BiRD: Race Detection in Software Binaries under Relaxed Memory Models

RIDHI JAIN and RAHUL PURANDARE, Indraprastha Institute of Information Technology Delhi
SUBODH SHARMA, Indian Institute of Technology Delhi

Instruction reordering and interleavings in program execution under relaxed memory semantics result in non-intuitive behaviors, making it difficult to provide assurances about program correctness. Studies have shown that up to 90% of the concurrency bugs reported by state-of-the-art static analyzers are false alarms. As a result, filtering false alarms and detecting real concurrency bugs is a challenging problem. Unsurprisingly, this problem has attracted the interest of the research community over the past few decades. Nonetheless, many of the existing techniques rely on analyzing source code, rarely consider the effects introduced by compilers, and assume a sequentially consistent memory model. In a practical setting, however, developers often do not have access to the source code, and even commodity architectures such as x86 and ARM are not sequentially consistent.

In this work, we present BiRD, a prototype tool, to dynamically detect harmful data races in x86 binaries under relaxed memory models, TSO and PSO. BiRD employs *source-DPOR* to explore all distinct feasible interleavings for a multithreaded application. Our evaluation of BiRD on 42 publicly available benchmarks and its comparison with the state-of-the-art tools indicate BiRD's potential in effectively detecting data races in software binaries.

CCS Concepts: • **Software and its engineering → Software verification and validation;**

Additional Key Words and Phrases: Relaxed memory models, TSO and PSO, dynamic race detection, software binaries

ACM Reference format:
Ridhi Jain, Rahul Purandare, and Subodh Sharma. 2022. BiRD: Race Detection in Software Binaries under Relaxed Memory Models. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 58 (July 2022), 29 pages.
https://doi.org/10.1145/3498538

1 INTRODUCTION

Concurrent model of computation has gained popularity since the introduction of shared-memory multiprocessors. It can improve system performance by manifold. However, a glaring caveat to the preceding is that concurrency bugs hamper developers' productivity. According to a survey conducted at Microsoft, more than 60% of the developers had to deal with concurrency-related bugs [29], which are often hard to reproduce.

Authors' addresses: R. Jain, A-518, Research and Development Block, IIIT-Delhi, New Delhi, 110020, India; email: ridhij@iiitd.ac.in; R. Purandare, A-503, Research and Development Block, IIIT-Delhi, New Delhi, 110020, India; email: purandare@iiitd.ac.in; S. Sharma, 419 Bharti Building. IIT Delhi, New Delhi, Hauz Khas 110016, India; email: svs@iitd.ac.in.

1049-331X/2022/07-ART58 $15.00
https://doi.org/10.1145/3498538

Memory models weaker than sequentially consistent are often employed in practice to achieve higher performance. However, as a downside, discovering bugs becomes only harder—weak (aka relaxed) memory models allow many more program behaviors. In some models, such as release-acquire consistency, the behaviors cannot even be explained by the intuitive operational semantics. As a matter of fact, most commodity architectures such as x86, Power, and ARM subscribe to memory models weaker than sequential consistency.

Unlike sequentially consistent or strong memory models, weak memory models allow read and write instructions in a program (also referred to as loads and stores, respectively) to be reordered. The set of instructions that are allowed to be reordered in a program execution depends on the strength of the memory model. For example, under **total store order (TSO)**, a write operation followed by a read operation in the same thread, on two different memory locations that are not separated by explicit memory barrier, can be reordered, whereas under **partial store order (PSO)**, in addition to TSO reorderings, a write operation followed by another write operation in the same thread, on a distinct memory location that not ordered by explicit memory barrier, can also be reordered. Clearly, PSO is weaker than TSO. There are even weaker memory models that allow more reorderings, but TSO and PSO are widely implemented.

Concurrency bugs are oftentimes a by-product of the performance gains offered by shared-memory multiprocessor programming when done without proper synchronization. Such bugs make the program behavior unpredictable, making it imperative to report and fix them. Data races are the most prevalent amongst these concurrency bugs. The problem of minimizing data race bugs while exploiting the capabilities of shared-memory multiprocessors has attracted the interest of the research community for over three decades [14, 16, 23, 61, 79, 88, 90, 93].

Prior research has proposed techniques to locate data races statically [18, 41, 42, 60, 80, 86] to achieve scalability. However, to achieve scalability, static analyses often compromise precision by over-approximating their intermediate results. As a consequence, many false positives are reported. Various *lockset*-based approaches [24, 36, 74] have also been proposed to detect inconsistent usage of locks on the same shared variable by different threads. However, they produce too many false alarms. These false alarms affect developers' productivity. A survey conducted at Google shows that a lower false-positive rate encourages developers to be proactive—they may even fix extra bugs [72].

Considering the importance of low false-positive rates, dynamic analyses [4, 28, 62, 91] that leverage some variant of **dynamic partial order reduction (DPOR)** to guarantee complete coverage for given inputs have been proposed. They explore only *representative* interleavings for those inputs that form a subset of total interleavings, thereby leading to substantial savings during the exploration. Several optimizations have been proposed for the DPOR technique [3, 5, 96].

Recent approaches [4, 47, 62, 96] have been proposed to detect concurrency bugs in relaxed memory models, but their analysis is limited to source code. It is worth observing that source codes may not always be available for real-world applications for non-disclosure of intellectual property. Not only is decompiling a binary generally illegal in such cases but also the decompiled code cannot be guaranteed to work equivalent to the original code. Even if the source code is available, it may contain calls to third-party library APIs for which the source code may not be available. This phenomenon holds even for open-source software, making recompiling the software a challenge, especially with the dynamic dependencies. For example, ChromeOS uses two stripped and obfuscated third-party libraries that are not a part of the open-source Chromium project. Therefore, the authors of VTV [82] had to manually replace the default failure function from these libraries with a white-list failure function for their analysis and evaluation. Another interesting aspect to this problem is that compilers are not bug-free and the proven property for the source code may not hold for the compiled code [78]. Despite the fact that there are scenarios in which analyzing binaries is the only alternative, a solution for detecting data races in software binaries under relaxed memory models remains elusive.

Our work focuses on detecting harmful data races in software binaries in the context of relaxed memory models, in particular, TSO and PSO. In general, data races are hard to detect for the following reasons: (1) the actual error may occur much later in the program after a concurrency bug is detected; (2) not all concurrency bugs lead to an error, 76% to 90% of data races detected by state-of-the-art tools are harmless [43, 61]; (3) these bugs are hard to reproduce, as they might get activated in a particular interleaving; (4) scheduling is mostly architecture dependent (e.g., Intel *x*86/64 architecture supports TSO, whereas IA-64 supports PSO as well); and (5) source code, particularly for third-party libraries, is not always available, and sometimes the binaries available are devoid of the debug information. Analyzing software binaries is challenging for various reasons, including the lack of type and symbol information. It requires variables to be tracked using their addresses, and detecting synchronization operations is non-trivial. We describe challenges in detail in Section 3. Nevertheless, analyzing libraries is extremely important to ensure the overall fault-free operation of the software application that depends on them.

This article presents BiRD, a prototype tool that dynamically detects potentially harmful data races in relaxed memory models. We classify harmful data races as those which lead to an assertion failure. For our evaluation, we have categorized interleavings reaching assertion failures as buggy. We have built BiRD using Intel PIN [54] that can dynamically analyze and instrument a software binary.

We have tested BiRD on 42 openly available benchmarks from litmus tests [10], SV-COMP [1], SCTBench [58], DataRaceBenchmark [33], and DPthread [95]. We were able to uncover bugs that would not manifest under sequentially consistent memory model. To the best of our knowledge, BiRD is the first extensible tool that can detect data races in binaries under TSO and PSO.

The key contributions and articulations that this article makes are the following:

(1) A prototype tool, BiRD, which can detect real data race bugs in software binaries under relaxed memory models (TSO and PSO). BiRD also reproduces all feasible relaxed memory reorderings for TSO and PSO on binaries by explicitly mimicking the cache flush operations and identifying lock and unlock regions, and
(2) Evaluation of BiRD on widely accepted litmus tests for relaxed memory models and its comparison with state-of-the-art tools, Nidhugg [4] and CDSChecker [62]. The article also presents results of BiRD's evaluation on standard *pthread* benchmarks and a case study to assess BiRD's potential to scale to real-world libraries.

The source code of BiRD and the evaluation results along with the benchmarks are available at https://github.com/pag-iiitd/BiRD and https://doi.org/10.5281/zenodo.5709789.

The rest of the article is organized as follows. Section 2 provides motivating examples. In Section 3, we discuss challenges inherent to binaries and dynamic analysis for locating concurrency bugs in TSO and PSO models. In Sections 4 and 5, we describe the design and implementation of the tool. In Section 6, we present the details of experiments performed along with the results. We review the related work in Section 8 and finally conclude our work with future directions in Section 9.

2 BACKGROUND AND MOTIVATION

In this section, we lay the background of important terminologies and techniques used in this article. We discuss and compare the prior arts in the area of detecting data races. We also describe the relevant memory models for this work along with the scheduling algorithms used to explore distinct program behaviors.

2.1 Data Races

A data race is a typical case of unintended non-determinism, which occurs when two or more processes concurrently access the same memory location, with at least one of the accesses being a write access [19]. Even though this non-determinism is often desirable for overall improved performance, it may introduce unexpected outcomes.

```
1 int a = b = 0;
2 void Thread1(){
3   b=a; //r(a), w(b)
4 }
5 void Thread2(){
6   a++; //w(a)
7 }
```

Listing 1. A data race example.

Listing 1 illustrates a data race example. An unsynchronized access to shared variable *a* on lines 3 and 6 by methods *Thread1* and *Thread2* from threads *T1* and *T2*, respectively, is the data race pointed in this example. Due to the data race on *a*, *b* at line 3 can either read 0 or 1 from *a* depending on the execution trace. If the increment operation on *a* at line 6 is performed before the assignment operation at line 3, *b* reads 1 from *a*. *b* is assigned 0 otherwise.

2.2 Data Race Detection Approaches

2.2.1 Lockset-Based Race Detection. Lockset-based approaches rely on the inconsistent usage of locks for the detection of a data race. These approaches rely on critical sections as their primary synchronization model to validate if the program adheres to a specific programming policy, called *locking-discipline*. The *locking-discipline* ensures that locks protect the access to a shared memory address by different threads. Lockset-based data race detection is sound. Nonetheless, this technique considers only locks as order-preserving events and ignores that two accesses to a shared memory location by different threads that are not synchronized by locks may be ordered by other means of synchronization. Therefore, it may detect many false positives and spurious data races.

Figure 1 illustrates a data race between events R_2x of *Thread1* and W_1x of *Thread2*. However, since lockset-based approaches consider only locks as synchronization events, they also report events R_1x and R_3x to be racing with W_1x, even though these events are ordered via thread *fork* and *join* events.

2.2.2 Happens-Before. The **happens-before (*HB*)** relation was originally introduced by Lamport [51] in the context of distributed systems. Since then, it has been extended to determine the dependencies among events in a program. For example, all instructions in the same thread are related by a total order under sequential consistency. Similarly, two synchronization events such as *lock-unlock*, on the same variable in different threads, are also ordered by an *HB* edge. It is denoted by the $\rightarrow$ symbol. The *HB* relation is transitive by nature such that if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$.

The *HB* relation captures all synchronization events and not just the *lock-unlock* events, and therefore does not capture the spurious data races. For example, in Figure 1, the *HB* relation does not report R_1x-W_1x and R_3x-W_1x as racing pairs. The synchronization edges captured by the *HB* relation often over-approximate the real ordering edges and thus misses some data races.

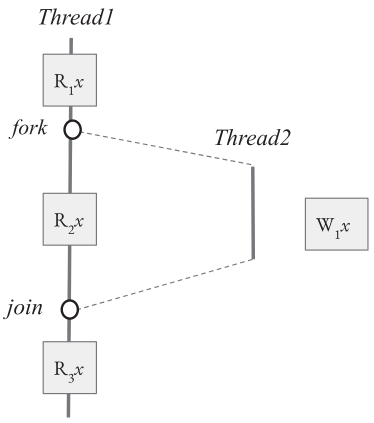


Fig. 1. False positives detected by lockset-based approaches.

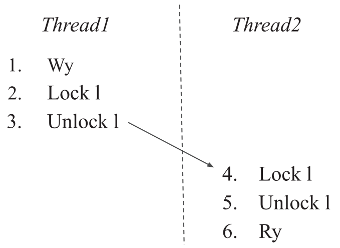


Fig. 2. Missed data race by the *HB* relation.

2.2.3 Causally-Precedes. **Causally-precedes (*CP*)** generalizes the *HB* relation such that it observes more data races without compromising the soundness. The *HB* relation often adds unnecessary dependency edges between two events that may occur simultaneously, thus missing some races, whereas *CP* edges are a subset of *HB* edges and therefore may detect more data races. A *CP* edge between two locked regions is only recorded if they contain conflicting events. Consider the example in Figure 2, where the *HB* relation misses the race on events W_y in *Thread1* and R_y in *Thread2* as it captures an ordering edge between *Unlock l* in *Thread1* and *Lock l* in *Thread2*. Nonetheless, since the two locked regions do not contain conflicting events, the *CP* relation does not record an ordering edge between them and therefore correctly identifies the data race. Figure 3 shows a *CP* edge between two locked regions containing conflicting events and missing a data race.

2.2.4 Maximal Causality Reduction. The aforementioned techniques consider locks as either sole or one of the deciding entities to restrict the ordering of two events. Although that holds true for certain cases, falsely capturing a dependency edge may lead to missed data races. For example, even though *CP* captures more races than the *HB* relation, it may still miss data races. **Maximal causality reduction (MCR)** [37] optimally solves this problem by correctly encoding the minimal and required set of constraints on the ordering of the events. For reference, see Figure 3. The *CP* relation fails to capture the data race between W_y in *Thread1* and R_y in *Thread2*, as it incorrectly captures a dependency edge between *Unlock l* in *Thread1* and *Lock l* in *Thread2*, whereas the only constraints captured by MCR for the same example are $(O_4 < O_5) \bigvee (O_5 < O_4)$. O_4 and O_5 represent the events at lines 4 and 5.

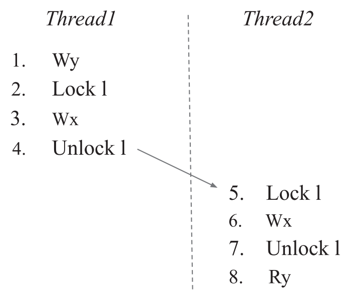


Fig. 3. Missed data race by the *CP* relation.

2.3 Thread Schedules and Partial Order Reduction

Concurrency bugs, including data races, do not always result in an unexpected output. Most of the detected data races are false alarms and may never lead to failure. However, harmful data races may result in a failure in a specific sequence of events. Therefore, most of the concurrency bugs go undetected during testing. Although these bugs are hard to reproduce during testing, they might get invoked in the production environment.

The state space for a program tends to grow exponentially in the number of threads and shared variables [84], resulting in a large number of interleavings. These interleavings are distinguished based on the order of execution of the events; therefore, many interleavings are *redundant*. The redundant interleavings are not the same sequences of events but the ones that produce the same result. **Partial order reduction (POR)** techniques have been extensively used in the past to verify concurrent programs [3, 4, 6, 96]. Such techniques are generally employed to reduce the space and time required to verify concurrent asynchronous systems based on commutativity between concurrently executed transitions [66]. The partial order semantics introduces constraints about commuting the order of events [66, 67], referred to as partial order. POR exploits the fact that reordering two non-conflicting (independent events) from two different threads do not produce a different output and hence prunes redundant interleavings. POR techniques also ensure soundness and complete coverage over a fixed set of inputs for the checked property by exploring at least one *representative* interleaving from each equivalence class.

2.3.1 Source-DPOR. *Source-DPOR* is a special case of DPOR developed by Abdulla et al. [6] that explores all distinct program behaviors in an almost minimal set of interleavings. The traditional DPOR uses persistent sets to store the backtrack information of a state. The persistent set is a selected subset of transitions enabled from a state such that the non-selected set does not interfere with the execution of selected ones [27]. *Source-DPOR* replaces persistent sets with *source-sets*. Similar to the persistent sets, *source-sets* also guarantee that at least one interleaving from each equivalence class (aka *Mazurkiewicz trace* [56]) is explored. *Source-sets* are often smaller than persistent sets and therefore explore lesser or equal interleavings compared to the persistent sets. *Source-sets* satisfy the necessary and sufficient condition for the correctness of any DPOR algorithm while assuring complete coverage.

2.4 Weak Memory Models

Memory consistency models define the guarantees governing the ordering and visibility of accesses to shared memory [83]. As relaxed memory models allow reordering of load and store instructions, new bugs may originate, which is not possible in a sequentially consistent memory model. This section discusses the weak memory models later referred to in this article.

```
1  int a = b = x = y = 0;
2  void Thread1(){
3     x = 1; //w(x)
4     p = y; //r(y)
5     if(p == 0)
6       a = 1;
7   }
8   void Thread2(){
9     y = 1; //w(y)
10    q = x; //r(x)
11    if(q == 0)
12      b = 1;
13 } //assert(a==0||b==0)
```

Listing 2. TSO example.

2.4.1 Total Store Order. Under the TSO memory consistency model, each processor maintains its own store buffer queue. A write performed by a processor is enqueued to its store buffer before the updates are written to the memory. These pending updates are only reflected in the memory when the store buffer is flushed. Even with the pending writes in the store buffer, a load executes without waiting. As a result, the load instruction may appear to execute before the store instruction. The order is preserved if the store and the load instructions access the same memory location. TSO guarantees that the order of execution of the store-store, load-store, and load-load instructions is equivalent to the sequence in which the processor issued them.

Listing 2, taken from litmus tests [10], illustrates a TSO example in which two methods, *Thread1* and *Thread2*, are executed by threads *T1* and *T2*, respectively. The assertion at the end will always hold for a sequentially consistent memory model. It will only fail when after the completion of threads *T1* and *T2* both a and b will hold the value 1. This can only happen when p at line 4 reads value 0 from y, and q at line 10 reads value 0 from x. To observe these values, *read(x)* in *Thread2* is executed before *write(x)* in *Thread1* and *read(y)* in *Thread1* is executed before *write(y)* in *Thread2*. Such interleaving is unattainable in a sequentially consistent memory model, as it restricts intra-thread reordering. The example has only three observable behaviors under sequential consistency, where the possible values for a and b can either be (0, 0), (0, 1), or (1, 0), respectively. However, the assertion may fail in TSO. Under the relaxed semantics of TSO, *write(x)* can execute after *read(y)* in *Thread1* and *write(y)* can execute after *read(x)* in *Thread2*, which may produce an added behavior, where both a and b are 1 that can reach the assertion failure.

2.4.2 Partial Store Order. PSO provides fewer guarantees than TSO and therefore is weaker than TSO. Under PSO, the store instructions can be reordered such that the sequence in which two store instructions access two different shared memory locations may differ from the order of their issuance from the processor. PSO restricts load-store and load-load reorderings.

Consider the PSO example in Listing 3 from litmus tests [10] in a sequentially consistent memory model. Under no interleaving, both x and y can have value 2 and the assertion will always hold, since the only possible values for x and y are either (1, 1), (1, 2), or (2, 1), respectively, at the end of the program. The assertion can only fail when *write(y)* is executed before *write(x)* in *Thread1* and *write(x)* is executed before *write(y)* in *Thread2*. Such interleaving can neither be produced under a sequentially consistent memory model nor under TSO, as they preserve store-store order. But PSO allows store-store reordering such that a store operation followed by another store operation on two different memory locations in a same thread can be reordered. For the given example, under

PSO, *write(x)* in *Thread1* can be held and flushed in the memory after *write(y)* has executed, and *write(y)* in *Thread2* can be held and written in the memory after *write(x)* has executed, which can result in assertion failure where both x and y hold value 2.

```
int x = y = 0;
 void Thread1(){
  x = 2; //w(x)
  y = 1; //w(y)
 }
 void Thread2(){
  y = 2; //w(y)
  x = 1; //w(x)
 }//assert(x==1||y==1)
```

Listing 3. PSO example.

2.4.3 *Weak Memory Model in C/C++11 Memory Model.*

The weak memory behaviors introduced by the C11 standard have been a subject of intense study in the past few years [7, 15, 49, 55, 76, 81]. The semantics are largely axiomatic, and the correctness is described via acyclicity axioms. For instance, a subset of C11 with the exclusion of out-of-thin-air behaviors will require the acyclicity of $hb \cup rf \cup sc$, where hb stands for happens-before, rf refers to reads-from, and sc stands for sequential consistency. The central idea in many of the preceding techniques is to compute rf and mo (modification order) relations precisely. This can be a challenge since mo and rf may not be directly observable from program executions. Recently proposed techniques (e.g., [52, 55]) are some of the few contributions that perform data race detection in C11 programs with low-level atomic operations. These techniques, like many others for simpler and stricter memory models, rely on per-thread *HB* clock vectors and thread schedulers based on record-replay. However, as testing techniques, they trade-off coverage for scalability. Notwithstanding the design choices, the C11 memory model is complex and weaker than hardware memory models such as TSO and PSO. The set of behaviors in a C11 program, as a result, could be significantly larger even for small programs with modest configuration such as number of threads, number of shared memory variables, and size of the iteration space of loops.

2.5 Software Binary Analysis

Software binaries (aka binaries) are executable files that contain machine code that computers can understand. Binaries can be analyzed statically or dynamically, like the source code. However, analyzing a binary can be much more challenging than analyzing a source code, as it might not have symbol or type information. In addition, binaries do not provide a high-level abstraction and therefore are complex to understand. Further, since binaries are not supposed to be modified, even simple instrumentations can crash the code [12]. Despite the challenges in analyzing a binary, software binary analysis has been used by several researchers for various solutions such as malware detection [32, 68, 73], extracting semantics of the code [11], and code reconstruction [89]. The execution of a binary can also be modified and controlled by inserting instructions at particular program points. This technique is referred to as software binary instrumentation. In this work, we use Intel PIN to analyze and instrument the binaries dynamically.

3 CHALLENGES

This section lists the challenges pertaining to the detection of data races in software binaries and in the relaxed memory models, which demand solutions specific to this problem. The low-level details along with our proposed solutions are discussed in Sections 4 and 5. The challenges can be conveniently grouped into the categories described in this section.

3.1 Challenges Inherent to Binary Analysis

(C_1) *Detection of locks*: Detecting locks in binaries is non-trivial since there is no unique way to identify them. In a binary, the locking mechanism may be implemented using POSIX thread locks or an atomic *cmpxchg* instruction.

(C_2) *Mapping thread create to join*: Mapping a thread *create* to its corresponding *join* in binaries is a challenge since the associated thread IDs cannot be directly tracked. Consequently, it requires an alternate solution to map the thread *create-join* calls.

(C_3) *Missing thread-joins*: A *pthread_create* call may not be always accompanied by a corresponding *pthread_join* call. Such cases may lead to an execution trace where the main thread finishes before the forked thread.

3.2 Challenges in Detecting Data Races

(C_4) *Reproducing unique interleavings*: Concurrency bugs are hard to detect, as they get triggered in a specific sequence of events. While exploring all possible execution orders of a program, many of them may be redundant. The redundant interleavings are not the same sequences of instructions but the ones that produce the same result. Hence, it is desirable to reproduce all unique interleavings in minimal runs.

(C_5) *Building dependency relation dynamically*: While building a dependency relation dynamically, the events may appear in an order different from the source code. For example, creation of two threads, t_1 and t_2, followed by joining them may execute in a way where t_2 starts executing after t_1 has finished. Incorrectly capturing this thread *join* to *create* edge may miss data races between threads t_1 and t_2, whereas ignoring this edge may add false data races when thread t_1 is created and joined before thread t_2 is created.

(C_6) *Scalability*: The number of interleavings to be explored can grow exponentially with the number of threads and shared variables. Therefore, scaling such tools, particularly in the presence of a large number of shared variables, is always a challenge.

3.3 Challenges Inherent to Relaxed Memory Models

(C_7) *Reordering loads and stores*: BiRD supports relaxed memory models TSO and PSO, which allow intra-thread reordering of load and store operations. Since the memory models are machine dependent, such reorderings cannot be reproduced with thread controlling function calls such as *wait-post* or *sleep*.

(C_8) *Insufficiency of interleaving semantics*: Some of the weaker memory models, such as in C11, may not even be modeled via interleaving semantics. For such memory models, the execution semantics require a change. The work by Kokologiannakis et al. [45] offers a promising direction by modeling executions as *execution graphs* as opposed to a linearized sequence of events. This challenge is left for future work.

4 ARCHITECTURE

In this section, we present the overall structure and design of BiRD, as depicted in Figure 4. BiRD has two phases that work in synchronization to detect harmful data races.

4.1 Detecting Races and Reorderable Instructions

In this phase, we collect all potential data races and reorderable instructions in a single pass with the help of the components handling the following functionalities.

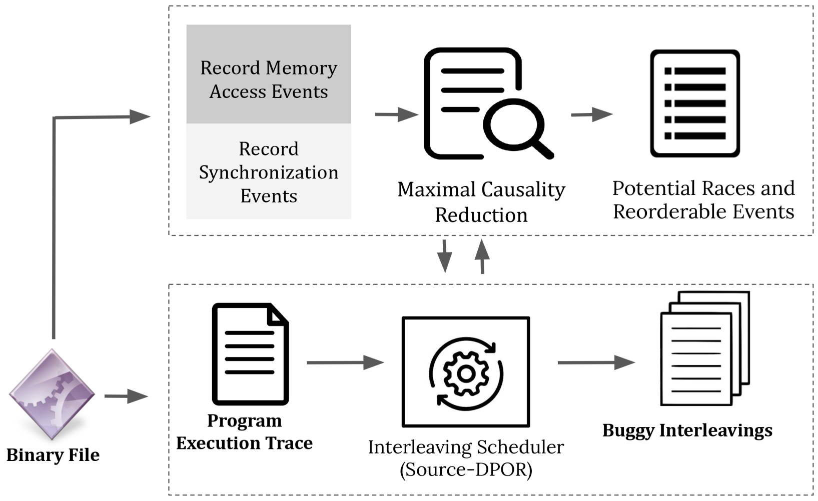

Record Memory Access Events
Record Synchronization Events
Maximal Causality Reduction
Potential Races and Reorderable Events
Binary File
Program Execution Trace
Interleaving Scheduler (Source-DPOR)
Buggy Interleavings

Fig. 4. An overview of BIRD.

Record memory access events. BIRD prunes variables local to a thread and collects events such as reads and writes on only shared memory locations, in phase I.

Record synchronization events. BIRD records the synchronization events such as *lock-unlock*, thread *create-join*, and *memory barriers* to capture the dependency relation across different shared memory access events.

Detecting locks. In a binary using POSIX thread locks to lock and unlock a region, direct function calls *pthread_mutex_lock* and *pthread_mutex_unlock* may be observed; alternatively, it may use an atomic *cmpxchg* instruction for a low-level implementation of a spinlock. Moreover, it does not directly map the lock to unlock instructions from source code to binary. For a single locked region, there can be multiple lock instructions in the binary. In our analysis, we capture *pthread* locks as well as the low-level implementation of spinlocks, as explained in Algorithm 1 in Section 5.

Capturing the thread create-join mapping. To record thread *create-join* calls in order of their appearance in the source code, routine calls *pthread_create* and *pthread_join* can be tracked such that the current thread is created by an application thread and not by PIN to analyze and instrument the client program. Collecting these routine calls does not suffice, as we need thread IDs to map a thread *join* to its corresponding *create*. However, the thread ID is not passed as an argument to these *pthread* routine calls. We discuss the lower-level challenges in mapping a thread *create* to its corresponding *join* and our proposed solution for it in Section 5.1.

In addition, at times, the threads created are not joined using explicit *pthread_join* calls in the program. Consequently, there may exist a trace for such programs where the main thread finishes, thus forcing the forked threads to also terminate without them (possibly) having completed their executions. Since dynamic approaches detect the data races based on the trace explored, they may miss data races under such circumstances. We discuss our approach for handling such cases in Section 5.1.

Maximal causality reduction. BIRD employs *MCR for TSO and PSO* [37] to identify the racing and reorderable pairs of instructions. It relies on **must happens-before (*MHB*)** [35] and lock consistency requirements to capture the control dependency. *MHB* defines ordering between events such as a thread must *begin* after it is forked from another thread and the *begin* instruction must be the first instruction of the forked thread, and a thread must *end* (where *end* is the last instruction in the thread) before it is joined. The lock consistency assures that two sequences of events protected by the same lock are not interleaved. Unlike *CP* and *HB* that are conservative in their ordering edge calculation, MCR uses the control flow information and therefore achieves a higher race detection capability. *MCR for TSO and PSO* relaxes the must happen-before edge for events that satisfy the TSO and PSO reordering semantics, enabling it to explore more interleavings and capture more data races. The *MHB* semantics in MCR correctly captures the thread *create-join* ordering as discussed in Section 5. The control dependency information generated by *MCR for TSO and PSO* dynamically with a fixed set of inputs is used by the scheduler to explore unique interleavings.

Detecting potential data races. Racing instructions can be captured as two events in two different threads on the same shared variable x such that there is no *causal* order between them, and at least one of them is writing to x. BiRD iterates over all events of an execution trace of the program to record the pairs of racing instructions. If it finds a conflicting event that can produce another unique interleaving, it adds the event to the *backtrack* [57] of that state. *Backtrack* for a state stores a set of enabled transitions that are to be explored from that specific program point. The process of reverting the execution from a state by selecting an enabled transition is backtracking.

Detecting reorderable instructions. BiRD targets TSO and PSO reorderable instructions that are located close to each other. We use a window size of up to four instructions to record these reorderable instructions.

4.2 Relaxed Memory Scheduler

In phase II, we explore alternate interleavings using a *source-DPOR* scheduler for relaxed memory. It explores a subset of interleavings with complete coverage.

Detecting the racing instructions is not sufficient, as not all data races are necessarily data race bugs. According to studies, only 10% of data races detected by state-of-the-art tools are harmful [61]. These bugs are triggered in a specific interleaving with a set of preconditions satisfied and remain inert in others. Only identifying the racing instructions does not suffice. Moreover, even the triggered data race bug may result in a crash way later in a program after the race was detected. Therefore, all unique interleavings must be replayed to capture the harmful data races and filter the benign races. However, reproducing all interleavings can be exhaustive and hence can be an expensive process. Further, it may be possible that a significant number of these interleavings is redundant, thus it is desirable to explore all unique interleavings in minimal runs. Therefore, we adopt a relaxed *source-DPOR* [4] to optimize the number of runs, but there remains scope for optimizations as discussed in Section 9. We leave more advanced optimizations as future work. Nevertheless, BiRD can be extended by its users to incorporate their customized optimization techniques. We use the information about racing and reorderable pairs generated in phase I to reproduce the interleavings. The scheduler smartly eliminates redundant interleavings and minimizes the number of runs to optimize the process.

We allow the scheduler to freely execute the program for the first time with a fixed set of inputs to record the reorderable and racing instructions. Every read or write event recorded is a state, and at every state we record the enabled threads that need to be explored from that point in *backtrack* of the state with the set of threads that provably need not be explored in the *sleep-set* of the state. Initially, all threads are enabled. The scheduler picks one thread from the enabled set and executes

```
<spinLock>:
        push    %ebp
        mov     %esp,%ebp
        mov     $0x1,%edx
<spin>:
        mov     [lock_var],%eax #fetch the value of lock_var
        test    %eax,%eax
        jne     c4 <spin> #jump if eax is non Zero
        lock cmpxchg %edx,[lock_var] #check if still locked
        test    %eax,%eax
        jne     c4 <spin>
        pop     %ebp
        ret
<spinUnlock>:
        push    %ebp
        mov     %esp,%ebp
        mov     $0x0,%eax
        xchg    %eax,[lock_var] #lock released
        pop     %ebp
        ret
```

Listing 4. Implementation of spinlock and unlock in x86 assembly.

its active state. Since all events in a thread execute sequentially, unless in a relaxed memory model, the active state of a thread is the next unexecuted event of the thread.

On executing every state, it checks if the current instruction races with any of the previously executed instructions. If found so, it adds the current instruction and the enabled thread to the *backtrack* set of the racing instruction executed before. Each active entry in the *backtrack* set is explored in a different run of the binary. A thread that has explored all interleavings or will not be executed soon is appended to the *sleep-set* [30] of that state. A *sleep-set* is a set of transitions associated with each state during the search. A transition enabled in the *sleep-set* of a state will not be explored from that state.

Bird is developed and evaluated on an Intel x86/64 TSO machine that can hold a write operation on a shared variable to be updated after a read operation in the same thread on a different memory location. Nevertheless, to deterministically explore all interleavings holding stores and flushing later needs to be controlled. We use PIN's native API call *PIN_SafeCopy* to implement an intra-thread scheduler, which reorders the read and write operations in the thread. *PIN_SafeCopy* allows copying a fixed number of bytes from source to destination. We use it to replicate a cache flush mechanism where a write to a memory location is not reflected until the value is flushed in the memory. We describe the details of it in Section 5.

5 IMPLEMENTATION

In this section, we explain the design and implementation details of Bird and how it addresses the challenges mentioned in Section 3.

As discussed in Section 4, we carry out the harmful race detection in two phases. In phase I, we record the execution trace along with the information required to schedule these instructions. While in phase II, we exploit the information collected in the first phase to generate alternate schedules. Both phases work in sync to update the scheduling information and to explore a new interleaving. The two phases are described in the following sections.

ALGORITHM 1: Detecting Spinlock

```
1  $locked \leftarrow false$
2  if $INS\_IsAtomicUpdate(ins)$ then                    ▷ If ins updates memory atomically
3      for $1 \leq i \leq num\_operands$ do
4          if $op_i$ is written then
5              if $op_i = eax$ then
6                  $val\_eax \leftarrow eax\$val_{bef\_write}$      ▷ Get eax register's value before it is written
7              else if $op_i \in mem$ then
8                  $val\_bef \leftarrow mem\$val_{bef\_write}$     ▷ Get memory operand's value before it is written
9                  $val\_aft \leftarrow mem\$val_{aft\_write}$     ▷ Get memory operand's value after it is written
10         if $val\_eax = 0 \wedge val\_bef = 0$ then
11             if $val\_aft != 0$ then
12                 $locked \leftarrow true$
13                 $allLocks.push(mem)$             ▷ Append memory operand to the lock variables' list
```

5.1 Detecting Racing and Reorderable Events

In this phase, we collect the execution trace of a sequence of events and the scheduling information. Each event is represented by its thread ID, index of the instruction in the thread, index of the operand in the instruction on which the event is performed, and type of access on the operand such as read or write. Since we have a pre-compiled binary, we do not have to worry about the compile-time reordering. A total order relates all instructions in a thread, and the index of an instruction in the thread remains unchanged irrespective of the number of runs for the same input values. The execution trace that is passed to the scheduler consists of a series of read and write events on shared memory locations with the *backtrack* and *sleep-set* information. We perform the following analysis to generate this information.

Filter shared addresses. Since all local variables are stored on stack, we use PIN's native API calls *INS_IsStackRead()* and *INS_IsStackWrite()* to discard variables on stack. Out of these, the shared variables are retrieved; we only store memory addresses written to by at least one thread.

Recording events. The read and write events are recorded in the order of their occurrence. We adopt the *MCR for TSO and PSO* relation to register the order between these events. We identify the locked regions by keeping track of a lock operation followed by an unlock operation on the same lock variable in the same thread. This handles the lock consistency requirements for an execution. Since the locked region might not always appear wrapped in the *pthread* locks, we also track the atomic update instructions.

For example, Listing 4 implements a spinlock in x86 assembly. Line 4 sets the value to be assigned to lock variable, *lock_var*, if the lock is obtained, in *edx*. Line 8 sends the control back to line 5 if the lock variable is non-zero. Otherwise, we may obtain the lock. Next, line 9 atomically compares the value of *lock_var* with *eax*, and if the *lock_var* is still zero, the value stored in *edx* is moved to *lock_var* and lock is obtained, else *eax* is set to 1 and the zero flag is set to 0, transferring control back to line 5. For releasing the lock, the *lock_var* is reset to 0.

Algorithms 1 and 2 demonstrate our technique to detect the atomic lock and unlock, respectively, in an x86 binary. We analyze an instruction only if it is an atomic update. An instruction may have multiple operands that are read or written. BiRD iterates over all operands in the instructions and checks if one of the operands written is the *eax* register and the other is a memory address, to

ALGORITHM 2: Detecting Unlock

```
unlocked ← false
found ← false
if INS_IsAtomicUpdate(ins) then                          ▷ If ins updates memory atomically
    for 1 ≤ i ≤ num_operands do
        if op_i is written then
            if op_i ∈ mem then
                if mem ∈ allLocks then                    ▷ If the memory operand is in the lock variables' list
                    found ← true
                val_bef ← mem$val_{bef_write}             ▷ Get locked variable's value before it was written
                val_aft ← mem$val_{aft_write}             ▷ Get locked variable's value after it was written
        if val_bef != 0 then
            if val_aft = 0 ∧ found then                   ▷ Verify if the lock was successfully held
                unlocked ← true
```

verify if the lock was successfully held. We further monitor their values to ensure if the lock was obtained. The lock is only held if before the execution of the instruction both *eax* and the lock variable, *mem*, has value 0 and after execution *mem* contains a non-zero value. After the lock is held, we add the lock variable's address to *allLocks* to correctly identify the unlock events. We record an instruction as a successful unlock operation if the memory operand it is writing to is present in *allLocks*, previously had a non-zero value, and is reset to zero. BIRD offers a modular implementation and could be extended with lock/unlock detection algorithms for architectures different from x86.

Analysis of Algorithms 1 and 2. BIRD ignores all but atomic update instructions for recording the *lock* and *unlock* events. The number of atomic update instructions is generally negligible compared to the execution trace. Nevertheless, under a worst case, all instructions in an execution trace may be of interest, hence the time complexity for Algorithms 1 and 2 is $O(n)$. The algorithms loop over the number of operands. An atomic update instruction can have up to three operands, thus not affecting the time complexity.

These algorithms were successfully able to capture all spinlocks and their corresponding unlocks linear time. n here is the total number of instructions in the execution trace.

BIRD also maintains the thread *create-join* information. Collecting the thread *create-join* details associated with a thread ID poses several challenges, as briefly discussed in Section 4.1. It is difficult to determine the ID of the created or joined thread as *pthread_t* object is passed to these routine calls, and none of the passed arguments can be resolved to thread ID. Although PIN provides thread *start* and *finish* instrumentation callbacks that record the thread IDs, *pthread_create* routine call is encountered before the thread *start* callback API, and *pthread_join* is observed after the thread *finish* callback API. Therefore, recording the thread ID where the routine call was made always returns the current thread's ID, which is the parent thread and, in most cases, the main thread.

The analysis of routines is done before the API is actually called. Hence, the *pthread_t* object is not assigned the value while we analyze the function, which makes it hard to map the thread *create* and *join* even if we capture the arguments passed to the *pthread_create* and *pthread_join* calls. Additionally, in case of two threads t_1 and t_2, such that t_1 created and joined followed by t_2 created and joined, t_1 and t_2 might have the same *pthread_t* object address, which may even complicate mapping by address.

ALGORITHM 3: Mapping Thread Create to Corresponding Join

```
Input: start_info_queue ← {}
Input: thread_info_queue ← {}
1  Function Trace (trace)                                  ▷ The main instrumentation routine
2      if $rtn = pthread\_create$ then
3          $pthread\_create\_CallBefore(rtn\$arg3)$        ▷ Instrument before pthread_create API call
4      $rtn = TRACE\_Rtn(trace)$                           ▷ get routine of the trace
5      if $rtn = pthread\_join$ then
6          $pthread\_join\_CallBefore(rtn\$arg1)$          ▷ Instrument before pthread_join API call
7      $img = getImage(trace)$                             ▷ get Image of the trace
8      if $img \in sharedLibrary \wedge img \notin libOfInterest$ then
9          $return$
10     foreach $Instruction\ ins \in trace$ do
11         $IncrementInsBefore(ins)$
12 Function pthread_create_CallBefore (init_addr)          ▷ Analysis routine
13     $tld \leftarrow threadLocalData(threadid)$
14     $start\_info\$addr \leftarrow init\_addr$
15     $start\_info\$count \leftarrow tld\$insCount$         ▷ Get parent's instruction index
16     $start\_info\_queue.add(start\_info)$
17 Function pthread_join_CallBefore (obj_addr)             ▷ Analysis routine
18     $tld \leftarrow threadLocalData(threadid)$
19     for $thread\_info \in thread\_info\_queue$ do
20         if $thread\_info\$reg\_addr = obj\_addr \wedge\ thread\_info\$end = 0$ then
21             $thread\_info\$end \leftarrow tld\$insCount$   ▷ Get parent's instruction index
22 Function IncrementInsBefore (ins)                       ▷ Analysis routine
23     $threadid \leftarrow getTID(ins)$
24     $tld \leftarrow threadLocalData(threadid)$
25     $tld\$insCount \leftarrow tld\$insCount + 1$
26     if $tld\$insCount = 1$ then
27         for $start\_info \in start\_info\_queue$ do
28             if $start\_info = ins\_addr$ then
29                 $start\_info\$tid = threadid$
30                 for $thread\_info \in thread\_info\_queue$ do
31                     if $thread\_info\$tid = start\_info\$tid$ then
32                         $thread\_info\$init\_addr = start\_info\$start\_addr$
33                         $thread\_info\$start\_count = start\_info\$start\_count$
34                         $start\_info\_queue.remove(start\_info)$
35                         $break$
36                 $break$
37 Function ThreadStart (threadid, context)                ▷ Callback routine
38     $thread\_info\$tid \leftarrow threadid$
39     $thread\_info\$reg\_addr \leftarrow context\$rbx$
40     $thread\_info\_queue.add(thread\_info)$
```

Figure 5 illustrates the thread *create-join* events in order of their appearance with the instrumentations injected. The gray boxes represent the events in the application thread, whereas the blue ones correspond to the injected instrumentations. The blue arrows indicate the program points where the instrumentation is added. Algorithm 3 describes our solution to map the *pthread_create*

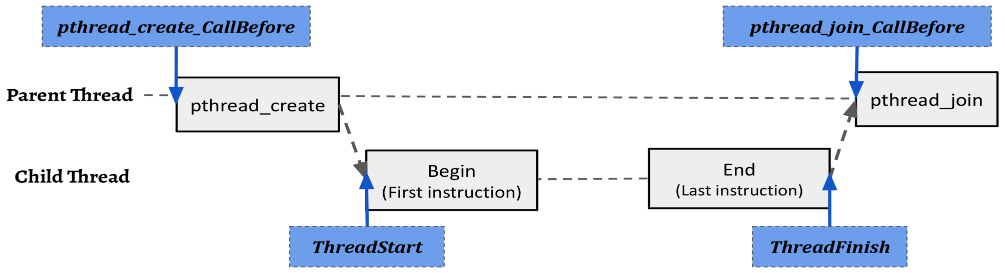


Fig. 5. Chronology of thread *create-join* events with the instrumentation.

call to its corresponding *pthread_join* call using these instrumentations. For brevity, the algorithm focuses only on instructions and routines of interest for capturing the thread *create-join* specifics. We initialize two empty queues, *start_info_queue* and *thread_info_queue*, for storing these details. Bird instruments analysis routine calls *pthread_create_CallBefore* and *pthread_join_CallBefore* before API calls *pthread_create* and *pthread_join*, respectively, in the instrumentation routine *Trace* defined at line 1. In the same routine, it instruments another analysis routine *IncrementInsBefore* before every instruction in a trace that does not belong to a shared library excluding any library of interest. Bird allows selective filtering of library instructions. A routine call *pthread_create* is observed before a thread is created, and therefore the call is devoid of *pthread_t* object information. Thus, in the analysis routine *pthread_create_CallBefore*, we record the initial address of the created thread, passed as the third argument to the *pthread_create* routine. The initial address is then pushed to the *the start_info_queue* along with the instruction count of the parent thread.

The *ThreadStart* callback routine is observed after the *pthread_create* routine and before the execution of the first instruction of that thread. The *rbx* register in the context passed to this routine contains the *pthread_t* object's address, which connects to the *pthread_join* routine call. We collect the ID of the thread started and the value contained in the *rbx* register in this routine, and append it to the *thread_info_queue*. After the thread has started, analysis routine *IncrementInsBefore* is called before every instruction of that thread. However, only the first instruction of a thread is required to associate the *pthread_create* routine call to its thread create information. The ID of the thread created by *pthread_create* routine call can be extracted by matching the *init_addr* stored in *start_info_queue* with the initial instruction's address. The thread ID is then used to copy details from *start_info_queue* to *thread_info_queue* before removing the entry from *start_info_queue*. We require two separate queues to collect these details, because otherwise a single queue would have multiple entries corresponding to the same thread. Finally, we look for *rbx* register's address in *thread_info_queue* that matches *pthread_t* object's address that is passed as an argument to *pthread_join_CallBefore*.

If found, we update the thread *end* with the parent thread's current instruction count. We update the end value only if it is zero, which resolves the cases when two threads have the same object address.

A *pthread_create* call may not always be accompanied by a subsequent *pthread_join* call, leading to missing of data races as explained in Section 3. To handle such cases, Bird waits for all forked threads to successfully complete before finishing the main thread. We place a check before the last routine called every time before terminating the program, *_fini*, to count the number of active threads other than the main thread. If one or more threads is active, the main thread is put to wait. The counter is rechecked after every thread finishes to prevent the main thread from waiting infinitely. We update the end instruction to the last instruction of the parent thread. Since the instructions that can be reordered in a thread do not follow a strict order, memory fences imply this order. Two instructions separated by a *memory barrier* cannot be reordered.

Analysis of Algorithm 3. To map thread *creates* to their corresponding *joins*, BiRD only analyzes *pthread create* and *join* instructions. The number of thread *create* instructions in an application is equal to the number of threads forked, whereas the number of *join* instructions can be up to the number of threads created. The number of threads created is often much lesser than the execution trace length. Although the algorithm iterates over data structures *start_info_queue* and *thread_info_queue*, their sizes are constrained to the number of threads in the target application and thus does not add much overhead. However, in a worst-case scenario, all instructions in an execution trace may correspond to either a thread *create* or a *join* event, making the worst-case complexity $O(n^2)$, where n is the total number of instructions in the execution trace.

Generating reorderable pairs of events. On the filtered addresses of interest, we generate the *MCR for TSO and PSO* to capture the racing and reorderable instructions. Events such as *lock-unlock*, thread *create-join*, and *memory barriers* are treated as synchronizing events. Additionally, *MCR for TSO and PSO* captures the reordering semantics permitted under TSO and PSO. In a combination, the preceding generated constrains capture all valid pairs of data races. We implement a per-thread store buffer to collect the pairs of instructions that can be reordered.

The collected racing pairs are then passed to a systematic DPOR scheduler to optimally schedule all of the races and reorder the load and store operations to produce all unique possible interleavings.

5.2 *Source-DPOR* Scheduler

The scheduler reproduces a subset of possible interleavings in a particular memory model. We use *source-DPOR* to schedule and explore the traces, often in an optimal manner. We implement a thread scheduler using *wait* and *post* functions provided by the standard C library to interleave the inter-thread racing instructions.

TSO and PSO memory models, supported by BiRD, allow intra-thread reordering of load and store operations. These memory models are machine dependent. Therefore, they cannot be rearranged using naive techniques, like for inter-thread reordering. Function calls such as *wait-post* or *sleep* that can control thread execution fail to modify or control the sequence of instructions within the thread.

We maintain store buffers for reorderable instructions. Every time a store operation is encountered in a thread, it is pushed back in the buffer of that thread. Depending on the window size of how many instructions these stores can surpass, it collects the pairs of instructions such that a store is followed by a load or a store on two different memory locations in the same thread within the window size. For our evaluation, we have kept the window size of 4, which can be modified, if required. Apart from maintaining these store buffers, we also keep track of reorderable instructions. When the current instruction can be reordered with any of its preceding instructions, the instruction is pushed to a different *backtrack* set. We maintain a separate setm as the reorderable instructions are handled in a manner different from the races.

To replay all unique interleavings, BiRD recursively checks the last state with an unresolved race. To control the program execution and reverse a race, we use *wait* and *post* calls. Once the race is reversed and an alternate schedule is explored, we add the racing thread with the event to the *sleep-set* of that state.

We use an API call *PIN_SafeCopy* provided by PIN to mimic a memory flush mechanism. Figure 6 is a pictorial representation of Algorithm 4 that shows how we inject the instrumentation to implement it. Initially, we identify instructions that read or write to shared memory locations

ALGORITHM 4: Reordering Instructions

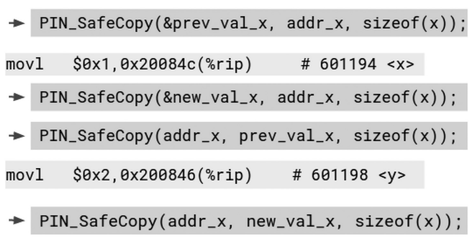
```
for $Instruction\ ins \in trace$ do
    if $ins = stackRead \lor ins = stackWrite$ then        ▷ Return if a local variable is read or written
        $return$
    for $Operand\ op_i \in ins$ do
        if $IsMemWritten(op_i) \lor IsMemRead(op_i)$ then
            for $reorder\_pair \in reorder\_pair\_list$ do
                if $ins = reorder\_pair \rightarrow first \land IsMemWritten(op_i)$ then
                    $reorder\_pair\$addr\_x \leftarrow op_i$
                    $Execute\_Before : reorder\_pair\$prev\_val\_x \leftarrow reorder\_pair\$addr\_x$
                    $Execute\_After : reorder\_pair\$new\_val\_x \leftarrow reorder\_pair\$addr\_x$
                    $Execute\_After : reorder\_pair\$addr\_x \leftarrow reorder\_pair\$prev\_val\_x$
                if $curr\_ins = reorder\_pair\$second$ then
                    $Execute\_After : reorder\_pair\$addr\_x \leftarrow new\_val\_x$
```

```
1 x = 1; // w(x)
2 y = 2; // w(y)
```

Listing 5. Events reorderable under PSO.

```
1 movl $0x1,0x20084c(%rip)
2 # 601194 <x>
3 movl $0x2,0x200846(%rip)
4 # 601198 <y>
```

Listing 6. Equivalent binary for Listing 5.



```
 ➔ PIN_SafeCopy(&prev_val_x, addr_x, sizeof(x));
movl   $0x1,0x20084c(%rip)     # 601194 <x>
 ➔ PIN_SafeCopy(&new_val_x, addr_x, sizeof(x));
 ➔ PIN_SafeCopy(addr_x, prev_val_x, sizeof(x));
movl   $0x2,0x200846(%rip)    # 601198 <y>
 ➔ PIN_SafeCopy(addr_x, new_val_x, sizeof(x));
```

Fig. 6. Instrumentation added to the binary in Listing 6 to reorder store instructions.

and are to be reordered in the current execution. Any read or write operation to a stack variable is ignored. The *reorder_pair_list* contains all pairs of instructions that are reorderable under either TSO or PSO. Consider the code snippet in Listing 5. Statements 1 and 2 can be reordered under PSO given that they do not have any explicit ordering imposed between them. Listing 6 represents a binary equivalent of the snippet. For the given example, *reorder_pair_list* will contain details of the pair of instructions 1 and 3 from Listing 6. To preserve the old value of variable x prior to it is written in instruction 1, we instrument a *PIN_SafeCopy* call before the instruction to save its value at memory address 601194 in *prev_val_x*. Similarly, we store the value of variable x after it is written in instruction 1 in *new_val_x* by instrumenting a *PIN_SafeCopy* call after the instruction. To discard any updates made to x by the first instruction, we overwrite the value stored in x with *prev_val_x*. Finally, when y at memory address 601198 is written value 2 by the second instruction, we update the value of x to *new_val_x*.

Since TSO and PSO allow only store-load and store-store reorderings, the first instruction in the reorderable pair is a store operation. Once reordered, the write operation is added to a different *sleep-set* to ensure the same interleaving is not explored again. While exploring the interleavings, if any particular sequence of events leads to a bug, the trace is recorded.

Analysis of Algorithm 4. Time taken by Algorithm 4 to reorder the candidate instructions is proportional to the number of instruction pair candidates for reordering. Each of these reorderings further calls *PIN_SafeCopy* four times. The number of instructions to be reordered is often very few compared to the entire execution trace. In addition, minimal information such as the old and new value of the first instruction in the pair of reorderable instructions is retained for each reorderable pair. We also keep a flag that indicates whether a pair of instructions was reordered. *MCR for TSO and PSO* precomputes these reorderable instruction pairs. Since the number of reorderable pairs does not exceed the trace length for a single execution, in a worst-case scenario where each instruction is a part of a reorderable pair, this algorithm's time complexity can be $O(n)$.

6 EVALUATION

6.1 Development and Execution Environments

We implemented BiRD on an Intel x86/64 Linux machine with 8 GB of RAM, Core Intel i7-4510U up to 3.10-GHz CPU with Ubuntu 14.04 and kernel 3.19.0-80-generic. We used PIN 2.14 for dynamic binary analysis and instrumentation. We conducted our experiments in the same environment.

6.2 Benchmarks

We selected binaries of 24 publicly available litmus tests [9] widely used by other researchers in the past [25, 70, 96]. Litmus tests contain multiple examples covering all possible communication and synchronization up to a certain size, for ARM and POWER relaxed memory models. Out of these tests, we are only interested in the ones designed for TSO and PSO. We picked tests that would reach the assertion failure in a TSO/PSO reordered interleaving while restricting the reordering in the case of explicit barriers.

In addition, we evaluated BiRD on concurrency benchmarks that have been extensively used in the past by data race detection tools. We used 10 out of 57 tests from SV-COMP's C *pthread* and *pthread-atomic* benchmarks. These benchmarks consist of POSIX threads-based program taken from Software Verification Competition (2019) that may contain a bug. We selected 4 tests from SCTBench such that 3 are from 7 tests in *maple/examples* and 1 is from 54 tests in *concurrent-software-benchmark*. These tests also included an extracted buggy snippet from a real-world application, MySQL. In addition, we picked 3 out of 69 tests from DataRaceBenchmark's *simplebuild* benchmark and 1 out of 12 tests from Deterministic Pthread benchmarks. These benchmarks are not mutually exclusive. For example, stateful01 was a part of both SV-COMP and SCTBench benchmarks. Each of these benchmarks contained information about the reachability of the assertion failure. Some of these benchmarks were used to evaluate data race detection tools, Maple [93] and Actul [34]. We compiled these tests using GCC[1] with no debug information. Since the number of interleavings to explore directly depends on the number of shared variables, we selected the tests based on them. Due to the general scalability issue posed by a dynamic analysis approach, we randomly shortlisted these tests based on the number of threads created, the number of shared variables, the absence of loops, and an assertion failure that might be reachable from a probable data race. We picked tests with up to six shared variables and 16 threads.

6.3 Methodology

To evaluate the effectiveness of BiRD, we conducted our experiments on the collected micro-benchmarks. We intentionally did not preserve the debug information while compiling the selected tests as the proprietary software is often devoid of this information. We used GCCś compiler optimization level zero throughout the experiment since optimizations are not expected to play a role in the outcome of the analysis.

[1]https://gcc.gnu.org/.

Table 1. Results of Running BiRD on Litmus Tests Compared with Nidhugg and CDSChecker

Test Details				BiRD				Nidhugg		CDSChecker		
Name	Execution Trace/ LoC	#Thds/ #Shd Vars	Analyzed Trace/ #Events	Inter-leav-ings	Buggy Traces	First Bug Trace	Time (ms)	Trace Count /Error	Time (ms)	Bug Free Trace	Buggy Traces	Time (ms)
CoRR1	104k/36	2/2	208/16	3	0	–	818	3/N	45	3	0	13
CoRR2	104k/64	4/3	268/22	89	0	–	28,187	72/N	69	47	0	9
CoRW	104k/36	2/2	205/16	3	0	–	888	3/N	43	3	0	14
CoWR	104k/36	2/2	205/16	3	0	–	831	3/N	47	3	0	13
R	104k/38	2/2	209/18	4	1	3	1,248	3/Y	42	3	1	5
R+dmb+po	104k/42	2/3	211/18	5	1	5	1,512	4/Y	82	6	2	10
R+dmbs	104k/41	2/3	213/18	3	0	–	902	3/N	50	10	0	5
R+po+dmb	104k/40	2/3	211/18	4	1	3	1,525	3/Y	43	6	2	13
2+2W	104k/34	2/2	203/17	4	1	3	1,438	3/Y	43	3	1	13
2+2W+reads	112k/60	4/4	293/25	181	2	58	59,851	48/Y	51	48	1	15
2+2W+dmbs	104k/36	2/2	207/17	3	0	–	891	3/N	44	10	0	8
2+2W+dmb+po	104k/36	2/2	205/17	5	1	5	1,837	4/Y	44	6	2	14
2+2W+dmbs+reads	104k/60	4/4	265/25	111	0	–	34,140	108/N	47	294	0	22
RWC	104k/48	4/4	239/22	10	1	10	3,027	8/Y	45	7	1	10
RWC+dmb+po	104k/49	3/4	241/22	10	1	10	3,310	8/Y	47	7	1	16
S	110k/38	3/3	207/18	5	1	3	1,586	3/Y	43	3	1	18
S+dmbs	104k/40	2/3	211/18	3	0	–	869	3/N	44	10	0	11
S+po+dmb	104k/39	2/3	209/18	5	1	3	1,523	3/Y	43	3	1	13
SB	110k/42	4/2	213/19	5	1	3	1,534	3/Y	43	3	1	12
SB+dmb+po	104k/42	2/4	215/19	5	1	5	1,566	4/Y	47	3	1	11
SB+dmbs	104k/44	2/4	213/19	3	0	–	865	3/N	43	4	0	10
WRW+2W	104k/45	2/4	225/21	11	1	11	3,350	11/Y	46	11	1	10
WRW+2W+dmb+po	104k/46	2/4	227/21	12	1	11	3,223	11/Y	48	11	1	16
WRW+2W+dmbs	104k/48	2/4	234/21	9	0	–	2,699	9/N	44	38	0	12

BiRD dynamically instruments software binaries to monitor and control them. We use a PIN specified callback, *PIN_AddThreadStartFunction*, to record the number of threads created by the application. While executing a binary, we use the *-filter_no_shared_libs* option provided by PIN to filter the instructions from shared libraries. Selective filtering can also be done to keep instructions that belong to the *images* of a specified library. The *execution trace length* in Tables 1 and 2 is the total number of instructions observed by PIN for a particular test. For example, PIN observed 1,541 instructions for the test *CoRR1*, out of which only 177 were from the main executable.

We use *MCR for TSO and PSO* [37] to report races and reorderable pairs of instructions that may be missed by the *HB* and *CP* relation. We have adopted the *source-DPOR* algorithm to schedule all runs, which guarantee completeness as well as the (input-specific) soundness of BiRD. *Source-DPOR* explores at least one trace per equivalence class (also known as Mazurkiewicz trace [56]) to ensure complete coverage. Time overhead is computed using function call *clock* defined in header file *ctime*. We finally report all interleavings that resulted in the assertion failure specified in the code as buggy.

6.4 Results

We compare the results of running BiRD on selected litmus tests with Nidhugg [4] and CDSChecker [62] in Table 1.

Since Nidhugg works with LLVM IR to explore TSO and PSO interleavings and CDSChecker needs subtle alterations in source code, such as replacing *main(int, char**)* with *user_main(int, char**)* and *assert* with their custom *MODEL_ASSERT*, we tailored the source code according to their requirements. CDSChecker explores the behavior of concurrent code under the C/C++ memory model. It enumerates the behavior of only atomic operations and the tests need to be compiled against CDSChecker's libraries. None of these tools work directly on binaries. Both of them require either source code or LLVM IR.

Table 2. Results of Running BiRD on Other Benchmarks

Test Details					BiRD			Nidhugg	
Test	Benchmark	#Thds/ #Shd Vars	Execution Trace/ LoC	Analyzed Trace/ #Events	#Inter-leavings	#First Bug	Time (ms)	Trace Count /Error	Time (ms)
bigshot_p	SV-COMP	2/1	208k/48	352/19	2	1	736	1/Y	40
bigshot_s2	SV-COMP	2/1	203k/48	158/20	1	–	308	2/N	32
bigshot_s	SV-COMP	2/1	203k/48	161/21	1	–	293	2/N	33
lazy01	SV-COMP	3/2	203k/64	206/29	6	1	2,376	1/Y	27
sync01	SV-COMP	2/4	205k/78	220/34	2	2	1,232	2/N	32
stateful01_1	SV-COMP	2/4	207k/69	213/35	6	1	2,554	1/Y	28
stateful01_2	SV-COMP	2/4	202k/69	201/34	6	–	2,214	6/N	28
singleton	SV-COMP	6/1	211k/79	265/33	22	4	12,184	4/Y	28
time_var_mutex	SV-COMP	2/3	203k/62	212/34	2	–	338	2/N	27
sigma	SV-COMP	16/2	231k/63	1,010/171	–	1	**TO**	1/Y	39
mysql_169_extracted	SCTBench	2/2	1,515k/120	3,816/178	3	2	3,151	1/Y	301
stringbuffer	SCTBench	1/2	1,506k/180	609/88	3	2	1,713	1/Y	78
bluetooth_driver_bad	SCTBench	1/2	201k/90	235/34	9	5	2,470	7/Y	38
bank_account	SCTBench	2/1	1,496k/95	360/48	9	1	4,326	1/Y	46
account_bad	DataRaceBenchmark	3/6	204k/50	192/34	6	4	2,133	4/Y	54
account_ok	DataRaceBenchmark	3/6	203k/50	192/34	6	–	1,234	6/N	34
actul_half_variable	DataRaceBenchmark	1/2	199k/41	170/16	7	4	2,570	4/Y	66
bankacct	DPthread	1/1	204k/115	203/31	9	1	1,913	1/Y	40

In Table 1, columns 1 through 4 represent details of the tests, such as name, length of the execution trace, lines of code, number of threads created by the application excluding the main thread, the number of shared variables, the length of execution trace analyzed, and the number of memory access events in a single execution. Columns 5 through 8 present the details of running BiRD on the specified tests. We report the number of interleavings explored, the number of buggy interleavings discovered, the first interleaving to reach an assertion failure, and the time overhead for BiRD. The time overhead recorded for all experiments is in milliseconds. Columns 9 and 10 highlight the results produced by Nidhugg, where column 9 shows the total interleavings explored and whether or not a bug was encountered, and column 10 features the cumulative time overhead to convert a test to LLVM IR and analyze it. Nidhugg terminates its exploration as soon as it detects a bug; therefore, to make a fair comparison, we also recorded the occurrence of the first bug on BiRD. Finally, in columns 11 through 13, we report the results of CDSChecker. Nidhugg, as well as BiRD, explore the program behavior under TSO and PSO, whereas CDSChecker explores the program behavior under the C++11 memory model. We observed that BiRD is equally effective in comparison with Nidhugg and CDSChecker in its ability to detect races.

Litmus tests provide information about ordering edges whether an interleaving to reach the assertion failure is allowed or forbidden. In our experiments conducted on litmus tests, we are able to capture all specified edges. Based on the captured edges, we explored all unique feasible interleavings using relaxed *source-DPOR*. BiRD was able to correctly identify the buggy interleavings under TSO and PSO memory models. It did not falsely report any buggy interleaving that was forbidden by the memory model.

We present the results of evaluating BiRD on the tests selected from other benchmarks and its comparison with Nidhugg in Table 2. BiRD was able to capture the reorderable and racing pairs of instructions for all selected tests. It successfully produced the buggy traces that lead to assertion failures. All tests, except *sigma*, were able to finish their exploration within the time-out of 15 minutes. Despite the fact that BiRD could not finish the analysis of *sigma* within the time-out, it could successfully detect the buggy trace leading to the assertion failure. We manually verified that the buggy traces produced by BiRD were indeed feasible traces that led to an assertion failure. BiRD did not explore any infeasible interleaving or report a false bug that cannot be reproduced.

6.5 Performance Analysis of Bird

Since Bird incurred higher runtime overhead compared to CDSChecker and Nidhugg, we investigated further to understand the root causes. We selected a program, *2+2W*, from litmus tests, which took 2 ms to run uninstrumented and executed it with a PIN tool *inscount0*, which is a part of PIN. This program counts the total number of instructions executed with the help of a counter variable. We observed that *2+2W* with *inscount0* took approximately 320 ms to complete an execution for this instrumentation. This can be accounted to the fact that 32 lines of C code for *2+2W* transforms into an execution trace length of approximately 104K instructions. Filtering an instruction or even a trace that is not of interest introduces an insignificant overhead of approximately 0.001 ms, which can cumulate to an elevated cost for the entire execution trace.

Bird explores four interleavings for the same test *2+2W*, which took a total of 1,438 ms. The time recorded by Bird for the test is comparable with the time recorded by *inscount0* when run four times with the same test.

We further recorded the time taken by the major instrumentation performed by Bird. To minimize the overhead, Bird only instruments and analyzes the read/write events on shared memory locations and synchronization events to control the program executions. The time taken by Bird to analyze and control each read/write access to a shared memory location is approximately 0.042 ms. Bird also uniquely associates each instruction with its thread ID and index, where it occurs in the thread. Recording such information like instruction ID and details of events such as synchronization events takes approximately 0.006 ms each. Bird analyzed 203 of total 140k instructions for the test *2+2W* after filtering instructions from the shared libraries. Out of these 203 instructions, there were 17 read/write accesses to shared memory locations. Therefore, the total runtime overhead added by these instruments is 1.93 ms for a single execution and 7.72 ms for four executions of *2+2W*, which is comparable with the runtime overhead of CDSChecker and Nidhugg, and is only 0.53% of the total time taken by Bird on same test. PIN introduced a similar overhead for all other tests. Although PIN offers high control over the execution of a binary, our experiments indicated that the runtime overhead that Bird incurs is mainly attributed to the instrumentation framework provided by PIN and the surged execution trace length compared to the source code. Limiting this overhead would require a more efficient instrumentation framework in the future, which will be an interesting future research direction.

6.6 Case Study to Understand Scalability

Bird, unlike most data race detection tools, does not require access to the source code for the analysis or exploration of interleavings. Although we filtered shared library instructions in our experiments, Bird can reach and analyze shared library instructions, and record data races and reorderable instructions. However, a library instruction trace is enormous compared to the application's instruction trace. Considering that an application may use several API constructs making the entire execution trace too large to analyze, Bird provides an option to selectively analyze instructions from one or more desired libraries and filter others.

To understand the challenges posed by dynamic binary analysis of real-world libraries, we ran Bird on a standard *libboost* multithreaded example[2] using *boost::thread*. The entire execution trace consisted of approximately 1.84M instructions, whereas after filtering all shared library instructions, Bird analyzed only approximately 5.8K instructions with no data race; therefore, only one interleaving was explored. On the same binary, when all shared libraries but *libboost*'s instructions filtered, Bird detected seven data races in the first pass and explored 136 interleavings on an execution trace length of approximately 9.5K instructions with no errors. A single execution

[2] https://theboostcpplibraries.com/boost.thread-management#ex.thread_01.

of the selected example takes approximately 5.0 seconds uninstrumented and about 7.8 seconds with BiRD. BiRD took approximately 19 minutes to exhaustively explore all interleavings and terminate for this *libboost* example.

This study indicates the potential of BiRD to be able to scale to real-world libraries without compromising its effectiveness. We acknowledge that exploration techniques based on a depth-first search can be computationally intensive. Moreover, to limit the number of explorations, we plan to selectively replay the interleavings and restrict the number of context switches as future work.

7 DISCUSSION

7.1 Soundness of the Approach

BiRD uses *MCR for TSO and PSO* to build a dependency relation among events in an execution trace. It is the state-of-the-art technique in reducing the number of false negatives. *MCR* generates the racing and reorderable instruction pairs for each execution. If an execution reports a new race or a reorderable pair of instructions, we check for two conditions before appending it to the *backtrack* of the relevant state: (1) if the *backtrack* of the corresponding state does not already contain a state that explores an equivalent interleaving and (2) the *sleep-set* of that state does not block this reordering. The *backtrack* sets are updated as per the *source-DPOR* algorithm. It ensures the exploration of all equivalence classes.

Since BiRD uses DPOR to uncover the concurrency bugs, the tool inherits the strengths and limitations of dynamic analysis. For example, coverage of dynamic analysis techniques [63, 74, 94] is dependent on inputs provided, the path taken by the program, and the thread interleavings explored at runtime, and hence is limited. Additionally, the number of interleavings to be explored overgrows with the number of shared variables and processes. However, DPOR techniques also provide soundness and completeness assurances for a program over the property checked for a predefined set of inputs. We use *source-DPOR* to optimally explore a subset of interleavings while offering a complete coverage on the given input [6]. We use *MCR for TSO and PSO* for the sound discovery of data races [37].

The litmus tests we selected are specially designed to reach assertion failures only under rare cases of TSO and PSO reorderings, which are unattainable under the sequentially consistent memory model. These tests also include cases where reordering is prohibited and reaching assertion failures is infeasible. As described in Section 6, all of these tests successfully passed on BiRD.

7.2 Strengths and Weaknesses

BiRD does not rely on a symbol table to perform its analysis. We conducted our experiments on binaries that did not contain the debug information. BiRD collects the commuting order of the events and reordering information dynamically during the execution.

BiRD is limited to TSO and PSO relaxed memory models. However, these models are also the ones most widely studied [4, 17, 96]. BiRD can be extended to other relaxed memory models such as C++11, POWER, and ARM.

PIN used by BiRD provides a high control over binaries; however, any internal bug in PIN may affect BiRDś results. Additionally, since the execution trace length is much larger than the lines of code in the original program, performance overhead is a bottleneck. We have implemented our algorithms optimally to reduce this overhead.

8 RELATED WORK

Verification of multithreaded applications for locating concurrency-related defects is a well-explored area [14, 26, 31, 59, 77, 90]. Locating and fixing these bugs has been an active area of research since the introduction of shared-memory multiprocessor programming.

8.1 Approaches for Building a Dependency Relation

Data races are the most common among concurrency bugs. Researchers have proposed static [2, 21, 41, 42, 80, 86, 94] as well as dynamic [13, 14, 22, 64, 65, 91] approaches to report data races. Most of the modern data race detection tools use either a lockset-based approach or *HB* relation to report data races. Lockset-based race detectors [20, 38, 74, 75] can detect data races that are rare in execution paths. The main disadvantage of using lockset-based approaches is that they are unsound and therefore may produce a lot of false alarms. *HB* relation, however, is sound but can miss some data races. FastTrack [26] proposes an adaptive lightweight representation for *HB* for faster detection of data races. Due to over-conservative *HB* edges, its data race detection ability is limited. As an improvement, *CP* relation [35], relaxes some *HB* edges to detect data races that *HB* relation can miss. Even though *CP* relation reports more races than *HB*, it may still miss some data races, as discussed in Section 2. Resettable Encoded Vector Clock [69] provides a scalable solution to implementing vector clocks; however, its practical implementation cannot be applied to any programming language that does not run on the JVM.

8.2 Data Race Detection Under Sequentially Consistent Memory Model

Static analysis [18, 21, 41, 42, 60, 80, 86] is often employed to scale the data race detection tools; however, it escalates the number of false positives by over-approximating the results. About 76% to 90% of data races reported by data race detection tools are benign and never lead to unexpected output. False alarms hamper developers' productivity. According to a survey conducted at Google, a lower false-positive rate encourages developers to be proactive—they may even fix extra bugs [72]. DPOR techniques [8, 27, 40, 71, 87, 92, 96] offer a solution to this by providing complete coverage and soundness on a predefined set of inputs. Dynamic race detection tools, such as Intel Thread Checker [14] and Linecheck [48], do not employ a systematic scheduler and randomly explore program interleavings, and therefore miss out on possible program behaviors. BARRACUDA [23] provides a binary-level analysis based solution for detecting data races. Since it works on PTX code, it can only detect races in CUDA programs. In addition, it does not differentiate harmful from benign data races. The recent dynamic binary analysis based tools ProRace [97] and Razzer [39] that use sampling and fuzzing, respectively, provide a solution for detecting data races. Similarly, Frost [85] explores complementary schedules to detect data races dynamically. Nonetheless, these solutions are neither sound nor do they provide support for relaxed memory models.

8.3 Data Race Detection under Relaxed Memory Models

Most race detectors are designed to work for a sequentially consistent memory model and consequently fail to capture bugs triggered in relaxed memory models. Recently, there has been an escalation in tools for detecting data races in relaxed memory models [4, 44, 53, 62]. Nidhugg [4], an efficient stateless model checker, uses chronological traces [4], which induces partial order relation between relaxed memory executions. It helps DPOR explore minimal interleavings, assuring complete coverage. CDSChecker [62] exhaustively explores a program under the C/C++ memory model. Rocker [50] is a prototype tool that checks the robustness of a concurrent program under C/C++11 release/acquire semantics. An extension to JPF leverages stateless model checking to detect concurrency bugs in Java byte code [44]. Similarly, GenMC [46] is a model checking algorithm for verifying weakly consistent libraries.

8.4 How Bird Differs from Existing Tools and Techniques

Considering the limitations of techniques discussed in Section 8.1, Bird employs *MCR for TSO and PSO* [37], which incorporates the control flow information for a much higher race detection capability compared to the existing techniques. In addition, it can detect data races under relaxed memory models TSO and PSO.

Unlike BiRD, all of the preceding tools work on either source code or its intermediate representation. Nonetheless, source for off-the-shelf software might not always be available. Binary analysis poses challenges, some of which are much bigger than source code analysis.

9 CONCLUSION AND FUTURE WORK

In this article, we present a dynamic binary analysis tool, BiRD, which can effectively detect data race bugs. We have tested BiRD on 24 openly available litmus tests [10] for relaxed memory models and 18 other standard POSIX thread benchmarks under TSO and PSO memory models. For all of the mentioned tests, we are successfully able to uncover the faulty interleavings and discard the false alarms. We use dynamic analysis to record and replay all unique interleavings in a sound and complete way. To the best of our knowledge, BiRD is the first tool that locates harmful data races in a x86 binary under relaxed memory models, particularly TSO and PSO.

Our future work will focus primarily on improving the scalability of BiRD without sacrificing its effectiveness. Although we use *source-DPOR* to reproduce minimal interleavings, it does not remove all redundant interleavings. As the number of interleavings may grow exponentially with the number of threads and shared variables, BiRD will benefit from more advanced optimizations to limit the number of interleavings. We also plan to employ smart heuristics that will prioritize the interleavings to analyze. Another promising research direction would be to combine our approach with static analysis to detect and prune interleavings, which involve racing instructions that perform observably equivalent writes. We also plan to extend this work to other relaxed memory models such as C11/C11++, under which the program executions may not be modeled via interleaving semantics but rather as partially ordered graphs.

REFERENCES

[1] ETAPS. 2019. *COMP 2019—8th International Competition on Software Verification.* Retrieved February 19, 2022 from https://sv-comp.sosy-lab.org/2019/.

[2] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems* 28, 2 (2006), 207–255.

[3] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices* 49, 1 (2014), 373–384.

[4] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless model checking for TSO and PSO. *Acta Informatica* 54, 8 (2017), 789–818.

[5] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source sets: A foundation for optimal dynamic partial order reduction. *Journal of the ACM* 64, 4 (Aug. 2017), Article 25, 49 pages. https://doi.org/10.1145/3073408

[6] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2017. Source sets: A foundation for optimal dynamic partial order reduction. *Journal of the ACM* 64, 4 (2017), 25.

[7] Parosh Aziz Abdulla, Jatin Arora, Mohamed Faouzi Atig, and Shankaranarayanan Krishna. 2019. Verification of programs under the release-acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19).* ACM, New York, NY, 1117–1132. https://doi.org/10.1145/3314221.3314649

[8] Elvira Albert, Puri Arenas, María García de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. 2017. Context-sensitive dynamic partial order reduction. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International, Cham, Switzerland, 526–543.

[9] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. 2013. Software verification for weak memory via program transformation. In *Proceedings of the European Symposium on Programming.* 512–532.

[10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running tests against hardware. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* 41–44.

[11] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. 2016. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). *Digital Investigation* 18 (2016), S11–S22.

[12] Dennis Andriesse. 2018. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly.* No Starch Press.

[13] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively spotting data races in large OpenMP applications. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16).* IEEE, Los Alamitos, CA, 53–62.

[14] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. 2006. Unraveling data race detection in the Intel thread checker. In *Proceedings of the 1st Workshop on Software Tools for MultiCore Systems (STMCS'06).*

[15] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11).* ACM, New York, NY, 55–66. https://doi.org/10.1145/1926385.1926394

[16] Ben Blum and Garth Gibson. 2016. Stateless model checking with data-race preemption points. *ACM SIGPLAN Notices* 51, 10 (Oct. 2016), 477–493. https://doi.org/10.1145/3022671.2984036

[17] Jabob Burnim, Koushik Sen, and Christos Stergiou. 2011. Sound and complete monitoring of sequential consistency for relaxed memory models. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer, Berlin, Germany, 11–25.

[18] Michael Burrows and Rustan Leino. 2004. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience* 16, 12 (2004), 1161–1172.

[19] Jong-Deok Choi and Sang Lyul Min. 1991. Race Frontier: Reproducing data races in parallel-program debugging. *ACM SIGPLAN Notices* 26, 7 (1991), 145–154.

[20] Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. 2008. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08).* ACM, New York, NY, 316–326. https://doi.org/10.1145/1375581.1375620

[21] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. 2003. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience* 15, 3–5 (2003), 485–499.

[22] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. 2012. IFRit: Interference-free regions for dynamic data-race detection. *ACM SIGPLAN Notices* 47 (2012), 467–484.

[23] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-level analysis of runtime RAces in CUDA programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17).* ACM, New York, NY, 126–140. https://doi.org/10.1145/3062341.3062342

[24] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A race and transaction-aware Java runtime. *ACM SIGPLAN Notices* 42 (2007), 245–255.

[25] M. Elver and V. Nagarajan. 2014. TSO-CC: Consistency directed cache coherence for TSO. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14).* 165–176. https://doi.org/10.1109/HPCA.2014.6835927

[26] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. *ACM SIGPLAN Notices* 44 (2009), 121–133.

[27] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. *ACM SIGPLAN Notices* 40, 1 (Jan. 2005), 110–121. https://doi.org/10.1145/1047659.1040315

[28] Patrice Godefroid. 1990. Using partial orders to improve automatic verification methods. In *Proceedings of the International Conference on Computer Aided Verification.* 176–185.

[29] Patrice Godefroid and Nachiappan Nagappan. 2008. Concurrency at Microsoft: An exploratory survey. In *Proceedings of the CAV Workshop on Exploiting Concurrency Efficiently and Correctly.*

[30] Patrice Godefroid, Jan van Leeuwen, Juris Hartmanis, Gerhard Goos, and Pierre Wolper. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Vol. 1032. Springer, Heidelberg, Germany.

[31] Dan Grossman. 2003. Type-safe multithreading in cyclone. *ACM SIGPLAN Notices* 38 (2003), 13–25.

[32] Kyoung Soo Han, Boojoong Kang, and Eul Gyu Im. 2011. Malware classification using instruction frequencies. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation.* 298–300.

[33] Marc Hartung. n.d. DataRaceBenchmark: Concurrency Benchmarks. Retrieved February 19, 2022 from https://github.com/marchartung/DataRaceBenchmark/; https://github.com/marchartung/DataRaceBenchmark/find/master.

[34] Marc Hartung, Florian Schintke, and Thorsten Schütt. 2019. Pinpoint data races via testing and classification. In *Proceedings of the 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'19).* IEEE, Los Alamitos, CA, 386–393.

[35] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14).* ACM, New York, NY, 337–348. https://doi.org/10.1145/2594291.2594315

[36] Jeff Huang and Charles Zhang. 2011. Persuasive prediction of concurrency access anomalies. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, New York, NY, 144–154.

[37] Shiyou Huang and Jeff Huang. 2016. Maximal causality reduction for TSO and PSO. *ACM SIGPLAN Notices* 51, 10 (2016), 447–461.

[38] Jasmin Jahić, Varun Kumar, Matthias Jung, Gerhard Wirrer, Norbert Wehn, and Thomas Kuhn. 2019. Rapid identification of shared memory in multithreaded embedded systems with static scheduling. In *Proceedings of the 48th International Conference on Parallel Processing: Workshops (ICPP'19)*. ACM, New York, NY, Article 15, 8 pages. https://doi.org/10.1145/3339186.3339195

[39] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP'19)*. IEEE, Los Alamitos, CA, 754–768.

[40] Vineet Kahlon, Aarti Gupta, and Nishant Sinha. 2006. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer, Berlin, Germany, 286–299.

[41] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*. ACM, New York, NY, 13–22. https://doi.org/10.1145/1595696.1595701

[42] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the International Conference on Computer Aided Verification*. 226–239.

[43] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: Telling the difference with portend. *ACM SIGPLAN Notices* 47, 4 (2012), 185–198.

[44] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. 2009. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. 495–499.

[45] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2018. Effective stateless model checking for C/C++ concurrency. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), Article 17, 32 pages. https://doi.org/10.1145/3158105

[46] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, NY, 96–110. https://doi.org/10.1145/3314221.3314609

[47] Michalis Kokologiannakis and Viktor Vafeiadis. 2020. HMC: Model checking for hardware memory models. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. ACM, New York, NY, 1157–1171. https://doi.org/10.1145/3373376.3378480

[48] Nikita Koval, Maria Sokolova, Alexander Fedorov, Dan Alistarh, and Dmitry Tsitelov. 2020. Testing concurrency on the JVM with Lincheck. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*. ACM, New York, NY, 423–424. https://doi.org/10.1145/3332466.3374503

[49] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. *ACM SIGPLAN Notices* 51, 1 (Jan. 2016), 649–662. https://doi.org/10.1145/2914770.2837643

[50] Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, New York, NY, 126–141. https://doi.org/10.1145/3314221.3314604

[51] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: The Works of Leslie Lamport*, Dahlia Malkhi (Ed). ACM Books, New York, NY, 179–196.

[52] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++ 11. *ACM SIGPLAN Notices* 52, 1 (2017), 443–457.

[53] Christopher Lidbury and Alastair F. Donaldson. 2017. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17)*. ACM, New York, NY, 443–457. https://doi.org/10.1145/3009837.3009857

[54] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40 (2005), 190–200.

[55] Weiyu Luo and Brian Demsky. 2021. C11Tester: A race detector for C/C++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 630–646.

[56] Antoni W. Mazurkiewicz. 1987. Trace theory. In *Proceedings of an Advanced Course on Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986—Part II*. Springer-Verlag, Berlin, Germany, 279–324. http://dl.acm.org/citation.cfm?id=647422.725772.

[57] David A. McAllester. 1993. Partial order backtracking. *Journal of Artificial Intelligence Research* 1 (1993), 17–24.

[58] Imperial College London Multicore Group. n.d. SCTBench: C/C++ Pthread Benchmarks. Retrieved February 19, 2022 from https://github.com/mc-imperial/sctbench.

[59] Abdullah Muzahid, Norimasa Otsuki, and Josep Torrellas. 2010. AtomTracker: A comprehensive approach to atomic region inference and violation detection. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, Los Alamitos, CA, 287–297.

[60] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. ACM, New York, NY, 308–319. https://doi.org/10.1145/1133981.1134018

[61] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data races using replay analysis. *ACM SIGPLAN Notices* 42 (2007), 22–31.

[62] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*. ACM, New York, NY, 131–150. https://doi.org/10.1145/2509136.2509514

[63] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*. ACM, New York, NY, 167–178. https://doi.org/10.1145/781498.781528

[64] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering—Volume 1*. ACM, New York, NY, 245–254.

[65] Andreas Pavlogiannis. 2019. Fast, sound, and effectively complete dynamic race prediction. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[66] Doron Peled. 2018. *Partial-Order Reduction*. Springer International, Cham, Switzerland, 173–190. https://doi.org/10.1007/978-3-319-10575-8_6

[67] Doron Peled, Antti Valmari, and Ilkka Kokkarinen. 2001. Relaxed visibility enhances partial order reduction. *Formal Methods in System Design* 19, 3 (2001), 275–289.

[68] Roberto Perdisci, Andrea Lanzi, and Wenke Lee. 2008. McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC'08)*. IEEE, Los Alamitos, CA, 301–310.

[69] Tommaso Pozzetti and Ajay D. Kshemkalyani. 2020. Resettable encoded vector clock for causality analysis with an application to dynamic race detection. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (2020), 772–785.

[70] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: Multicopy-atomic axiomatic and operational models for ARMv8. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), Article 19, 29 pages. https://doi.org/10.1145/3158107

[71] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. ACM, New York, NY, 747–762. https://doi.org/10.1145/3385412.3385993

[72] Caitlin Sadowski and Jaeheon Yi. 2014. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*. 43–51.

[73] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K. Penya, Borja Sanz, Carlos Laorden, and Pablo G. Bringas. 2010. Idea: Opcode-sequence-based malware detection. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*. 35–43.

[74] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[75] Ohad Shacham, Mooly Sagiv, and Assaf Schuster. 2005. Scaling model checking of dataraces using dynamic information. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*. ACM, New York, NY, 107–118. https://doi.org/10.1145/1065944.1065958

[76] Divyanjali Sharma and Subodh Sharma. n.d. Thread-modular analysis of release-acquire concurrency. *arXiv preprint arXiv:2107.02346 [accepted in SAS'21]* (n.d.).

[77] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. 2011. RACEZ: A lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 2011 33rd International Conference on Software Engineering (ICSE'11)*. IEEE, Los Alamitos, CA, 401–410.

[78] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, et al. 2016. SOK: (State of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, Los Alamitos, CA, 138–157.

[79] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. *ACM SIGPLAN Notices* 47 (2012), 387–400.

[80] Nicholas Sterling. 1993. WARLOCK—A static data race analysis tool. In *Proceedings of the USENIX Winter 1993 Conference*. 97–106.

[81] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. 2015. Verifying read-copy-update in a logic for weak memory. *ACM SIGPLAN Notices* 50, 6 (2015), 110–120.

[82] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC and LLVM. In *Proceedings of the 23rd USENIX Security Symposium (USENIX'14)*. 941–955.

[83] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. 2017. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. *ACM SIGPLAN Notices* 52, 4 (April 2017), 119–133. https://doi.org/10.1145/3093336.3037719

[84] Antti Valmari. 1996. The state explosion problem. In *Advanced Course on Petri Nets*. Springer, 429–528.

[85] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 369–384.

[86] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*. ACM, New York, NY, 205–214. https://doi.org/10.1145/1287624.1287654

[87] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Germany, 382–396.

[88] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2013. Parallelizing data race detection. *ACM SIGPLAN Notices* 48, 4 (March 2013), 27–38. https://doi.org/10.1145/2499368.2451120

[89] Arne Wichmann. 2012. *Binary Analysis for Code Reconstruction of Control Software*. Diploma Thesis. Hamburg University of Technology.

[90] James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. 2018. VerifiedFT: A verified, high-performance precise dynamic race detector. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York, NY, 354–367.

[91] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. 2008. *Inspect: A Runtime Model Checker for Multithreaded C Programs*. Technical Report UUCS-08-004. University of Utah.

[92] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Efficient stateful dynamic partial order reduction. In *Model Checking Software*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer, Berlin, Germany.

[93] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A coverage-driven testing tool for multithreaded programs. *ACM SIGPLAN Notices* 47 (2012), 485–502.

[94] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM, New York, NY, 221–234. https://doi.org/10.1145/1095810.1095832

[95] Heecul Yun. n.d. DPthread—Deterministic Pthread Benchmarks. Retrieved February 19, 2022 from https://github.com/heechul/dpthread.

[96] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. *ACM SIGPLAN Notices* 50 (2015), 250–259.

[97] Tong Zhang, Changhee Jung, and Dongyoon Lee. 2017. ProRace: Practical data race detection for production use. *ACM SIGPLAN Notices* 52, 4 (2017), 149–162.

Received March 2021; revised September 2021; accepted November 2021