrid algorithm on M2 and developed RECONP. A set of experiments shows the effectiveness and efficiency of RECONP compared with several recent works M2 and SEQCHECK.

In the rest of this paper: Section 2 introduces the preliminaries and the motivation. Section 3 presents NRE and the hybrid algorithm. Section 4 shows the implementation of RECONP. Section 5 shows the evaluation results. Sections 6 and 7 introduce related works and summarize this paper.

II. PRELIMINARIES AND MOTIVATIONS

A. Basic Definitions

Following the prior works [14]–[16], [18], [19], [23], [25], we assume that multi-threaded programs follow the *sequential consistency* memory model. Given a multi-threaded program, concurrency bug prediction methods first execute the program to collect an execution trace. Then they infer potential execution sequences from traces to predict concurrency bugs. Existing prediction methods (including the method proposed in this paper) handle lock constraints in the same way. Therefore, for simplicity, all discussions in this paper will omit the handling of locks. For more details, please refer to [18].

Execution Trace. An (execution) trace σ is an ordered sequence of events, corresponding to an execution of a multi-threaded program execution. A trace consists of three types of events (other events like synchronizations can be modeled similarly):

- Memory event: **write/read**, denoted by $wr(t, x, v)$ / $rd(t, x, v)$, indicating thread t writes/reads a value v to/from a memory location x.
- Lock event: **lock/unlock**, denoted by $acq(t, l)$/ $rel(t, l)$, indicating thread t acquires/releases a lock l.
- Branch event: **branch**, denoted by $br(t, d)$, indicating a conditional branch in thread t chooses the destination basic block named d. The implicit conditional branches due to function calls are also included here.

For ease of presentation, we use $T(e)$ to denote the thread of event e. Given a trace σ, we use $\sigma^e_{t\cdot p}$ and $\sigma^e_{t\cdot s}$ to denote the prefix and the suffix of the projection of σ on thread t that ends and starts at e, respectively. Given an event e, if e is a memory or a lock event, we use $loc(e)$ to denote the memory location or the lock of e; if e is a memory/branch event, we use $val(e)$ to denote the value/destination of e. We also use $op(e)$ to denote the type of e, where $op(e) \in \{rd, wr, acq, rel, br\}$.

Read-from relation. Following the prior works [14]–[16], [18], we use the **read-from** relation to model the read-after-write dependency. Given a trace σ, we say a read event e_r reads from a write event e_w if e_r observes the value written by e_w in σ, represented by $e_w = RF_\sigma(e_r)$, where σ can be omitted. If a read event e_r reads from a write event in the same thread, we call it an **Inner-thread Read Event**; otherwise, we call it a **Cross-thread Read Event**. For example, in the trace shown in Figure 2 (b), we denote the event in line i with e_i. Then we have $e_1 = RF(e_2)$ and $e_4 = RF(e_6)$, where e_2 is an inner-thread read event and e_6 is a cross-thread read event.

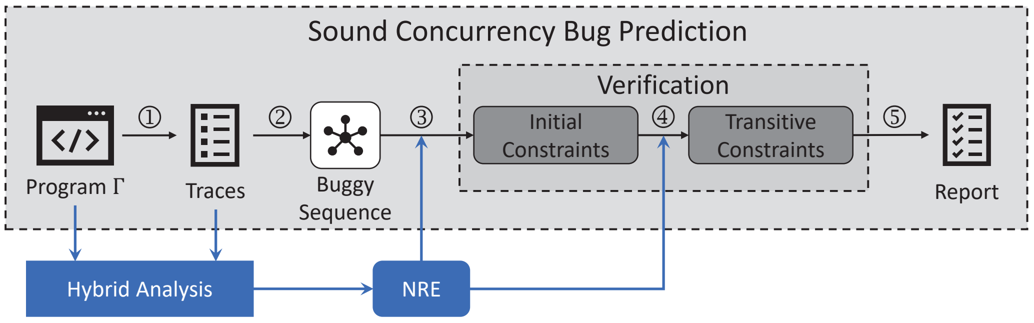


Fig. 1: Sound Concurrency Bug Prediction with NRE.

Data Race. Given two distinct memory events from different threads, if they access the same memory location and at least one of them is a write event, we say they form a **conflict pair**. A (data) race occurs when the events in a conflict pair occur concurrently. Data race is the most common type of concurrency bug and has been extensively studied [1].

Given a sequence of events ρ, we say ρ is feasible if there exists an actual execution that can generate ρ, i.e., ρ can occur.

B. Sound Concurrency Bug Prediction Methods

The gray shaded area in Figure 1 illustrates the workflow of existing sound concurrency bug prediction methods. These methods start by running a program Γ and collecting traces. Then they analyze the traces to identify potential concurrency bugs. For a potential bug, they construct a sequence that can trigger it. For example, given a potential data race (e_1, e_2), the sequence will be $\langle e_1, e_2 \rangle$. Subsequently, these methods employ various algorithms to verify the feasibility of the buggy sequence. Specifically, they define the conditions that need to be met for the sequence to occur using constraints and then check whether these constraints have conflicts.

The verification consists of two steps: (1) extracting the initial set of constraints from the buggy sequence and the trace; and (2) handling transitive constraints based on the initial set. Typically, the initial set includes only the constraints that directly impact the behavior and feasibility of the target events, meaning that their influence is not transmitted through other read-from relations. Furthermore, to satisfy the initial set of constraints, it is necessary to ensure that all events affecting these constraints can also occur and their behavior remains consistent with what is observed in the collected traces. Transitive constraints refer to the constraints that arise due to the above requirements. If no conflicts are found in verification, the potential bug can indeed occur. Following the verification, the prediction methods generate an inferred execution based on the constraints and produce a bug report.

Sound concurrency bug predictions can be broadly categorized into two types based on the verification algorithms they employ: partial-order-based methods and constraint-solver-based methods. **Partial-order-based methods** use partial orders to describe constraints and perform closure on the initial partial order set to handle transitive partial orders. These are excellent in bug prediction capability, efficiency and scalability. The state-of-the-art works include M2 [16], SEQCHECK [18], and TOCCRACE [19]. On the other hand, **Constraint-solver-based methods** employ formal statements to describe constraints and use solvers to resolve them, addressing transitive constraints during the solving process. These have excellent bug prediction capability. However, they suffer from low efficiency because of state explosion, especially on large traces. The state-of-the-art one is DIRK.

As previously mentioned, existing sound prediction methods necessitate that the values obtained by read events in the inferred executions must align with those in the original trace. Partial-order-based methods ensure this by maintaining the read-from relations, while constraint-solver-based methods use solvers to guarantee this. In other words, the former must preserve the same read-from relations as the original trace, while the latter does not. This distinction leads to some variations in how they handle transitive constraints. Theoretically, the prediction capability of the latter is stronger than that of the former, but due to the windowing techniques used by the latter, this is not always the practice case.

C. Motivations

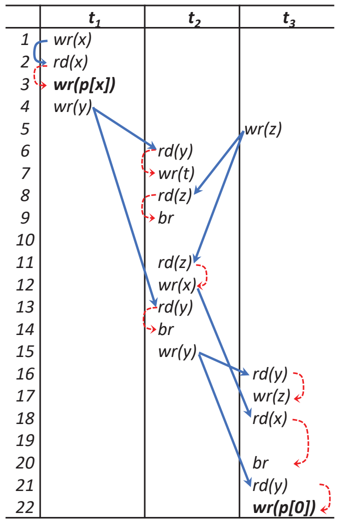
TABLE I: Constraints of Each Tools

Tool	Initial Constraints	Transitive Constraints
M2	$e_1 = RF(e_2), e_{15} = RF(e_{16})$ $e_{12} = RF(e_{18}), e_{15} = RF(e_{21})$	$e_4 = RF(e_6), e_5 = RF(e_8)$ $e_5 = RF(e_{11}), e_4 = RF(e_{13})$
SEQCHECK	$e_{15} = RF(e_{16}), e_{12} = RF(e_{18})$	$e_{11} = RF(e_{12}), e_6 = RF(e_9)$ $e_5 = RF(e_{11}), e_4 = RF(e_{13})$
TOCCRACE	$e_{15} = RF(e_{16}), e_{12} = RF(e_{18})$	$e_{11} = RF(e_{12}), e_6 = RF(e_9)$ $e_5 = RF(e_{11}), e_4 = RF(e_{13})$
DIRK	$e_1 = RF(e_2), e_{15} = RF(e_{16})$ $e_{12} = RF(e_{18}), e_{15} = RF(e_{21})$	$e_4 = RF(e_6), e_5 = RF(e_8)$ $e_5 = RF(e_{11}), e_4 = RF(e_{13})$
RECONP	$e_1 = RF(e_2), e_{12} = RF(e_{18})$	$e_5 = RF(e_8), e_5 = RF(e_{11})$

As previously stated, existing sound concurrency bug prediction methods all assume that **any (memory) read can fully affect subsequent program execution via control-flow and data-flow**. This assumption introduces many unnecessary constraints, severely limiting the bug prediction capabilities. Let's illustrate this with an example. Figure 2 shows a program with three threads (a) and a corresponding trace σ (b). The blue edges represent the read-from relations. In line 20 of Figure 2 (a), $l1$ and $l2$ represent two different basic blocks. We denote

	t_1	t_2	t_3	SSA of t_3
1	x = 0			
2	**p[x] = 0**			
3				
4	y = 1			
5			z = 2	
6		t = y		
7				
8		if(z == 0)		
9		~~z = t;~~		
10				
11		x = z;		
12				
13		if(y > 0)		
14		printf();		
15		y = 3		
16			z = y	%1 = load y
17				store %1, z
18			if(x == 2)	%2 = load x
19			**p[0] = y;**	%3 = cmp %2, 2
20				br %3, l1, l2
21				%4 = load y
22				%5 = store %4, p, 0

(a) A program



(b) A trace

	t_1	t_2	t_3
1	x = 0		
2	a = x		
3			
4			z = 2
5		t = y	
6			
7		if(z == 0)	
8		~~z = t;~~	
9			
10		x = z;	
11			z = y
12			
13			if(x == 2)
14			**p[0] = y;**
15			
16			
17			
18	**p[a] = 0**		
19	y = 1		
20		if(y > 0)	
21		printf();	
22		y = 3	

(c) An execution with a race

Fig. 2: An example with three threads.
The blue and red arrows represent the read-from relations and the def-use relations, respectively.

the event in line i in Figure 2 (b) with e_i. Consider the potential race (e_3, e_{22}), Table I shows the constraints generated by M2, SeqCheck, ToccRace and Dirk:

- **M2** requires that all read events, except for those in the constructed sequence, should follow the read-from relations in the original trace. Therefore, the initial constraints includes the read-from relations of e_2, e_{16}, e_{18} and e_{21}, where $e_{15} = RF(e_{16})$ requires e_{15} to occur before e_{16}. Additionally, due to the transitive constraint $e_4 = RF(e_6)$ caused by e_{15}, M2 requires e_4 to occur before e_6. According to the program semantics (or the program orders [18]),M2 infers that e_3 must occur before e_{22}, and determines that (e_3, e_{22}) is not a race.
- **SeqCheck** finds that read events without branch events in the suffix will not affect the feasibility of subsequent events. Therefore, given a read event e, if there is no branch event in the suffix of e, SeqCheck allows e to violate the corresponding read-from relation in the original trace. Because of the branch event e_{20} and e_{14}, SeqCheck maintains the constraints $e_{15} = RF(e_{16})$ and $e_4 = RF(e_6)$. This makes SeqCheck get the same conclusion as M2.
- **ToccRace** leverages the static information of the program to ascertain that, even control-flow changes, certain events will invariably occur and their behavior will remain unchanged. ToccRace proposed **Equivalent Pair** according to this fact. In this example, ToccRace finds that e_{18} affects the destination of the branch event e_{20} and further affects whether e_{22} can occur. Therefore, it cannot construct an equivalent pair for (e_3, e_{22}), and employ the algorithm of SeqCheck to detect this race. Eventually, it gets the same conclusion as SeqCheck.
- **Dirk** requires all read events to get the same value as the given trace. Therefore, e_6 must read from e_4 to get value 1 and e_{16} must read from e_{15} to get value 3. As a result, Dirk maintains the same constraints as M2, and determines that (e_3, e_{22}) does not form a race.

Despite the above results, (e_3, e_{22}) does form a race. This false negative is caused by the aforementioned assumption. Ignoring the assumption, from the source code, we can know that not all read events will impact e_3 and e_{22}. Specifically, in thread t_3, only the value of x obtained by e_{18} affects the execution result of the branch event e_{20}, thus affecting the feasibility of e_{22}. Therefore, to ensure that e_{22} occurs and its target memory location does not change, we only need to ensure that the value obtained by e_{18} is consistent with the given trace. The read-from relations corresponding to e_{16} and e_{21} can be safely eliminated. Similarly, e_6 does not affect e_{18}, the read-from relation corresponding to e_6 can be also eliminated. Consequently, we can infer the execution shown in Figure 2(c), and detect the race (e_3, e_{22}) (a is a temporary variable, introduced for the sake of clarity).

If we can refine the dependencies between events considering static program information, we can break the aforementioned assumption, and eliminate unnecessary constraints to detect more bugs, without losing soundness. Inspired by this, we propose Necessary Consistent Read Event and a hybrid analysis algorithm.

III. Our Approach

A. Framework

Figure 1 shows how NRE and the hybrid analysis algorithm are incorporated into the workflow of existing sound prediction methods. Given a program Γ and a corresponding trace, we first run the hybrid analysis algorithm to calculate the NREs for all events based on the program's static semantics and the dynamic information from the trace. We then proceed with the workflow of the existing methods.

In the steps of computing the initial set of constraints and handling transitive constraints, NRE is used to eliminate unnecessary constraints. The read-from relations extracted from the NREs of the events in the buggy sequence and synchronization relations form the initial set of constraints, while other constraints are removed from the initial set. Starting from the events involved in the initial set of constraints, the read-from relations extracted from the NREs of these events and synchronization relations further form the transitive constraints, with other constraints similarly being removed. This process eliminates many unnecessary constraints, allowing for increased exploration of thread interleaving space and thereby enhancing the concurrency bug prediction capabilities of existing methods.

In the subsequent subsections, we provide a detailed introduction to NRE and the hybrid analysis algorithm. Our discussions are based on a given program Γ and a trace σ. The hybrid analysis is performed on static single assignment (SSA) instructions, and the corresponding instruction of event e is denoted as $I(e)$. It should be noted that an event corresponds to an instruction, but an instruction can correspond to multiple events. For ease of understanding, we expand the discussions with the examples in Figure 2, where the blue arrows represent the read-from relations and the red arrows represent the def-use relations.

B. Necessary Consistent Read Event

To clearly refine the dependencies between events, we first define DRE (Direct Dependent Read Event) to describe the direct data dependencies between events. Based on the direct data dependencies described by DRE, we further define NRE (Necessary Consistent Read Event) to refine all possible dependencies between events, including data dependencies and control dependencies.

Before detailing DRE and NRE, we emphasize that DRE and NRE only consider dependencies involved in the execution path corresponding to the trace, and do not account for dependencies outside of this execution path. This allows for a more precise description of data dependencies between events. For example, in t_2 in Figure 2 (a), considering all possible execution paths of this thread, we can conclude that the value written to x at line 11 might be influenced by the value of y read at line 6. If we map this dependency to events in Figure 2 (b), we would infer that e_{12} is data-dependent on e_6. However, in this trace, e_{12} does not have any dependency on e_6. By considering only the dependencies involved in the execution path corresponding to the trace, we can conclude consistent with this fact.

Given an event e in a trace σ, its DREs are in the same thread as e and refer to the read events that affect the behavior of e through def-use relations [27] between instructions in σ. The formal definition of DRE is as follows:

Direct Dependent Read Event (DRE). Given an event $e \in \sigma_t$ and a read event e_r in $\sigma^e_{t\cdot p}$, where $e \neq e_r$, we use ε to represent the sequence of SSA instructions that were executed between e_r and e in thread t. If there is a SSA instruction sequence $\alpha = \langle I_0, I_1, I_2, ..., I_n \rangle$, that satisfies the following conditions, then we say e_r is a direct dependent read event of e in trace σ:

1) α is a sub-sequence of ε.
2) $I_0 = I(e_r) \wedge I_n = I(e)$.
3) $\forall i \in [0, n)$, I_{i+1} uses the value defined by I_i, i.e., I_i and I_{i+1} are in a def-use chain.

The first condition ensures that DRE considers only the def-use relations on the execution path corresponding to the trace, while the second and third conditions guarantee the completeness of the def-use chain. In Figure 2 (b), the DRE event set for e_{19} is $\{e_{18}\}$.

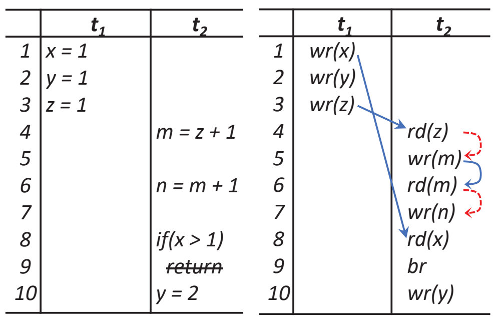
Based on DRE, we further define NRE, which takes into account both data dependencies and control dependencies. Specifically, given a read event e_r and another event e in the same thread that occurs after e_r, if e_r affects the behavior of e through def-use relations, or affects the control-flow or critical sections before e, then e_r is an NRE of e. For example, in Figure 2 (b), e_{18} is a DRE of e_{19}, which means e_{18} affects the control-flow before e_{22}; therefore, e_{18} is an NRE of e_{22}. The formal definition is as follows:

Necessary Consistent Read Event (NRE). Given an event $e \in \sigma_t$ and a read event e_r in $\sigma^e_{t\cdot p}$, where $e \neq e_r$, if e_r satisfies one of the following conditions, then we say e_r is a necessary consistent read event of e:

1) e_r is a direct dependent read event of e in σ.
2) $\exists e' \in \sigma^e_{t\cdot p} \mid e' \neq e \wedge op(e') \in \{br, acq, rel\}$, e_r is a direct dependent read event of e' in σ.
3) $\exists e' \in \sigma^e_{t\cdot p} \mid e' \neq e \wedge op(e') = wr$, e_r is a direct dependent read event of e' in σ, and the data dependency lies on $loc(e')$.

We use $NRE(e)$ to represent the set of the NREs of e. The first condition states that e_r directly affects e through def-use relations; if the value obtained by e_r changes, the behavior of e will also change. The second condition states that if e_r affects the subsequent control-flow or critical sections, then any change in the value obtained by e_r may prevent the subsequent events from occurring normally. Lastly, the third condition states that if e_r affects the target memory location of a write event e', then any change in the value obtained by e_r may make the subsequent behavior of the program unpredictable. For example, the target memory location of e' may become illegal and cause a segment fault, which would prevent e from occurring. In conclusion, to ensure that e occurs and the behavior of e does not change, the NREs of e should get the same value as σ.

It is important to emphasize that $NRE(e)$ is a superset, rather than an exact set, of the read events that should read the same value with the original trace to ensure that e occurs with unchanged behavior. Accurately computing dependencies between events while allowing changes in control-flow, critical sections, and memory addresses of write events is an NP problem. Therefore, for ease of algorithm design, NRE adopts a conservative assumption: any change in control flow, critical sections, and memory addresses of write events may affect



	t_1	t_2
1	x = 1	
2	y = 1	
3	z = 1	
4		m = z + 1
5		
6		n = m + 1
7		
8		if(x > 1)
9		~~return~~
10		y = 2

(a) A Program

	t_1	t_2
1	wr(x)	
2	wr(y)	
3	wr(z)	
4		rd(z)
5		wr(m)
6		rd(m)
7		wr(n)
8		rd(x)
9		br
10		wr(y)

(b) A trace

Fig. 3: A false negative example caused by partial simulation.

the feasibility and behavior of subsequent events. The conservative assumption causes NRE(e) to be a superset, which may ultimately result in false negatives in bug prediction. Despite this, the assumption of NRE is significantly more relaxed than that of existing methods, enabling it to effectively filter out unnecessary constraints and enhance the capability of concurrency bug prediction.

C. Hybrid Analysis

Based on the definition of NRE, we can calculate the NREs for all events in the trace. To do this, it is essential to analyze the dependencies between events according to the static semantics of the program. An intuitive approach is to obtain the dependencies between instructions through static analysis and then map these dependencies onto the events. However, as mentioned in Section III-B, this approach will introduce dependencies outside the execution path corresponding to the trace, significantly reducing the accuracy of NRE calculation. On the other hand, relying solely on the dynamic information in traces to analyze dependencies can lead to inaccuracy due to the lack of static information (e.g., def-use relations). Therefore, to calculate NREs accurately, it is necessary to perform a hybrid analysis that integrates both dynamic and static information.

The simplest method to perform a hybrid analysis is to simulate the execution of the trace by following the order in which the instructions are executed. During this simulation, dependencies between events are maintained according to the def-use relations between instructions, and NREs are calculated accordingly. Temporary variables can be represented with the SSA instructions.

1) Full Simulation vs Partial Simulation:

Since dependencies between events often traverse functions, the analysis should be inter-procedural. Fully simulating the program execution process instruction-by-instruction according to the trace can ensure the accuracy of calculating the NREs. However, this approach is time-intensive and memory-intensive, making it impractical for concurrency bug prediction. Therefore, we adopt a partial simulation strategy. And the main challenge comes to balancing accuracy and efficiency.

Compared with inner-thread read events, cross-thread read events are more critical for ensuring program correctness, as they can affect the synchronizations and data transfer between threads. Based on this, we adopt a heuristic strategy for hybrid analysis: focus on cross-thread read events. Specifically, we assume that all inner-thread read events are the NREs of all subsequent events, and then for each event e, we check which cross-thread read event is an NRE of e. This strategy can significantly improve efficiency since cross-thread read events constitute only a small portion of all read events.

Based on the above heuristic strategy, to further improve the analysis efficiency, the simulation starts from the first cross-thread read event instead of the first event. During the simulation, several def-use dependency chains are maintained for each cross-thread read event. These chains record temporary variables (represented with SSA instructions) that have data dependencies on the read event. Depending on the type of the event e being processed, different measures will be taken:

- e is a read event: for any cross-thread read event e_r, if $I(e)$ is appended to a dependency chain starting from e_r, then e_r is considered as an NRE of e.
- e is a branch event or a lock event: for any cross-thread read event e_r, if $I(e)$ is appended to a dependency chain starting from e_r, then all dependency chains of e_r are erased, and e_r is considered as an NRE of all subsequent events of e.
- e is a write event: for any cross-thread read event e_r, if $I(e)$ is appended to a dependency chain starting from e_r, and $I(loc(e))$ is in the dependency chain, then all dependency chains of e_r are erased and e_r is considered as an NRE of all subsequent events of e.

The above three situations correspond to the three conditions in the definition of NRE.

During the simulation, for efficiency, if the dependency chains of all cross-thread read events are erased, we can move on to the next cross-thread read event and restart the simulation. However, this may lead to incomplete stack recording. Hence, to guarantee soundness of concurrency bug detection, if a return instruction is encountered when the stack is empty, we will clear all dependency chains and consider any corresponding cross-thread read events as an NRE of all subsequent events.

By utilizing partial simulation and the strategies mentioned above, the efficiency of hybrid analysis can be significantly improved. However, this comes at the cost of sacrificing some accuracy, meaning that the NRE set calculated will be a superset of the actual one. As a result, some unnecessary constraints will not be recognized, which may eventually result in false negatives when detecting concurrency bugs. For example, consider the race (e_2, e_{10}) in Figure 3. The partial simulation algorithm identifies the inner-thread read event e_6 as an NRE of e_{10}. After handling transitive dependencies, $e_3 = RF(e_4)$ would be identified as a necessary constraint, implying that e_2 must occur before e_{10}, which results in a false positive. Despite this fact, it is important to note that (1) partial simulation will not compromise soundness, and (2)

	source code	SSA	trace σ	dep of σ_1
1	F(x, y){	F(%0, %1)	I_8	$\langle I_8 \rangle$
2	… return 0;	… return 0;	I_9	$\langle I_8 \rangle$
3	return x*2 + F(y, x);	%2 = %0 * 2;	I_{10}	$\langle I_8 \rangle$
4		call F(%1, %0)	I_3	$\langle I_8, I_3 \rangle$
5		%3 = %2 + %ret	I_4	$\langle I_8, I_3 \rangle$
6	}	return %3	I_3	$\langle I_8 \rangle$
7			I_4	$\langle I_8 \rangle$
8	c = F(a, b)	%4 = rd(a)	I_2	$\langle I_8 \rangle$
9		%5 = rd(b)	I_5	$\langle I_8 \rangle$
10		call F(%4, %5)	I_6	$\langle I_8 \rangle$
11		store %ret, c	I_5	$\langle I_8 \rangle$
12			I_6	$\langle I_8 \rangle$
13			I_{11}	$\langle I_8 \rangle$

Fig. 4: An example about recursive call.

the set of constraints extracted from the NREs is still a subset of that extracted by existing sound methods, thereby exposing more concurrency bugs.

2) Recursive Function Call:

As previously mentioned, temporary variables are represented with SSA instructions. However, recursive function calls can lead to a situation where one instruction, say I, corresponds to multiple temporary variables. Using I to represent the temporary variable corresponding to the last execution of I can result in inaccuracy in NRE calculations.

Consider the program in Figure 4 with a recursive call. Assuming that code in line 2 has no data dependencies on x and y, the temporary variable $\%ret$ represents the return value of the last function call. One possible execution of this program is shown as trace σ, where Func returns at depth 3. We use σ_i and I_i to denote the event and instruction in line i, respectively. The def-use chain starting from event σ_1 is shown in the last column. In the trace, when I_3 executes for the first time, the value of $\%0$ comes from event σ_1. Therefore, I_3 has a data dependency on σ_1, and I_3 is appended to the end of the dependency chain. When I_3 executes for the second time, the latest value of $\%0$ comes from event σ_2. As a result, I_3 has no data dependency on σ_1, and I_3 is removed from the dependency chain. When σ_{11} occurs, I_3 is not in the dependency chain, therefore, I_5 is not appended to the dependency chain. Since the corresponding instruction of $\%ret$ in I_{11} is I_5, σ_1 is not recognized as an NRE of σ_{13}. However, the truth is that σ_1 is an NRE of σ_{13} according to the definition of NRE.

To avoid the above inaccuracy during NRE calculation, it is necessary to differentiate between temporary variables that are represented by the same instruction. This can be achieved by saving a copy of the dependency chains when a recursive function call is encountered, and restoring it when the function returns. For instance, in Figure 4, when $Func$ is called for the second time at σ_5, we need to back up the dependency chain of σ_1 and restore it when the function returns at σ_{10}. However, this will result in additional overhead.

Alternatively, when a recursive function call is encountered, we can choose to abort the simulation and treat any cross-thread read event that has been simulated as an NRE of all subsequent events. Similar to partial simulation, this may result in a loss in accuracy, but will not compromise the soundness of concurrency bug detection.

Algorithm 1: Hybrid Algorithm

```
   /* dep[e_r]: Given cross-thread read event e_r,
      record the data dependency chains of e_r     */
   /* stack: Record the function call stack        */
   /* emptyStack: Indicates whether a function
      returns with stack is empty                  */
1  Function Hybrid($\Gamma, \sigma, CRE$):
2      for $t \leftarrow 1$ to $T_\sigma$ do
3          for $e \in IRE[t]$ do
4              setSuffixNRE($t, e, e$);
5          $e \leftarrow$ first event in $CRE[t]$
6          initDepChain($e$)
7          while $e \in \mathcal{E}_{\sigma_t}$ do
8              while $I \neq I(e) \wedge !emptyStack$ do
9                  handleInst($dep, stack, e, I, emptyStack$)
10             if $emptyStack$ then
11                 for $e_r \in dep$ do
12                     if $\neg dep[e_r].empty$ then
13                         setSuffixNRE($t, e_r, e$)
14                 $e \leftarrow$ the first event after $e$ in $CRE[t]$
15                 initDepChain($e$)
16             else
17                 handleEvent(dep, NRE, e)
18                 clearAndUpdate($t, e$)
19     return $NRE$
```

Since most applications have few recursive function calls, both of these solutions are feasible. Tricks on accuracy and efficiency can be made based on the actual situation. To maximize the constraint filtering effectiveness of NRE and enhance concurrency bug prediction capabilities as much as possible, we choose the first one in this paper.

3) Hybrid Analysis Algorithm:

Based on partial simulation, we design a hybrid analysis algorithm to calculate the NREs. This algorithm, outlined in Algorithm 1, operates on the program Γ and a trace σ. The set $IRE[t]/CRE[t]$ includes all inner-thread/cross-thread read events in thread t in the order in which they occur. $NRE[e]$ stores all NREs of e. Given a cross-thread read event e_r, $dep[e_r]$ records the data dependency chains starting from e_r. The variable $stack$ records the function call stack information.

The function $Hybrid$ processes all threads sequentially. For each thread t, it first sets all inner-thread read events to be the NREs of all subsequent events (lines $3-4$). When $setSuffixNRE$ is called with parameters as $\{t, e_r, e\}$, it sets e_r as an NRE of all events that occur after e in thread t. Then, for each event e in thread t, $Hybrid$ identifies which cross-thread read event is an NRE of e. It starts the analysis from the first cross-thread read event in line 5 and traverses suffix events in order (lines $7-18$). The function $initDepChain$ is used to initialize the dependency chains and clear the stack. When encountering a new event e, $Hybrid$ first handles all instructions executed between e and its previous event (lines $8-9$). The function $handleInst$ updates the dependency chains and $stack$ according to the def-use relations. Additionally, it backs up or restores the dependency chains when a function is called or returns. Whenever a function returns while $stack$ is empty, $Hybrid$ sets all cross-thread read events

TABLE II: Hybrid analysis on thread t_3 in Figure 2

Steps	**e**	**I**	dep	NRE
0	e_{16}	I_{16}	$e_{16} : \langle I_{16} \rangle$	/
1	e_{17}	I_{17}	$e_{16} : \langle I_{16}, I_{17} \rangle$	$e_{17} : \{e_{16}\}$
2	e_{18}	I_{18}	$e_{16} : \langle I_{16}, I_{17} \rangle$; $e_{18} : \langle I_{18} \rangle$	$e_{17} : \{e_{16}\}$
3	e_{20}	I_{19}	$e_{16} : \langle I_{16}, I_{17} \rangle$; $e_{18} : \langle I_{18}, I_{19} \rangle$	$e_{17} : \{e_{16}\}$
4	e_{20}	I_{20}	$e_{16} : \langle I_{16}, I_{17} \rangle$	$e_{17} : \{e_{16}\}$ $e_{20} : \{e_{18}\}$ $e_{21} : \{e_{18}\}$ $e_{22} : \{e_{18}\}$
5	e_{21}	I_{21}	$e_{16} : \langle I_{16}, I_{17} \rangle$; $e_{21} : \langle I_{21} \rangle$	$e_{17} : \{e_{16}\}$ $e_{20} : \{e_{18}\}$ $e_{21} : \{e_{18}\}$ $e_{22} : \{e_{18}\}$
6	e_{22}	I_{22}	$e_{16} : \langle I_{16}, I_{17} \rangle$; $e_{21} : \langle I_{21}, I_{22} \rangle$	$e_{17} : \{e_{16}\}$ $e_{20} : \{e_{18}\}$ $e_{21} : \{e_{18}\}$ $e_{22} : \{e_{18}, e_{21}\}$

that have data dependency chains in dep to be the NREs of all subsequent events (lines $11-13$). Then, it restarts the analysis from the next cross-thread read event (lines $14-15$). After that, $Hybrid$ handles e to update NRE according to what has been stated in partial simulation (line 17). The data dependency chains are also updated. Line 18 checks whether all the dependency chains are erased, if so, it moves on to the next cross-thread read event and restarts the simulation. Once the analysis finishes, $Hybrid$ returns NRE.

For ease of understanding, we illustrate the hybrid analysis algorithm with the example in Figure 2 (a). Take thread t_3 as an example, the calculation process is shown in Table II. The second and third columns show the value of e and I before line 8 in Algorithm 1 is executed, and the last two columns show the data in dep and NRE after line 18 is executed.

$Hybrid$ starts the analysis from the first cross-thread read event e_{16} and sets $e = e_{17}, I = I_{17}$ initially. Since $I = I(e) \wedge op(e) = wr$, $Hybrid$ handles this according to the third case described in partial simulation. Here $Hybrid$ finds that e_{17} uses the temporary variable defined by I_{16}. As I_{16} is in the data dependency chain of e_{16}, $Hybrid$ sets e_{16} to be an NRE of e_{17}. Then, $Hybrid$ updates e and I, and continues the analysis. In step 2, $Hybrid$ encounters a new cross-thread read event e_{18}, therefore, a new dependency chain is initialized. When it comes to Step 4, $Hybrid$ finds that the condition of branch event e_{20} is defined by I_{19}, and I_{19} is in the dependency chain of e_{18}. Therefore, $Hybrid$ clears the dependency chains starting from e_{18} and sets e_{18} to be an NRE of all events after e_{20} (including e_{20}). When $Hybrid$ finishes, we can get the NRE shown in Table II.

4) Algorithm Analyses:

We use n to denote the number of events in trace σ, and m to denote the number of instructions in Γ. Algorithm 1 first traverses all inner-thread read events in $O(n)$ and calls $setSuffixNRE$ for each event. The function $setSuffixNRE$ can finish in a constant time by setting a boundary for the read event, where the read event is an NRE of all the events after the boundary. Then Algorithm 1 traverses all cross-thread read events in $O(n)$. Since the trace records all memory, lock, and branch events, the number of instructions executed between two events can be assumed as a constant. For each instruction, $handleInst$ may operate on dep and $stack$. It first checks whether a function call is recursive in $O(\log m)$ (the number of functions is at most m) and updates $stack$ in $O(1)$. Then it traverses all cross-thread read events in dep in $O(n)$. And for each read event, the dependency chains consist of at most m instructions, therefore, searching and updating the dependency chains costs $O(\log m)$. In summary, the time complexity of $handleInst$ is $O(n) \times O(\log m) + O(\log m) + O(1)$ i.e., $O(n \times \log m)$. Similarly, the code in lines $10 - 18$ searches and updates dep in $O(n \times \log m)$, as well as updates NRE in $O(\log n)$. Finally, we can conclude that the hybrid analysis algorithm has a polynomial time complexity of $O(n^2 \times \log m)$.

IV. Application on M2

In this section, we illustrate the application of NRE and the hybrid analysis algorithm by presenting the application on the partial-order-based method M2, which is designed to predict data race. Given a program and a corresponding trace, we first invoke Algorithm 1 to compute the NREs for all events and then perform the workflow of M2. Since the identification of potential races and the construction of buggy sequences are the same before and after applying NRE, we focus primarily on verifying the feasibility of buggy sequences, specifically the computation of the initial set of constraints and the handling of transitive constraints.

The verification algorithm of M2 is shown by the black part in Algorithm 2. It involves reasoning on a directed graph G, where nodes represent events and edges represent partial orders. If there is a path from e_1 to e_2 in G, it means that e_1 should occur before e_2, i.e., $e_1 \prec_G e_2$. M2 computes the initial constraint set and handles transitive constraints in lines $8-10$. Specifically, M2 takes all read-from relations in the original trace as the initial constraint set, inserts the corresponding partial orders into G in line 18, and then performs a closure on the graph in lines $19-20$ to handle transitive constraints. The buggy sequence is deemed feasible if and only if the graph G has no cycles when the algorithm finishes.

To apply NRE to M2, we add the blue part in Algorithm 2. It first collects the NREs of all events in the buggy sequence and stores them into $CNRE$ (lines $2-6$). The read-from relations corresponding to this set and synchronization relations constitute the initial set of constraints. Then, Algorithm 2 follows the verification workflow of M2 and eliminates unnecessary constraints based on $CNRE$ in this progress. Specifically, a read-from relation is considered a necessary constraint and its corresponding partial order is inserted into G only if the involved read event is in $CNRE$. Besides, whenever a read-from relation is considered a necessary constraint, the NREs of the involved write event are inserted into $CNRE$ to handle transitive constraints (line 9). With the help of $CNRE$, Algorithm 2 excludes many unnecessary constraints, thus detecting more concurrency bugs.

One thing worth noting is that triggering concurrency bugs does not always require all events in the buggy sequence to

Algorithm 2: Bug-triggering Sequence Verification

```
1  Function Verify($\sigma, \rho, NRE$):
2      $CNRE \leftarrow \emptyset$
3      for $e \in \rho$ do
4          for $e_r \in NRE[e]$ do
5              if $Necessary(e_r)$ then
6                  $CNRE \leftarrow CNRE \cup e_r$
7      initialize graph $G$
8      foreach read event $e_r \in G| \ e_r \in CNRE$ do
9          $CNRE \leftarrow CNRE \cup NRE[RF(e_r)]$
10         InsertAndClose ($G, RF(e_r), e_r, CNRE$)
11     report FEASIBLE
12 Function InsertAndClose($G, e_1, e_2, CNRE$):
13     q$\leftarrow \emptyset$
14     q.push($\langle e_1, e_2 \rangle$)
15     while $\neg q.empty()$ do
16         $\langle e_a, e_b \rangle \leftarrow$ q.pop()
17         if $e_b \prec_G e_a$ then report INFEASIBLE
18         G.insert($\langle e_a, e_b \rangle$)
19         q.push(ObsClosure ($G, e_a, e_b, CNRE$))
20         q.push(LockClosure ($G, e_a, e_b, CNRE$))
21 Function ObsClosure($G, e_a, e_b, CNRE$):
22     $P \leftarrow \emptyset$
23     foreach read event $r \in G| \ r \in CNRE$ do
24         $w \leftarrow RF(r)$
25         if $\exists w' \in G \mid loc(w) = loc(w') \wedge w' \prec_G e_a \wedge e_b \prec_G r$
           then
26             $P.insert(\langle w', w \rangle)$
27         if $\exists w' \in G \mid loc(w) = loc(w') \wedge w \prec_G e_a \wedge e_b \prec_G w'$
           then
28             $P.insert(\langle r, w' \rangle)$
29     return $P$
30 Function LockClosure($\sigma, Q, e_a, e_b, CNRE$):
31     $P \leftarrow \emptyset$
32     foreach lock event $e \in G$ do
33         if $\exists$ unlock event
           $e' \in G \mid loc(e) = loc(e') \wedge e \prec_G e_a \wedge e_b \prec_G e' \wedge e'$ is
           not the unlock event of $e$ then
34             $rel \leftarrow$ the unlock event of $e$
35             $acq \leftarrow$ the lock event of $e'$
36             $P.insert(\langle rel, acq \rangle)$
37     return $P$
```

maintain the same behavior as the original trace. For instance, to trigger a data race on a write event e_w, we must ensure that e_w can occur and its target memory location remains unchanged. However, the value written can differ from the original one. The function $Necessary$ in line 5 is used to handle this.

In Figure 2, to detect the race (e_3, e_{22}), the buggy sequence will require e_3 and e_{22} to occur consecutively, i.e., no other events occur between them. According to the NREs of e_3 and e_{22}, Algorithm 2 gets $CNRE = \{e_2, e_{18}\}$ in lines $2-6$. In the subsequent process, four read-from constraints are identified as necessary constraints: $\{e_1 = RF(e_2), e_5 = RF(e_8), e_5 = RF(e_{11}), e_{12} = RF(e_{18})\}$, as shown in Table I. Compared to M2, four read-from constraints are excluded: $\{e_4 = RF(e_6), e_4 = RF(e_{13}), e_{15} = RF(e_{16}), e_{15} = RF(e_{21})\}$. As a result, when the algorithm finishes, there will be no cycles in the graph, indicating a successful detection of the race.

V. Evaluation

A. Evaluation Method

We choose to evaluate NRE and the hybrid analysis algorithm based on partial-order-based methods, which are more balanced in terms of effectiveness, efficiency, and scalability than constraint-solver-based methods. The state-of-the-art partial-order-based tools include M2, SeqCheck, and ToccRace. All of them have polynomial time complexity. In fact, SeqCheck can be considered the most efficient existing sound prediction method, as it has demonstrated performance in practice that far exceeds linear complexity algorithms [18].

The working principles and core ideas of these three have already been introduced with examples in Section II-B and II-C. SeqCheck is essentially optimized for special situations based on M2, and applying NRE to M2 and SeqCheck will result in the same tool. ToccRace, on the other hand, tolerates control-flow changes to a certain extent by introducing equivalent pairs. Applying NRE to ToccRace can achieve better effectiveness than applying NRE to M2 or SeqCheck, but it requires major adjustments to the concept of equivalent pair and algorithms. Moreover, ToccRace has a much higher overhead than M2 and SeqCheck on large traces, which can result in significant time and computational resource consumption when conducting large-scale experiments. Taking into account the above factors, we decided to implement the tool ReconP by applying NRE to M2, and compared ReconP with M2 and SeqCheck on race detection.

B. Benchmarks and Setup

We implemented ReconP based on LLVM. Both M2 and SeqCheck are provided by the authors of SeqCheck [18] for testing C/C++ programs, where M2 has been optimized to have almost the same efficiency as SeqCheck. ReconP is a dynamic tool plus an offline analysis. Like many other tools (e.g., ThreadSanitizer in LLVM), it can be easily compiled and run along with the software under test to produce traces. And the traces can be automatically analyzed to detect bugs.

We selected a large-scale software MySQL as our benchmark program, which contains 22 official test suites, and generated traces by running the test suites. We randomly chose the traces used for experiments, with the following criteria: remove all single-threaded traces, keep all traces with size under 10 GB, keep up to 10 traces with size between 10-20 GB, and keep up to 5 traces with size over 20 GB. In total, we had 256 traces, with each trace ranging from 16 MB (2.2 million events) to 39 GB (1.5 billion events), and the number of threads ranging from 2 to 40. Table III provides information about the traces, including the names of test suites, the number of traces per suite, and the thread number.

The experiments are conducted on servers installed with Ubuntu Linux 20.04 (64 bits). Each server has an Intel Xeon Gold 6240R CPU and 256GB RAM. All tools are executed concurrently with the same number of threads.

TABLE III: Results on MySQL.

Suite	**Traces**		**M2**			**SeqCheck**				**ReconP**		
	Num	Threads	Races	Time (h)	RAM (GB)	Races	FP	Time (h)	RAM (GB)	Races	Time (h)	RAM (GB)
auth_sec	14	2-27	388	1.92	85	478	5	2.17	89	607	4.35	170
binlog	5	2-27	374	1.05	77	461	23	1.17	79	552	2.90	141
con_ctrol.	2	2-27	172	0.20	72	212	4	0.20	76	222	0.87	111
federated	4	2-27	354	0.72	76	443	11	0.81	79	541	1.99	119
funcs_1	3	2-27	276	0.69	73	337	10	0.74	76	435	1.80	117
gcol	6	2-30	443	1.42	74	563	18	1.47	76	670	6.04	170
gis	2	2-27	179	0.20	69	211	3	0.20	72	230	0.84	107
innodb	54	2-33	820	6.12	104	1,026	40	6.64	108	1,164	19.75	185
innodb_fts	3	2-27	274	0.38	71	344	9	0.40	74	420	1.19	116
innodb_gis	4	2-27	348	0.55	72	433	14	0.57	75	499	1.78	111
innodb_undo	3	2-27	274	0.49	70	336	8	0.53	73	422	1.46	138
innodb_zip	26	2-28	620	3.60	91	743	24	3.69	96	882	11.67	176
json	2	2-27	179	0.19	71	212	4	0.21	73	221	0.92	106
main	28	2-27	426	3.92	107	519	10	4.20	111	635	7.69	143
opt_trace	3	2-27	433	0.72	71	510	6	0.74	74	600	1.90	154
parts	14	2-27	404	1.65	87	507	15	1.83	90	626	4.16	131
perfschema	5	2-28	353	2.73	102	445	9	3.26	106	535	4.35	213
query_re.	8	2-40	561	6.08	97	664	14	6.66	100	786	18.07	203
rpl	11	2-32	461	3.58	84	602	32	3.71	88	701	9.32	189
sys_vars	44	2-38	460	9.47	101	566	11	9.78	105	721	17.49	215
sysschema	3	2-27	270	0.45	73	340	10	0.51	76	427	1.18	118
test_ser.	12	2-32	525	3.08	87	645	9	3.84	91	779	8.63	174
Total	256	2-40	8,594	49.21	107	10,597	289	53.33	111	12,675	128.35	215
Possible Race Pairs			220,759,080			381,213,471				4,001,390,978		
Avg	**Additional Races** Detected by ReconP than M2/SeqCheck across each suite: 46.9%/22.4% **Time/Memory overhead** on M2 and SeqCheck across each suite: 198%/81.6% and 177%/74.5%											

C. Results Analysis

In our experiments, we distinguish races based on the instructions corresponding to the events [16]. If the instruction pairs of two races are the same, we consider the two races to be duplicates. For each tool tested, Table III shows the number of detected races, the time cost, and the memory consumption. Additionally, the second-to-last row shows the number of possible race pairs identified by each tool. Possible race pairs refer to the conflict pairs that each tool considers as potential races. Each tool will perform verification on the possible race pairs it has identified. The tested tools identify possible data races using the same method, but the results vary due to differences in the extracted constraints. We evaluate ReconP from three aspects: effectiveness, efficiency, and memory cost.

1) Effectiveness:

Theoretically, the bug prediction capability of ReconP fully encompasses that of M2 and SeqCheck. Table III reveals that, on average, ReconP detected 46.9%/22.4% more races than M2/SeqCheck on each test suite. Since SeqCheck may produce false positives in C/C++, as stated in [19], we manually verified the races and listed the number of false positives in SeqCheck in Table III (FP). Upon comparing the detailed bug reports, we found that ReconP identified all true races detected by M2 and SeqCheck and avoided all false positives. The experimental results align with the theoretical expectations. These findings indicate that NRE can significantly improve the effectiveness of sound concurrency bug prediction methods, while still maintaining soundness.

2) Efficiency:

When utilizing static information to detect more concurrency bugs, there is an inevitable introduction of time and memory overhead. For instance, ToccRace incurs a time overhead which is over 10 times that of SeqCheck [19]. Regarding ReconP, when compared to M2/SeqCheck, it shows an average time overhead of 198% and 177%, respectively, across each suite. The overhead is primarily caused by the hybrid analysis and the verification of a larger number of possible race pairs.

Although the time overhead seems high, we can still declare that the hybrid analysis algorithm is very efficient. This is evident from the following two facts:

1) The hybrid analysis algorithm accounts for only 34% of the total running time. The majority of the time overhead is due to the need to verify a larger number of possible race pairs, which is more than 10 times that of M2 and SeqCheck, as shown in Table III.
2) Compared to other sound prediction methods, M2 and SeqCheck are the most efficient. The time overhead of ToccRace is more than 10 times that of SeqCheck [19]. Constraint-solver-based methods are even less efficient than ToccRace. Considering that the time overhead of the hybrid analysis algorithm is only related to the size of the trace, it is foreseeable that when NRE and the hybrid analysis algorithm are applied to ToccRace or constraint-solver-based methods, the execution time of the hybrid analysis algorithm could drop to approximately 9.4% or even lower ($34\% \times (1 + 177\%)/10$).

3) Memory Cost:

According to the data presented in Table III, when compared with M2 and SeqCheck, the memory overhead of ReconP is on average 81.6% and 74.5%, respectively. This overhead is primarily due to the recording of the hybrid analysis results.

To summarize, the results indicate that NRE can greatly effectively improve the capabilities of existing sound concurrency bug prediction methods, and the hybrid analysis algorithm is highly efficient. Furthermore, both NRE and the hybrid analysis algorithm do not compromise the soundness of bug prediction. These findings demonstrate that NRE and the hybrid analysis algorithm proposed in this paper hold significant practical value and have broad applications in the field of concurrency bug prediction.

VI. Related Works

Static approaches analyze program source code (or intermediate code) and combine characteristics of multi-threaded programs to predict concurrency bugs [28]–[30]. However, accurately modeling program behaviors statically is challenging due to imprecise pointer analysis. Hence, these approaches can report many false positives.

Early **dynamic works** are based on either Happens-before relations [31] or Lockset principle [32]. The former [4]–[13], [33], [34] are usually based on vector clocks. They only monitor high-level thread communication like lock acquisition/release and hence report both false negatives and false positives. Lockset-based ones [35]–[38] can be sound but report too many false negatives.

Many works aim at improving accuracy for Happens-before-based approaches, including SHB [39], CP [40], WCP [41], DC [42], SDP and WDP [43]. They further track memory operations to build **sound partial orders**. SDP and WDP even attempted to incorporate control/data dependency information. However, they rely on a conservative strategy to guess control/data dependency information solely based on the traces. The absence of static information makes the dependency information highly imprecise, leading to limited improvements in effectiveness and the potential for false positives. Compared with these approaches, NRE integrates both static and dynamic information to gather precise dependency information, thus enables us to detect more races.

Constraint solver can be adopted to soundly predict data races where all necessary partial orders are turned into constraints [14], [44]. Theoretically, these approaches can detect all races in given traces. However, in practice, their effectiveness is limited by time-consuming constraint solvers and they usually analyze a window of traces [14]. While constraint solvers can be optimized based on the characteristics of multi-threaded programs, this can only serve as a mitigation and does not fundamentally solve the efficiency problem. Static information has been integrated into constraint solvers for optimization in certain methods. For instance, MCR-S [45] reduces constraints by incorporating dependency information with the help of the system dependency graph. However, the accuracy of static information adopted by these methods currently is much lower than that of NRE, limiting its effectiveness in optimization.

Systematically searching execution space can reduce false negatives. Model checkers can explore all possible thread interleavings [46], [47]. Although many partial order reduction techniques have been proposed to reduce execution states [48], they still suffer from space explosion. A more practical approach is to first identify a set of potential scheduling point and then to systematically exploring the identified space [49]–[58]. This can alleviate space explosion but cannot avoid it. NRE and the hybrid analysis algorithm can also be combined with these techniques to predict races under different traces.

VII. Conclusion

In this paper, we propose **Necessary Consistent Read Event** to break the common assumption of existing sound concurrency bug prediction methods: any (memory) read can fully affect subsequent program execution via control-flow and data-flow. We further develop a hybrid analysis algorithm to calculate NRE efficiently. Both NRE and the hybrid analysis algorithm can be easily integrated into existing concurrency bug prediction methods to improve their bug prediction capabilities with low overhead while maintaining soundness, regardless of the type of concurrency bug they target and whether they are partial-order-based or constraint-solver-based.

ACKNOWLEDGMENT

We thank anonymous reviewers for their insights on improving this work. This work is supported in part by the National Key Research and Development Program of China (No. 2022YFB3104004), National Natural Science Foundation of China (NSFC) (Grant No. 61932012, 62132020, 62232016, 62272465), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. Y2021041), and the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006). The results presented in this paper were partially obtained using the Chameleon testbed.

References

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: Association for Computing Machinery, 2008, p. 329–339. [Online]. Available: https://doi.org/10.1145/1346281.1346323

[2] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM J. Comput.*, vol. 26, no. 4, p. 1208–1244, aug 1997. [Online]. Available: https://doi.org/10.1137/S0097539794279614

[3] U. Mathur, A. Pavlogiannis, and M. Viswanathan, "The complexity of dynamic data race prediction," in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 713–727.

[4] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: A race and transaction-aware java runtime," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 245–255. [Online]. Available: https://doi.org/10.1145/1250734.1250762

[5] E. Pozniansky and A. Schuster, “Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs,” *ACM Trans. Comput. Syst.*, vol. 19, no. 3, p. 327–340, Nov. 2007.

[6] K. Serebryany and T. Iskhodzhanov, “Threadsanitizer: data race detection in practice,” in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA ’09, 2009, pp. 62–71.

[7] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov, “Dynamic race detection with llvm compiler,” in *Runtime Verification*, ser. RV 2011, 2012, pp. 110–114.

[8] K. Sen, “Race directed random testing of concurrent programs,” in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 11–21. [Online]. Available: https://doi.org/10.1145/1375581.1375584

[9] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, “Effective data-race detection for the kernel,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI ’10, 2010, p. 151–162.

[10] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy, “Detecting and surviving data races using complementary schedules,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11, 2011, p. 369–384.

[11] L. Effinger-Dean, B. lucia, l. Ceze, D. Grossman, and H.-J. Boehm, “Ifrit: interference-free regions for dynamic data-race detection,” in *Acm International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’12, 2012.

[12] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia, “Valor: Efficient, software-only region conflict exceptions,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 241–259. [Online]. Available: https://doi.org/10.1145/2814270.2814292

[13] S. Biswas, M. Cao, M. Zhang, M. D. Bond, and B. P. Wood, “Lightweight data race detection for production runs,” in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC’17. ustin, TX, USA: Association for Computing Machinery, 2017, pp. 11–21.

[14] J. Huang, “Ufo: Predictive concurrency use-after-free detection,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 609–619.

[15] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang, “Detecting concurrency memory corruption vulnerabilities,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 706–717.

[16] A. Pavlogiannis, “Fast, sound, and effectively complete dynamic race prediction,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, dec 2019. [Online]. Available: https://doi.org/10.1145/3371085

[17] U. Mathur, A. Pavlogiannis, and M. Viswanathan, “Optimal prediction of synchronization-preserving races,” *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021.

[18] Y. Cai, H. Yun, J. Wang, L. Qiao, and J. Palsberg, “Sound and efficient concurrency bug prediction,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 255–267. [Online]. Available: https://doi.org/10.1145/3468264.3468549

[19] S. Zhu, Y. Guo, L. Zhang, and Y. Cai, “Tolerate control-flow changes for sound data race prediction,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1342–1354.

[20] Y. Guo, S. Zhu, Y. Cai, L. He, and J. Zhang, “Reorder pointer flow in sound concurrency bug prediction,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[21] Y. Cai and L. Cao, “Effective and precise dynamic detection of hidden races for java programs,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 450–461.

[22] M. Said, C. Wang, Z. Yang, and K. Sakallah, “Generating data race witnesses by an smt-based analysis,” in *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*. Springer, 2011, pp. 313–327.

[23] J. Huang, P. O. Meredith, and G. Rosu, “Maximal sound predictive race detection with control flow abstraction,” in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 337–348.

[24] J. Huang and A. K. Rajagopalan, “Precise and maximal race detection from incomplete traces,” *Acm Sigplan Notices*, vol. 51, no. 10, pp. 462–476, 2016.

[25] C. G. Kalhauge and J. Palsberg, “Sound deadlock prediction,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.

[26] Y. Cai, J. Zhang, L. Cao, and J. Liu, “A deployable sampling strategy for data race detection,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 810–821.

[27] S. Muchnick, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[28] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, “Effective static analysis of concurrency use-after-free bugs in linux device drivers,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’19. USA: USENIX Association, 2019, p. 255–268.

[29] Y. Cai, P. Yao, and C. Zhang, “Canary: practical static detection of inter-thread value-flow bugs,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1126–1140.

[30] F. He, Z. Sun, and H. Fan, “Satisfiability modulo ordering consistency theory for multi-threaded program verification,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1264–1279.

[31] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978.

[32] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, p. 391–411, Nov. 1997.

[33] C. Flanagan and S. N. Freund, “Fasttrack: Efficient and precise dynamic race detection,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’09. New York, NY, USA: ACM, 2009, pp. 121–133.

[34] M. D. Bond, K. E. Coons, and K. S. McKinley, “Pacer: Proportional detection of data races,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 255–268.

[35] Y. Yu, T. Rodeheffer, and W. Chen, “Racetrack: Efficient detection of data race conditions via adaptive tracking,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 221–234.

[36] C. von Praun and T. R. Gross, “Object race detection,” in *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA ’01. New York, NY, USA: Association for Computing Machinery, 2001, p. 70–82. [Online]. Available: https://doi.org/10.1145/504282.504288

[37] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and precise datarace detection for multithreaded object-oriented programs,” in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI ’02. New York, NY, USA: Association for Computing Machinery, 2002, p. 258–269. [Online]. Available: https://doi.org/10.1145/512529.512560

[38] H. Nishiyama, “Detecting data races using dynamic escape analysis based on read barrier,” in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, ser. VM ’04, New York, NY, USA, 2004, pp. 127–138.

[39] U. Mathur, D. Kini, and M. Viswanathan, “What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018.

[40] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, “Sound predictive race detection in polynomial time,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 387–400.

[41] D. Kini, U. Mathur, and M. Viswanathan, "Dynamic race prediction in linear time," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 157–170.

[42] J. Roemer, K. Genç, and M. D. Bond, "High-coverage, unbounded sound predictive race detection," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 374–389.

[43] K. Genç, J. Roemer, Y. Xu, and M. D. Bond, "Dependence-aware, unbounded sound predictive race detection," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019.

[44] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: Association for Computing Machinery, 2012.

[45] S. Huang and J. Huang, "Speeding up maximal causality reduction with static dependency analysis," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[46] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[47] K. Havelund and T. Pressburger, "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 366–381, 2000.

[48] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 446–455.

[49] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 151–162.

[50] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: Association for Computing Machinery, 2009, p. 25–36.

[51] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 162–180.

[52] M. Abdelrasoul, "Promoting secondary orders of event pairs in randomized scheduling using a randomized stride," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 741–752.

[53] Y. Cai and Z. Yang, "Radius aware probabilistic testing of deadlocks with guarantees," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 356–367.

[54] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "SKI: Exposing kernel concurrency bugs through systematic schedule exploration," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 415–431.

[55] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *ACM Trans. Parallel Comput.*, vol. 2, no. 4, feb 2016.

[56] Z. Wang, D. Zhang, S. Liu, J. Sun, and Y. Zhao, "Adaptive randomized scheduling for concurrency bug detection," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019, pp. 124–133.

[57] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, "Controlled concurrency testing via periodical scheduling," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 474–486.

[58] Y. Cai and W. K. Chan, "Magicfuzzer: Scalable deadlock detection for large-scale applications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 606–616.