CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2019; **00**:1–15
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.5160

AVPredictor: Comprehensive Prediction and Detection of Atomicity Violations

Pengfei WANG[1]*, Jens KRINKE[2], Xu ZHOU[1], Kai LU[134]

[1]*College of Computer, National University of Defense Technology, Changsha 410073, P.R.China*

[2]*Centre for Research in Evolution, Search and Testing, University College London, London WC1E 6BT, UK*

[3]*Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha 410073, P.R.China*

[4]*Collaborative Innovation Center of High-Performance Computing, National University of Defense Technology, Changsha 410073, P.R. China*

SUMMARY

Concurrency bugs, such as atomicity-violation bugs, are difficult to detect due to the uncertainty of thread-scheduling. It is particularly difficult to conduct a thorough bug fix when an atomicity-violation bug can be triggered by different buggy interleavings. This paper proposes a prediction-based approach to comprehensively detect atomicity-violation bugs. A bug fix can be incomplete when the developer cannot have all the buggy interleavings. Based on the candidate interleavings, this approach can predict unmanifested atomicity-violation bugs from a non-buggy execution and comprehensively display all the buggy interleavings for the same bug to assist a thorough fix. We use a monitored execution to record execution traces and predict potential buggy interleavings based on the candidate interleavings identified from the trace. Then we use controlled executions to verify the predicted buggy interleavings by controlling the thread-scheduling. We implemented a prototype tool called AVPredictor and evaluated it with real-world tests. Experiments show that AVPredictor can effectively find all the known atomicity-violation bugs as well as a previously unknown bug together with all the buggy interleavings for each bug. The runtime overhead is 13x for the monitored execution and 18x for the controlled execution.

Received: Jul. 17th 2018, Accepted: Dec. 23rd 2018.

KEY WORDS: Concurrency Bug; Atomicity-violation Bug; Candidate Interleaving; Group Verification; Prediction-based Detection

1. INTRODUCTION

The increasing application of the multi-core technology in the hardware makes multi-threading pervasive in the software, meanwhile, the reliability of concurrent programs has become a crucial issue. Bug detection in concurrent programs is complicated. Unlike sequential programs whose running states are only affected by the program inputs, the thread-scheduling in concurrent programs is like an implicit input and introduces uncertainty to triggering concurrency bugs, making them difficult to detect and replay. The biggest challenge in concurrent program test is exposing buggy interleavings that are rare. Sometimes the tester has to run the program hundreds of times to accidentally expose

*Correspondence to: College of Computer, National University of Defense Technology, Changsha 410073, P.R.China. E-mail: pfwang@nudt.edu.cn
This is the peer reviewed version of the following article: P. Wang et al. (2019) *AVPredictor: Comprehensive Prediction and Detection of Atomicity Violations*, which has been published in final form at https://doi.org/10.1002/cpe.5160. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

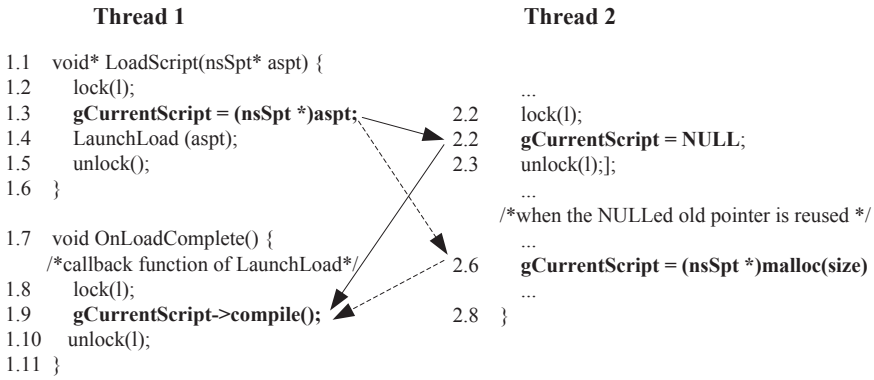

Figure 1. A real atomicity-violation bug in the Mozilla Application Suite.

a buggy interleaving to trigger a hidden concurrency bug. When a concurrency bug can be triggered by different buggy interleavings, it is particularly difficult to comprehensively identify all the buggy interleavings to thoroughly understand the bug and eliminate it. Thus, the bug fix might be incomplete and ineffective, especially when the program is large and complicated. Thus, concurrency bugs tend to escape from in-house testing before software release and cause severe consequences, such as the Therac-25 accident [1] and the 2003 American Blackout [2].

Among the varieties of concurrency bugs, atomicity-violation bugs (abbreviated as AV bugs) [3–6] are one of the most common and significant types, accounting for 70% of the known non-deadlock concurrency bugs [7]. An atomicity violation occurs when the execution of a block of code is unexpectedly interleaved by operations concurrently performed by other threads. It turns to an AV bug if the unfortunate interleaving breaks the atomicity assumptions made by the programmer and leads to incorrect program behaviors. AV bugs widely exist because many programmers are used to sequential thinking and frequently assume code regions to be atomic without proper enforcement [5]. Figure 1 shows a simplified AV bug from the Mozilla Application Suite. In this case, two threads are handling the same shared pointer `gCurrentScript`. In Thread 1, `gCurrentScript` is assigned from an argument pointer `aspt` (line 1.3) and used in function `OnLoadComplete` (line 1.9). In Thread 2, `gCurrentScript` is set to NULL (line 2.2). If the statements execute as 1.3→1.9→2.2, no problem occurs. However, owing to the thread-scheduling, an alternative execution order is 1.3→2.2→1.9, which is problematic as it can cause a NULL pointer dereference.

An AV bug is particularly difficult to fix as it can be triggered by different buggy interleavings. Take Figure 1 for an example, when the deleted pointer in Thread 2 `gCurrentScript` is used again, such as being pointed to a new variable (line 2.6), this AV bug has another buggy interleaving 1.3→2.6→1.9. If the programmer patches this bug only with the first buggy interleaving, then the patch will only modify the locks in Thread 1 to guarantee the atomicity as line 2.2 is already protected by the same lock. However, the program is still buggy because another lock-free access (line 2.6) is neglected. In order to completely fix this bug, the programmer must know the second buggy interleaving and patch line 2.6 with locks. Thus, knowing all the buggy interleavings is of vital importance to thoroughly eliminate an AV bug.

Many approaches [6,8–14] focus on data races to detect AV bugs. However, these approaches are inadequate because: (1) being data race-free does not mean bug-free (such as the case in Figure 1), in many cases, what developers want is the atomicity of code segments, not necessarily freedom from data-race [15]; (2) a data race is not always a bug, and some race situations are intentionally allowed for performance purpose, such as race-based synchronizations, thus, race-based approaches have to distinguish harmful and benign races [16–18]; (3) race-based approaches do not consider different buggy interleavings of AV bugs, which might lead to incomplete bug fix; (4) AV bugs can exist in future race-free transaction-based secure environments [19], such as the Intel SGX. Thus, AV bugs are important problems that need dedicated research.

This paper proposes a prediction-based approach to comprehensively detect AV bugs. This approach can predict unmanifested AV bugs from a non-buggy execution and comprehensively display all the buggy interleavings for the same AV bug to assist a thorough bug fix. We use a monitored execution to dynamically record the execution trace of the target program. Then we analyze the trace and predict

DOI: 10.1002/cpe.5160

potential buggy interleavings based on a metric called the *candidate interleaving*. Finally, we actively control the thread-scheduling to verify the predicted buggy interleavings or to replay a known bug. In summary, we make the following contributions in this paper.

- We present the first work (to the best of our knowledge) that focuses on finding all the buggy interleavings for the same AV bug to assist a thorough bug fix.
- We propose a metric for AV bugs called *candidate interleaving*, which is used to predict unmanifested buggy interleavings from non-buggy executions.
- We propose a prediction-based approach to comprehensively detect AV bugs with all the buggy interleavings. Our approach improves efficiency from three aspects: (1) only cover the potential buggy interleavings by prediction; (2) prune violation-free interleavings before verification; (3) group the candidate interleavings for verification.
- We implemented a prototype tool named AVPredictor and evaluate it with real-world test programs. AVPredictor is now publicly available online.

The rest of the paper is organized as follows: Section 2 describes the candidate interleavings in our approach. Section 3 gives the design of our approach and the key techniques we use. Section 4 introduces the implementation of AVPredictor. Section 5 evaluates AVPredictor by some test experiments. Section 6 discusses the advantages and limitations of our approach. Section 7 surveys the related works of the same background. Finally, our work is concluded in Section 8.

2. CANDIDATE INTERLEAVINGS

An AV bug involves three memory accesses to the same shared variable between two threads – two local and one remote in the other thread, which should theoretically form eight interleaving combinations (as Table I shows). We call the situation that the remote access occurs between the local access pair (Column 2) as *interleaved situation*, and we call the situation that the remote access either occurs before (Column 6) or after (Column 7) the local access pair as a *non-interleaved situation*.

Previous research [5] used *serializability* to detect AV bugs. An interleaving is serializable when the final state of its local and remote accesses in an interleaved execution is equivalent to that in at least one serial non-interleaved execution [15, 20]. Serializability is an indicator used to identify AV bugs because such bugs must be caused by unserializable interleavings and serializable interleavings won't cause bugs.

Owing to the uncertainty of the thread-scheduling, a buggy interleaving (the interleaved situation) in one execution might manifest as a non-buggy interleaving (the non-interleaved situation) in other executions, which aggravates the difficulty in detecting and replaying the bug. Previous detection work [15, 21, 22] focussed on the interleaved situations because they can manifest AV bugs directly. However, since the non-interleaved situations manifest the same serializability as the interleaved situations, they can be used to detect AV bugs from non-buggy executions.

In this paper, we propose a prediction-based approach to detect AV bugs, aiming to connect the interleaved situations with the non-interleaved situations and predict the buggy interleavings from the non-buggy interleavings. As Table I shows, four (No. 3, 4, 6, 7) out of the eight interleavings can cause buggy results due to the unserializability. We name their interleaved situation as the *Atomicity Violation Interleaving* (*AVI*) or *Buggy Interleaving* (*BI*), and name their corresponding non-interleaved situation as the *Candidate Interleaving* (*CI*). More specifically, the *Front Candidate Interleaving* (*FCI*) stands for the situation that the remote access executes prior to the local access pair, and the *Back Candidate Interleaving* (*BCI*) stands for the situation that the remote access executes after the local access pair. Since an AVI and its corresponding CIs are representing the same unserializable interleaving and the serializability does not change across different executions, thus, the buggy interleaving can be predicted based on the candidate interleavings identified from a non-buggy execution.

Our approach can predict and detect the unmanifested buggy interleavings from the non-buggy executions to thoroughly fix the bug. Besides, our approach saves a large number of the verification executions by only covering the potential buggy interleavings and skipping a majority of the violation-free interleavings, which greatly improves the detection efficiency.

DOI: 10.1002/cpe.5160

Table I. Unserializable interleavings and candidate interleavings.

#	AVI*	Description	Serializable	Problem	FCI*	BCI*
1	R_1 R_i R_2	Two local reads interleaved by a remote read.	Yes	N/A – Both FCI and BCI lead to the same state as AVI.	N/A	N/A
2	W_1 R_i R_2	Local read after write interleaved by a remote read.	Yes	N/A – BCI and AVI lead to the same state.	N/A	N/A
3	R_1 W_i R_2	Two local reads interleaved by a remote write.	No	The interleaving write makes the two reads have different views of the same memory location.	W_i R_1 R_2	R_1 R_2 W_i
4	W_1 W_i R_2	Local read after write interleaved by a remote write.	No	The local read does not get the local result it expects.	W_i W_1 R_2	W_1 R_2 W_i
5	R_1 R_i W_2	Local write after read interleaved by a remote read.	Yes	N/A – FCI and AVI lead to the same state.	N/A	N/A
6	W_1 R_i W_2	Two local writes interleaved by a remote read.	No	The intermediate result that is assumed to be invisible to other threads is read by a remote access.	R_i W_1 W_2	W_1 W_2 R_i
7	R_1 W_i W_2	Local write after read interleaved by a remote write.	No	The local write relies on a value from the preceding local read that is then over-written by the remote write.	W_i R_1 W_2	R_1 W_2 W_i
8	W_1 W_i W_2	Two local writes interleaved by a remote write.	Yes	N/A – Both FCI and BCI lead to the same state as AVI.	N/A	N/A

* The subscripts *1* and *2* indicate the two local accesses while *i* indicates the memory access from the remote thread.

3. APPROACH

Our approach relies on two instrumented executions of the target program, the *monitored execution* and the *controlled execution*. We record a memory trace in the monitored execution and predict potential buggy interleavings based on the CIs identified from the recorded trace. Then, we verify potential bugs in the controlled execution. As Figure 2 shows, the approach is divided into five stages.

(1) **Monitoring**: The monitor instruments the target program and dynamically records the execution trace into three log files (the memory access log, lock and unlock log, and synchronization log). The trace includes all memory reads and writes as well as the standard POSIX thread (pthread) operations. If buggy results occur during this execution, then a bug is confirmed. The monitor reports the bug and logs the buggy interleaving for replay.

(2) **Prediction**: The predictor loads traces from the logs and identifies CIs from the trace. If a CI is identified, then a potential AVI is indicated, which can cause an AV bug in another execution when the thread-scheduling changes. The predictor records such CIs for verification.

(3) **Pruning**: The pruner removes the CIs that are infeasible to convert to the corresponding AVI by controlling the thread-scheduling. The happen-before relations that exist in the program enforce the relative ordering of events, making some CIs deterministic, thus, they won't have AVIs. The pruning improves the efficiency by avoiding to execute such violation-free interleavings.

(4) **Grouping**: The grouper puts the non-interfere CIs in a group and verifies them one after another in one controlled execution. The grouping improves the efficiency by reducing the number of controlled executions.

DOI: 10.1002/cpe.5160

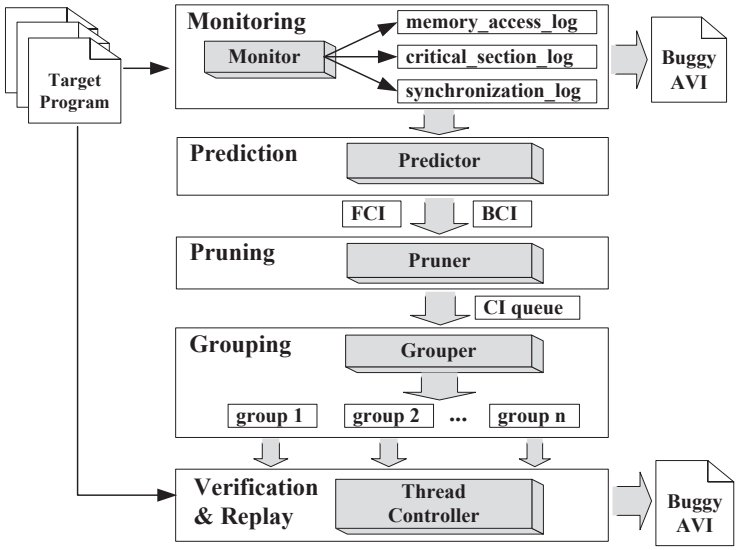

Figure 2. Overview of the prediction-based AV bug detection approach.

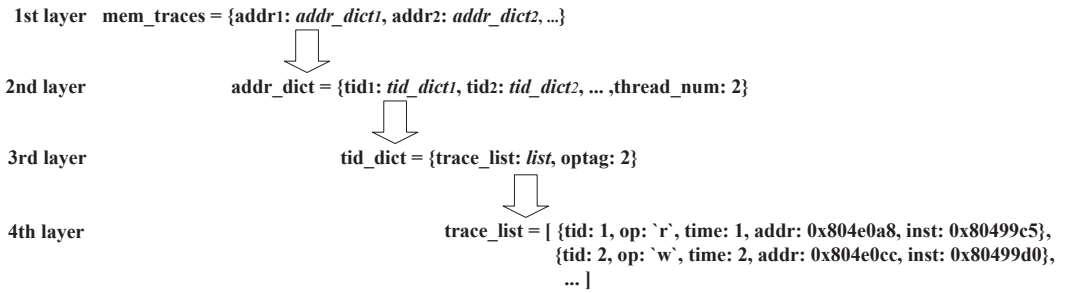

Figure 3. The four-layer memory access storage structure.

(5) **Verification and Reply**: The thread controller loads CIs from each group and controls the thread-scheduling to convert each CI to the corresponding AVI. If the conversion succeeds and a bug occurs, a buggy interleaving is confirmed.

3.1. Identify Candidate Interleavings from the Trace

We identify CIs by matching the memory accesses from the trace. Since memory accesses usually have hundreds of thousands of occurrences, we improve the matching efficiency by applying a four-layer memory access storage and an optimized CI search algorithm.

3.1.1. Four-Layer Memory Access Storage The four-layer memory access storage (as Figure 3 shows) is implemented based on the Python dictionary. In the first layer, memory accesses are divided by the address, and each key-value `address` is mapped by a second-layer structure, which contains all the memory accesses to this address. In the second layer, the memory accesses are further divided by the thread id, and each key-value `tid` is mapped by a third-layer structure, which contains all the memory accesses from thread `tid` and reference address `address`. Besides, the second-layer structure also maintains a thread number counter (`thread_num`). The third layer has two fields, the first field (`trace_list`) points to a fourth-layer structure, which is a list of the memory accesses sorted by timestamp, and the second field (`optag`) is an indicator stating if the accesses in `trace_list` are all reads, all writes, and mixed, respectively.

Prepared using cpeauth.cls
DOI: 10.1002/cpe.5160

Algorithm 1: Identify CIs from the trace.

In: *TRACE* - A structured storage for the accesses.
Out: *CI* - A list to collect the identified CIs.
AVI - A list to collect the buggy AVIs.

```
01  CI ← ∅, AVI ← ∅
02  for Tadr ∈ TRACE do
03    if ThreadNum(Tadr) < 2 then continue
04    if AccessNum(Tadr) < 3 then continue
05    for (Ai, Aj) ∈ Tadr do
06      if AllReads(Ai) and AllReads(Aj) then continue
07      if AllWrites(Ai) and AllWrites(Aj) then continue
08      for (an,am) ∈ Ai, bl ∈ Aj do
09        if (am.op = 'r' ∧ an.op = 'r' ∧ bl.op = 'w') ∨
10          (am.op = 'w' ∧ an.op = 'r' ∧ bl.op = 'w') ∨
11          (am.op = 'w' ∧ an.op = 'w' ∧ bl.op = 'r') ∨
12          (am.op = 'r' ∧ an.op = 'w' ∧ bl.op = 'w') then
13            ui ← (am,an,bl)    /* unserializable interleaving */
14            if bl.time < am.time ∨ bl.time > an.time then
15              CI.append(ui)
16            elif ProgramBuggy() then
17              AVI.append(ui)
18  return CI, AVI
```

3.1.2. Optimized CI Search Algorithm Instead of exhaustively searching CIs from all the access combinations, we use an optimized searching algorithm (Algorithm 1) to improve the efficiency. The search starts by traversing the storage first layer by address. We use T_{adr} to represent a set whose accesses reference the same address. Then, in the second layer, if the number of threads in T_{adr} is less than 2 or the number of accesses in T_{adr} is less than 3, then the accesses in T_{adr} won't form an AV bug. We then skip to the next address. In the third layer, we use T_{tid} to represent a subset of T_{adr} whose accesses are from the same thread. For each (A_i,A_j) pair from two threads, if the accesses of this pair are all reads or all writes, they won't form an unserializable interleaving. We skip to check the next pair. In the fourth layer, if an access pair (a_n,a_m) from A_i and an access b_l from A_j can form an unserializable interleaving, then a CI or an already buggy AVI is identified. We record them for verification. This algorithm improves efficiency by terminating the search for infeasible combinations early, which avoids useless searches.

3.1.3. Complexity Analysis

Time Complexity: Suppose we collect n memory accesses, which are divided into m sets (the second layer) according to the address number (the first layer). Each set is further divided into t subsets (the third layer) according to the thread number. Thus, each subset has $\frac{n}{mt}$ accesses on average. Since we have t threads, they can form $t(t\text{-}1)$ directed thread pairs. For each directed thread pair, the first thread subset can form $\frac{n}{2mt}(\frac{n}{mt}-1)$ local access pairs, and for each local access pair, it can form $\frac{n}{mt}$ interleavings with the accesses in the remote thread subset. Therefore, we totally have to search for $t(t\text{-}1)\frac{n}{2mt}(\frac{n}{mt}-1)\frac{n}{mt} = t(t\text{-}1)\frac{n^2}{2(mt)^2}(\frac{n}{mt}-1)$. The time complexity is $O(t^2(\frac{n}{mt})^3) = O(\frac{n^3}{m^3t})$. However, in practice, the four-layer data structure optimizes the algorithm by terminating the search for infeasible combinations early, which dramatically improves the efficiency.

Space Complexity: The first layer is $O(m)$, the second layer is $O(t)$, the third layer is $O(t)$, and the fourth layer is $O(n)$, thus, the overall space complexity is $O(m+2t+n)$, which is linear.

For a systematic approach that exhaustively searches for a CI from the whole interleaving space, the time complexity is $O(n^3)$ and the space complexity is $O(n)$. Thus, in order to achieve a better complexity tradeoff, our algorithm trades space for time.

DOI: 10.1002/cpe.5160

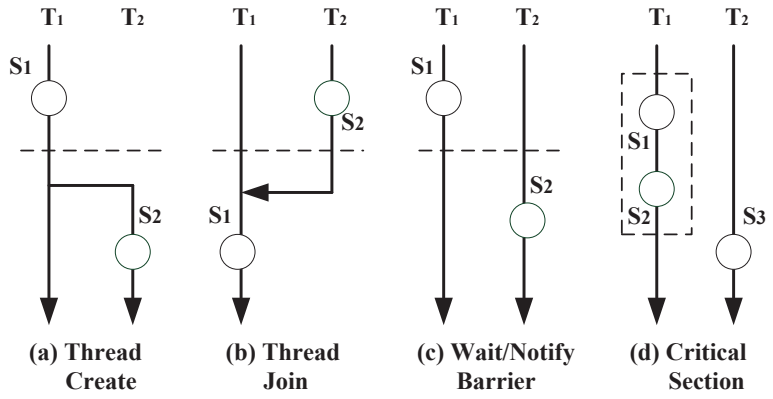

Figure 4. Non-convertible cases owing to the happen-before relations.

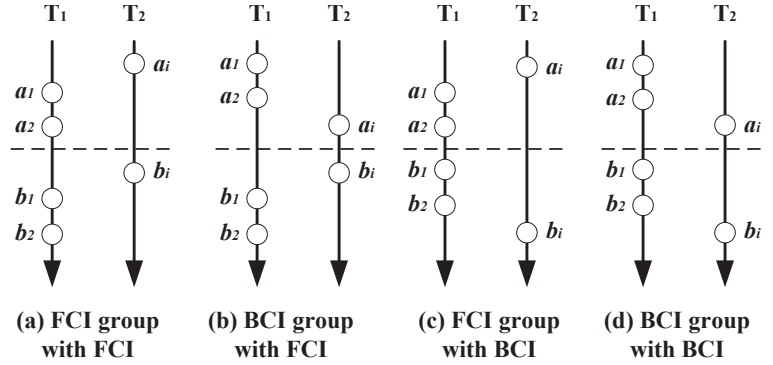

Figure 5. Four combinations of two non-interfere CIs in one group

3.2. *Pruning the Non-convertible CIs*

When converting the CIs to the corresponding AVIs by controlling the thread-scheduling in the controlled executions, some CIs are non-convertible due to the happen-before relations in the program. These happen-before relations enforce the relative ordering of events, which makes changing the event order to realize the desired interleaving impossible. Figure 4 lists 5 happen-before relations that widely exist in concurrent programs. In sub figure (a) and (b), the order of statement S1 in the parent thread and statement S2 in the child thread is enforced by thread create and join. Thus, S1 must execute before S2 in (a), and S2 must execute before S1 in (b). In sub figure (c), the order of statement S1 and S2 is enforced by synchronization primitives, such as the barrier and the wait/notify signal. Thus, S1 must execute before S2. In sub figure (d), statements S1 and S2 in the same critical section cannot be interleaved by a remote statement S3 owing to the locks. Thus, S3 executes either before S1 or after S2.

We identify such happen-before relations from programs and remove non-convertible CIs before verification. Based on the pthread interfaces, we record the thread creations and joins, the locks and unlocks, and the invocations of synchronization primitives into log files during the monitored execution. When the pruner finds an identified CI involved with such happen-before relations, it removes the CI from the queue before the controlled execution. The pruning improves the efficiency by avoiding to execute violation-free interleavings.

3.3. *Grouping the Candidate Interleavings*

To further improve the efficiency, our approach adopts a grouping strategy and verifies a group of potential buggy interleavings in each controlled execution. To group the CIs, we must make sure that the CIs in the same group (possibly have FCIs, BCIs, or both) do not interfere with each other. As Figure 5 shows, given two CIs A (a_1, a_2, a_i) and B (b_1, b_2, b_i), we check whether all the accesses in

DOI: 10.1002/cpe.5160

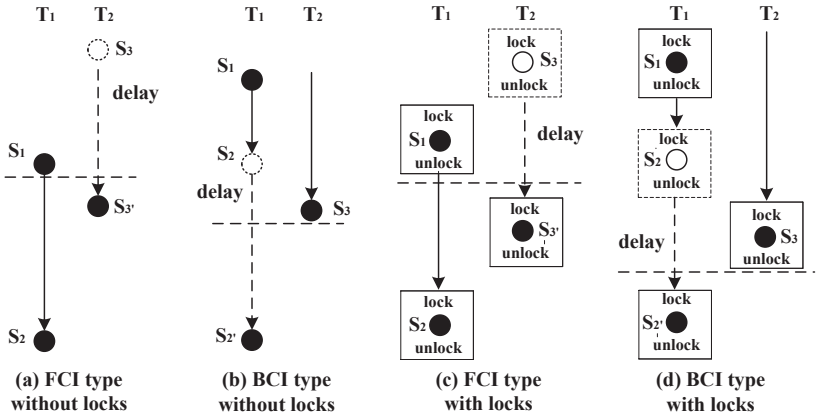

Figure 6. Control the thread-scheduling by inserting delays.

A execute before or after all the accesses in *B* according to the timestamp, if yes, then *A* and *B* do not interfere with each other and can be put into the same group.

We first build a group set $G = \{g_i \mid i = 1, 2, ...\}$, which includes all the existing groups. Then, we traverse all the CIs from the queue generated by the prediction and check whether a CI *ci* does not interfere with all the CIs in any group *g* in *G*. If *g* is found, we insert *ci* into *g*. Otherwise, we create a new group *g'*, insert *ci* into *g'*, and add *g'* into *G*. Besides, the CIs in each group are sorted by the timestamp as they are verified by this order in the controlled execution. In order to balance the number of CIs in each group, we rank the groups by the number of CIs already in the group. The group with the fewest CIs will have the highest priority to receive a new CI.

Suppose we have *n* CIs, when we insert the k^{th} CI into a group, for the worst case, we have to compare it with the *k*-1 CIs that have been previously inserted. Totally, we have to compare the CIs for $1 + 2 + ... + n\text{-}1 = \frac{n(n-1)}{2}$ times, which has a time complexity of $O(n^2)$. However, in practice, we do not need to compare as many CIs as the worst case.

3.4. Controlling the Thread-Scheduling

We verify a predicted AV bug or replay a known AV bug by converting the CI to the corresponding AVI. As Figure 6 shows, the conversion works by controlling the thread-scheduling. For the FCI type, as sub figure (a) shows, we insert delays before the remote access `S3` and wait for the first local access `S1` to execute first. When `S3` is executed, we allow the second local access `S2` to execute. The BCI type is similar, as sub figure (b) shows: we execute `S1` first, then we insert delays before `S2` and wait for `S3` to execute first. Finally, we execute `S2` after the execution of `S3`. We record the thread id (*tid*), the instruction (*ip*), and the memory address (*addr*) of each access, using a three-tuple (*tid*, *ip*, *addr*) to uniquely determine a memory access and precisely locate it in the controlled executions.

When a thread delay is needed, we use a loop to keep inserting time intervals to that thread. The time interval is implemented by invoking the `sleep()` function to block the current thread so as to wait for the desired thread to execute. The loop stops when either the desired instruction in the other thread is executed or the predefined timeout is reached. A timeout implies the CI cannot be converted to the AVI, thus, the CI is non-buggy. The time interval is set to 1ms and the timeout is set to 1s.

However, since memory accesses are usually protected by critical sections formed by locks, when we simply add delays right before the memory accesses, deadlocks can be caused. Thus, for the memory accesses protected by the same lock, we first identify which critical section that each access belongs to by traversing the log, then we move the whole critical section instead of a single memory access (as subfigure (c) and (d) show). We record the thread id (*tid*), the call site value (*cv*), and the entry point value (*ev*) of each lock and unlock call. The call site value can distinguish each lock and unlock call, and the entry point value is used to decide whether two critical sections use the same lock. We use a 3-tuple (*tid*, *cv*, *ev*) to uniquely determine a critical section and precisely relocate it in the controlled executions.

DOI: 10.1002/cpe.5160

Table II. Description of the test cases.

#	Name	Type	Description
1	BankAccount	$R_1W_iW_2$	An atomicity-violation bug caused by simultaneously withdrawal and deposit money.
2	CircularList	$W_1W_iR_2$	Non-atomitically removing and adding work units to the circular list causes a bug.
3	LogProcSweep	$R_1W_iR_2$	An atomicity-violation bug caused by inconsistently manipulating a shared log.
4	Stringbuffer	$R_1W_iR_2$	An atomicity-violation bug caused by using different locks from different class objects.
5	MySQL	$R_1W_iW_2$	Non-atomicity operations of the content and the log in MySQL-1.6.9 causes inconsistency.
6	Apache	$R_1W_iW_2$	Non-atomicity accesses to the server log results in log corruption of Apache Httpd-2.0.48.
7	Mozilla	$W_1W_iR_2$	Atomicity violation in the Mozilla Application Suite causes a null-pointer dereference.
8	Pbzip2	$W_1W_iR_2$	An atomicity-violation bug crashes Pbzip2-0.9.4 when decompressing files.
9	Pfscan	$R_1W_iW_2$	An atomicity-violation bug crashes Pfscan when scanning files.
10	Aget	$W_1R_iW_2$	An atomicity-violation bug crashes Aget-0.4 when stop the download process by ctrl-c.

4. IMPLEMENTATION

We implemented AVPredictor as a prototype tool of our approach. As Figure 2 shows, AVPredictor mainly consists of five components (the shadowed parts): the monitor, the predictor, the pruner, the grouper, and the thread controller. The monitor dynamically records execution traces of the target program into log files. Then the predictor loads the traces from the log files, filters CIs from it, and predicts AVIs. The pruner removes inconvertible cases from the identified CIs. The grouper groups the remained CIs and writes them into different files in the groupset. Finally, the thread controller loads CIs from each group file and launches controlled executions to verify the potential atomicity-violation bugs. The monitor and the thread controller are implemented by C++ based on the Pin instrumentation framework [23], whereas the predictor, the pruner, and the grouper are implemented as offline tools by Python to reduce the runtime overhead. AVPredictor is available at `https://github.com/wpengfei/AVPredictor.git`.

5. EVALUATION

AVPredictor is evaluated with a number of C/C++ programs (as Table II shows). Among them, seven (No. 1 to No. 7) are benchmarks or buggy kernels extracted from real-world programs and three (No. 8 to No. 10) are full utility programs. Each of the test cases has a known AV bug. The experiments are conducted on a machine with a 1.4GHz, 2-core CPU, 8 GB physical memory, running Ubuntu 16.04 and Pin framework 3.7.

5.1. Effectiveness

We test both the compress and decompress modes of Pbzip2, and the results are noted as Pbzip2-com and Pbzip2-dec, respectively. In addition to the ability to detect bugs, we also pay attention to the effectiveness of the pruning and grouping strategies. The results are shown in Table III.

(1) **False reports.** AVPredictor successfully found all the known AV bugs from the test programs. AVPredictor can produce false positives during the prediction process, however, all the false positives are eliminated either by the pruning or the verification in the controlled executions, thus, no false positives remained. Besides, as Column 5 shows, 72.7% (8/11) of the known AV bugs in the test

DOI: 10.1002/cpe.5160

Table III. Effectiveness test results

Name	Original CI Num.	Pruned CI Num.	Group Num.	BI Num.	Pruning Rate	Grouping Rate
BankAccount	2	2	2	2	0	0
CircularList	28	3	1	3	89.3%	66.7%
LogProcSweep	4	2	2	2	50%	0
Stringbuffer	2	1	1	1	50%	0
MySQL	42	39	38	14	7.1%	2.5%
Apache	4	4	4	3	0	0
Mozilla	1	1	1	1	0	0
Pbzip2-com	26	25	22	2	3.8%	12%
Pbzip2-dec	54	52	37	7	3.7%	28.8%
Pfscan	13	5	5	1	61.5%	0
Aget	23	15	8	3	34.8%	46.7%

Column 2 and 3 show the numbers of CIs before and after pruning; Column 4 shows the number of groups generated by the grouper; Column 5 shows the number of buggy interleavings reported; Column 6 shows the percentage of the CIs that are pruned; Column 7 shows the percentage of the executions that saved by the grouping strategy.

programs have more than one buggy interleaving, and AVPredictor could display all the buggy interleavings for the same bug to assist a thorough bug fix.

(2) **A new bug.** In addition to exposing the known AV bugs, AVPredictor also found a new AV bug in the decompress mode of Pbzip2. The already known bug in Pbzip2 is caused by an unserializable access to a lock variable, in which the main thread could delete the lock variable between the lock and unlock operations based on this lock variable in a worker thread [5]. The program crashes (manifests as a segment fault) when the worker thread tries to acquire an already-deleted lock. However, the new bug is caused by an unserializable access to a shared pointer that points to a dynamically allocated buffer. Interleaved modification on this buffer could crash the program (manifests as "`munmap_chunk() invalid pointer`" and a memory dump) when deleting the buffer via this pointer.

(3) **Pruning.** As Column 6 shows, the pruning strategy is successfully applied to four out of the seven extracted kernels with the highest pruning rate of 89.3%; For the utility programs, all of them are pruned with a highest pruning rate of 61.5%. Thus, the pruning strategy is effective and could improve the efficiency by reducing the CIs need to be verified.

(4) **Grouping.** As Column 7 shows, the grouping strategy is effective in two out of the seven extracted kernels. This is because the kernels are relatively small, thus, the CIs tend to interfere with each other. However, the grouping strategy works better for the utility programs, and three out of the four tests are successfully grouped. Nevertheless, the grouping strategy can save up to 66.7% and 46.7% of the verify executions for the extracted kernels and utility programs, respectively, which also improves the efficiency.

5.2. *Efficiency*

For the efficiency tests, we use the full utility programs (Pfscan, Pbzip2, Aget) as well as programs (FFT, RADIX, LU, FMM, OCEAN) from the prevalent Splash2 benchmarks, giving up the previous bug kernels owning to the too short running time. Splash2 benchmarks use multithreading to do the scientific calculations, which are memory access intensive, it also provides precise execution time, which facilitates the measurement.

5.2.1. Runtime Overhead In order to get the precise and reasonable runtime overhead, we ignore the initial setup of the Pin instrumentation and compute the runtime overhead in the following way: $overhead = (Ti - Tb)/Tn$, where Ti is the overall execution time with instrumentation enabled, Tb is the base execution time with Pin but without any instrumentation, and Tn is the native execution time without Pin or any instrumentation.

The runtime overhead of the monitor is mainly introduced by the logging process when writing traces to the log files. We implement the monitor using the fast buffering API of Pin. The fast buffering uses the thread local buffer to store the trace data during the execution and writes out the data when the buffer is

DOI: 10.1002/cpe.5160

Table IV. Runtime overhead evaluation results.

Name	References	Native	Base	Monitor	Controller	Overhead$_M$	Overhead$_C$
Pfscan	3971578k	7.21s	19.09s	80.14s	38.32s	8.5x	2.7x
Pbzip2-com	54k	11.94s	15.68s	17.89s	19.12s	0.2x	0.3x
Pbzip2-dec	34933k	4.24s	6.76s	12.48s	7.35s	1.3x	0.1x
Aget	1k	0.21s	2.01s	2.07s	2.03s	0.3x	0.1x
FFT	628175k	1.59s	3.24s	32.16s	34.35s	18.2x	19.6x
RADIX	20214k	0.08s	1.01s	3.42s	1.68s	30.1x	8.4x
LU	8230582k	7.82s	9.10s	86.72s	423.72s	9.9x	53.0x
FMM	711709k	0.89s	2.28s	32.32s	28.83s	33.7x	29.8x
OCEAN	2574687k	2.31s	3.99s	42.5	119.29s	16.7x	49.9x
Average	-	-	-	-	-	13x	18x

Column 2 (References) provides the memory reference number of each test program, which is used as an indicator of the program size. Column 3 (Native) are the execution time without Pin. Column 4 (Base) shows execution time with Pin but without any instrumentation. Column 5 (Monitor) and Column 6 (Controller) represent the execution time with the monitor and the controller enabled, respectively; Column 7 (Overhead$_M$) and Column 8 (Overhead$_C$) represent the overheads introduced by the monitor and controller, respectively.

full. As the fast buffering calls the `INS_InsertFillBuffer()` API to directly write the data into the buffer, it avoids calling other analysis routines that introduce extra overhead. When writing out the data, we write the whole buffer in binary to the file, which is much faster than converting the traces to text and writing them out line by line. All of the above optimizations significantly reduce the runtime overhead.

For the controller, the runtime overhead is mainly introduced by loading the CIs and controlling the thread-scheduling. When loading the CIs, the controller opens the grouped file and read CIs from the file, which incurs runtime overheads. The controller controls the thread-scheduling by adding time delays before the undesired thread, which inevitably introduces additional overhead.

As Table IV shows, the average runtime overhead of AVPredictor is 13x for the monitoring and 18x for the thread-controlling. The overhead is acceptable for a prototype tool, which is lower than the state-of-the-art approaches such as AVIO [15] (25x for the monitor) and Maple [24] (50x for the profiler and 30x for the active scheduler).

5.2.2. Memory Overhead The monitor incurs memory overhead mainly by the buffers when logging the trace. The memory reference buffer size is set to 4096 pages, which will be dumped into the log file when it is full. Since the lock references and synchronization references are much fewer than the memory references, we use fixed-size arrays to buffer the lock references and synchronization references, and write them out when the execution finishes. Both of the two buffer lengths are set to 100k, and the space consumptions are 2.4MB and 1.6MB, respectively. For the controller, the memory overhead is negligible as it does not need to buffer data in addition to a few local variables.

5.3. Comparison

We compare AVPredictor with the state-of-the-art works in two groups. For the open sourced tools, we conduct the experiments by running them with the same test programs and compare the results directly. Whereas for the tools whose source code is unavailable, we conduct the comparison indirectly based on the already known experiment results with the same test programs.

In the first group, we compared AVPredictor with PCT and Maple. PCT [25] is a random testing technique that provides a probabilistic guarantee in exposing concurrency bugs. Maple [24] is a systematic-testing approach that can detect AV bugs by controlling the thread-scheduling to expose untested interleavings. As Table V shows, both Maple and AVPredictor can expose all the known AV bugs from the test programs, whereas PCT failed to expose four of them before timing out (within 24 hours). Thus, random-based testing without controlling the thread-scheduling is less effective in detecting AV bugs caused by rare buggy interleavings. As for efficiency, AVPredictor exposes the AV bug faster than Maple in each test, and the average speedup is 5.8x. This is benefited from the

DOI: 10.1002/cpe.5160

Table V. Comparison with PCT and Maple.

Name	PCT	Maple	AVPredictor	Speedup
BankAccount	17.4s	10.0s	3.6s	2.8x
CircularList	9.1s	10.6s	2.4s	4.4x
LogProcSweep	Timeout	17.1s	3.5s	4.9x
Stringbuffer	56.4s	12.8s	2.5s	5.1x
MySQL	29s	133.9s	58.8s	2.3x
Apache	24.6s	21.4s	2.4s	3.6x
Mozilla	24.6s	21.4s	2.4s	9x
Pbzip2-com	Timeout	155.1s	42.6s	3.6x
Pbzip2-dec	Timeout	168.4s	64.7s	2.6x
Pfscan	Timeout	326.4s	18.5s	17.6x
Aget	355s	177.4s	23.0s	7.7x

prediction technique we adopt, which covers only the possible buggy interleavings instead of trying to search the whole interleaving space as Maple does.

In the second group, we compare AVpredictor with AVIO and Atom-aid. AVIO [15] is an AV bug detection tool based on an observation called the access interleaving invariant. It detects violations of these invariants at runtime to find AV bugs in the monitored runs. Atom-Aid [26] uses hardware signatures to both detect and survive atomicity violations by preventing their manifestation. For the effectiveness, AVPredictor can find all the known AV bugs from the test programs as AVIO and Atom-Aid did. However, AVIO can only report bugs that manifest in the monitored runs and cannot predict non-exposed bugs, while AVPredictor can both detect exposed bugs and predict non-exposed bugs (e.g., the new bug in Section 5.1). For the efficiency, AVIO introduces 25x runtime overhead for the monitored runs, whereas AVPredictor only introduces 13x runtime overheads for the monitored runs and 18x for the controlled runs. Although the hardware version of AVIO and Atom-Aid introduced lower runtime overhead than AVPredictor, they rely on the support of specific hardware feature.

Therefore, AVPredictor is an effective tool to detect AV bugs with acceptable runtime overheads. It does not rely on the additional support such as training runs or hardware features. It can predict unmanifested AV bugs from a non-buggy execution and display all the buggy interleavings for the same AV bug to assist a thorough bug fix.

6. DISCUSSION

Among different types of concurrency bugs, AV bugs are one of the most significant kind. They are difficult to detect and fix when it can be triggered by different buggy interleavings introduced by the thread-scheduling. When the program is large and complicated, even for a known AV bug, the developer can not imagine all the possible buggy interleavings. This makes the generated patch incomplete and ineffective under certain thread-scheduling (such as the example in Figure 1). As the experiment shows, 72.7% (8/11) of the known AV bugs in the test programs have more than one buggy interleaving. Therefore, comprehensively detecting all the possible buggy interleaving for the same AV bug is important to thoroughly fix the bug.

In this paper, we propose a prediction-based approach to comprehensively detect AV bugs. Our approach can predict unmanifested AV bugs from a non-buggy execution and display all the buggy interleavings for the same AV bug to assist a thorough bug fix. In order to achieve this goal and improve efficiency, our approach adopts three strategies: (1) covering only the potential buggy interleavings by prediction; (2) pruning violation-free interleavings before verification; (3) grouping the CIs for verification. In the future, we can further improve the efficiency by distributing the grouped CIs to different machines and parallelize the verification executions. These strategies successfully reduce the runtime overhead of our approach to an acceptable level of 13x for the monitored execution and 18x for the controlled execution, which is lower than similar works [15,21,24]. As the experiments show, the pruning strategy is applied to 8/11 of the test programs with the highest pruning rate of 89.3%; The grouping

DOI: 10.1002/cpe.5160

strategy is applied to 5/11 of the test programs, saving up to 66.7% of the verification executions. Thus, our prediction-based approach achieves a speedup of 5.8x compared to the systematic approach of Maple.

Our approach currently focuses on the AV bugs that occur between user threads, neglecting special types such as the double-fetch bugs [27, 28] that occur between the kernel and user threads or other concurrency bugs such as the order-violation bugs [7, 29]. In the future, we would take more concurrency bug types into consideration. The modularized design of AVPredictor provides a portable way to integrate new bug types. We only need to add new rules to the analysis module without modifying the monitor and thread controller. Thus, AVPredictor has good scalability. In addition, our approach does not rely on the source code [20], the hardware support [26], or the training runs [15], thus, it is practical for program testing even after software release.

7. RELATED WORK

Various approaches have been proposed to detect atomicity-violation bugs. Stress-testing approaches [5] are easy but less effective due to the low probability of exposing bugs. Different runs in stress-testing tend to cover similar interleavings. Systematic-testing approaches [30–32], such as random-based techniques [25, 33] and the model checking techniques [34–36], are less efficient because the interleaving space is huge and a majority of the covered interleavings are non-buggy.

Active-testing approaches control the thread-scheduling to increase the probability of bug manifestation. Dynamic checkers such as AVIO [15], SVD [21], Atomizer [22], and ATOMFUZZER [3] detect atomicity-violation bugs via monitored runs. However, these dynamic checkers are limited to expose concurrency bugs that manifest in the monitored runs, whereas failing to expose the unmonitored ones. AVPredictor can detect exposed bugs and predict non-exposed bugs. Besides, AVPredictor does not rely on training runs as AVIO, does not produce large numbers of false reports as Atomizer, and does not introduce severe runtime overhead as SVD.

Prediction-based active testing imporves bug manifestation by predictive algorithms. Maple [24] and CTrigger [5] use profiling runs to collect traces and use controlled runs to expose atomicity-violation bugs, which is similar to our approach. However, they control the thread-scheduling to exercise the low-probability interleavings instead of to cover the potential buggy interleavings as AVPredictor does. Thus, the buggy-exposing efficiency is still low (demonstrated in Section 5.3). Besides, AVPredictor further improves the efficiency by a grouping strategy.

PECAN [6] uses the predictive trace analysis for detecting general access anomalies in concurrent Java programs. It predicts access anomalies and generates “bug hatching clips” that deterministically instruct the input program to exercise the predicted access anomalies. However, a majority of such access anomalies are non-buggy. CCI [37] detects concurrency bugs by tracking specific thread interleavings at runtime and using statistical models to identify strong failure predictors among them. It uses the random sampling strategies to lower the runtime overhead, which inevitably reduces the interleaving coverage, and misses potential bugs. Prediction-based active testing techniques are also used to detect other types of concurrency bugs, such as data races [14] and deadlocks [38] [39]. However, none of these approaches pays attention to the atomicity-violation characteristics. AVPredictor is specific to the AV bugs and can expose the different buggy interleavings of the same bug to assist a thorough fix.

In addition, some inspiring works use hardware feature to detect AV bugs. Atom-Aid [26] uses “implicit atomicity" provided by the hardware chunk to both detect and survive atomicity violations by preventing their manifestation. Colorsafe [20] dynamically detects and avoids atomicity violations by grouping related data into colors and monitoring access interleavings in the “color space" via transactions. However, the implementation of such approaches relies on additional supports, such as the hardware feature, the availability of the source code, or the prior manual efforts, thus, the practicability is limited. AVPredictor is free from such limitations.

DOI: 10.1002/cpe.5160

8. CONCLUSION

The presented prediction-based approach to comprehensively detect atomicity-violation bugs can predict unmanifested atomicity-violation bugs from a non-buggy execution and display all the buggy interleavings for the same atomicity-violation bug to assist a thorough bug fix. A prototype named AVPredictor is implemented and evaluated. Experiments show that 72.7% of the test programs have more than one buggy interleaving and AVPredictor could effectively find all the known atomicity-violation bugs as well as a previously unknown bug together with all the buggy interleavings. The runtime overhead is 13x for the monitored execution and 18x for the controlled execution.

ACKNOWLEDGEMENTS

This work is partially supported by the National Key Research and Development Program of China (2016YFB0200401), the Program for New Century Excellent Talents in University, the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013), the HUNAN Province Science Foundation (2017RS3045), and the National Science Foundation China (61472437).

REFERENCES

1. Leveson NG, Turner CS. An investigation of the therac-25 accidents. *Computer* 1993; **26**(7):18–41.
2. Jesdanun A. General electric acknowledges northeastern blackout bug 2004. URL http://www.securityfocus.com/news/8032.
3. Park CS, Sen K. Randomized active atomicity violation detection in concurrent programs. *Foundations of Software Engineering*, 2008.
4. Lai Z. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. *Intl. Conf. on Software Engineering*, 2011.
5. Lu S, Park S, Zhou Y. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering* 2012; **38**(4).
6. Huang J, Zhang C. Persuasive prediction of concurrency access anomalies. *Intl. Symposium on Software Testing and Analysis*, 2011.
7. Lu S, Park S, Seo E, Zhou Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *Architectural Support for Programming Languages and Operating Systems*, 2008.
8. Voung JW, Jhala R, Lerner S. Relay: static race detection on millions of lines of code. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.
9. Pratikakis P, Foster JS, Hicks M. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems* 2011; **33**(1).
10. Chen J, MacDonald S. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, 2008.
11. Engler D, Ashcraft K. RacerX: effective, static detection of race conditions and deadlocks. *ACM Symposium on Operating Systems Principles*, 2003.
12. Sen K. Race directed random testing of concurrent programs. *Programming Language Design and Implementation*, 2008.
13. Kasikci B, Zamfir C, Candea G. RaceMob: crowdsourced data race detection. *ACM Symposium on Operating Systems Principles*, 2013.
14. Lu K, Wu Z, Wang X, Chen C, Zhou X. RaceChecker: Efficient identification of harmful data races. *Intl. Conf. on Parallel, Distributed and Network-Based Processing*, 2015.
15. Lu S, Tucek J, Qin F, Zhou Y. AVIO: detecting atomicity violations via access interleaving invariants. *Architectural Support for Programming Languages and Operating Systems*, 2006.
16. Wu Z, Lu K, Wang X, Zhou X, Chen C. Detecting harmful data races through parallel verification. *The Journal of Supercomputing* 2015; **71**(8).
17. Narayanasamy S, Wang Z, Tigani J, Edwards A, Calder B. Automatically classifying benign and harmful data races using replay analysis. *ACM SIGPLAN Notices*, vol. 42, 2007.
18. Kasikci BCC, Zamfir C, Candea G. Data races vs. data race bugs: Telling the difference with portend. *Asplos Xvii: Seventeenth Intl. Conf. on Architectural Support For Programming Languages And Operating Systems*, EPFL-CONF-173730, 2012.
19. Volos H, Tack AJ, Swift MM, Lu S. Applying transactional memory to concurrency bugs. *ACM SIGPLAN Notices*, vol. 47, 2012.
20. Lucia B, Ceze L, Strauss K. ColorSafe: architectural support for debugging and dynamically avoiding multi-variable atomicity violations. *Intl. Symposium on Computer Architecture*, 2010.
21. Xu M, Bodík R, Hill MD. A serializability violation detector for shared-memory server programs. *Programming Language Design and Implementation*, 2005.
22. Flanagan C, Freund SN. Atomizer: a dynamic atomicity checker for multithreaded programs. *Symposium on Principles of Programming Languages*, 2004.
23. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. *Programming Language Design and Implementation*, 2005.

DOI: 10.1002/cpe.5160

24. Yu J, Narayanasamy S, Pereira C, Pokam G. Maple: a coverage-driven testing tool for multithreaded programs. *Object Oriented Programming Systems Languages and Applications*, 2012.
25. Burckhardt S, Kothari P, Musuvathi M, Nagarakatte S. A randomized scheduler with probabilistic guarantees of finding bugs. *Architectural Support for Programming Languages and Operating Systems*, 2010.
26. Lucia B, Devietti J, Strauss K, Ceze L. Atom-aid: Detecting and surviving atomicity violations. *Intl. Symposium on Computer Architecture*, 2008.
27. Wang P, Krinke J, Lu K, Li G, Dodier-Lazaro S. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. *Usenix Security Symposium*, 2017.
28. Wang P, Lu K, Li G, Zhou X. A survey of the double-fetch vulnerabilities. *Concurrency and Computation Practice and Experience* 2017; (8).
29. Shi Y, Park S, Yin Z, Lu S, Zhou Y, Chen W, Zheng W. Do i use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. *ACM Sigplan Notices*, vol. 45, ACM, 2010; 160–174.
30. Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. *Programming Language Design and Implementation*, 2007.
31. Wang C, Said M, Gupta A. Coverage guided systematic concurrency testing. *Intl. Conf. on Software Engineering*, 2011.
32. Musuvathi M, Qadeer S, Ball T, Basler G, Nainar PA, Neamtiu I. Finding and reproducing heisenbugs in concurrent programs. *Usenix Conf. on Operating Systems Design and Implementation*, 2008.
33. Kidd N, Reps T, Dolby J, Vaziri M. Finding concurrency-related bugs using random isolation. *Verification, Model Checking, and Abstract Interpretation*, 2009.
34. Flanagan C. Verifying commit-atomicity using model-checking. *Intl. SPIN Workshop on Model Checking of Software*, 2004.
35. Hatcliff J, Robby, Dwyer MB. Verifying atomicity specifications for concurrent object-oriented software using model-checking. *Verification, Model Checking, and Abstract Interpretation*, 2004.
36. Godefroid P. Model checking for programming languages using verisoft. 1997.
37. Jin G, Thakur A, Liblit B, Lu S. Instrumentation and sampling strategies for cooperative concurrency bug isolation. *ACM Sigplan Notices*, vol. 45, ACM, 2010; 241–255.
38. Cai Y, Chan W. Magicfuzzer: scalable deadlock detection for large-scale applications. *Proceedings of the 34th International Conference on Software Engineering*, IEEE Press, 2012; 606–616.
39. Cai Y, Lu Q. Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering* 2016; **42**(9):825–842.

A. RUNTIME OUTPUT OF AVPREDICTOR

This section provides the runtime output information of AVPredictor. The following information is only available in the debug model (turn on the DEBUG macro) as printing such information slows down the execution.

DOI: 10.1002/cpe.5160

Table VI. Monitor Output

```
[ThreadStart] Thread 0 created, total number is 0
[Thread−−Lock] Tid: 0, time: 0xcb97128a1939, lock_callsite_v: 0xb44e3bb6, lock_entry_v: 0xb7fff4e4.
[ThreadUnLock] Tid: 0, time: 0xcb9713b856f6, unlock_callsite_v: 0xb44e3d59, unlock_entry_v: 0xb7fff4e4.
[ThreadStart] Thread 1 created, total number is 1
[ThreadStart] Thread 2 created, total number is 2
[ThreadStart] start logging
[Thread−−Lock] Tid: 2, time: 0xcb97315eab30, lock_callsite_v: 0x8048757, lock_entry_v: 0x804fa14.
[Thread−−Lock] Tid: 1, time: 0xcb9731932042, lock_callsite_v: 0x8048757, lock_entry_v: 0x804fa14.
[ThreadUnLock] Tid: 2, time: 0xcb97320f8804, unlock_callsite_v: 0x8048771, unlock_entry_v: 0x804fa14.
[Thread−−Lock] Tid: 2, time: 0xcb97322f7d60, lock_callsite_v: 0x804878e, lock_entry_v: 0x804fa14.
[ThreadUnLock] Tid: 2, time: 0xcb97325182c0, unlock_callsite_v: 0x80487a8, unlock_entry_v: 0x804fa14.
[ThreadUnLock] Tid: 1, time: 0xcb97327e7511, unlock_callsite_v: 0x8048771, unlock_entry_v: 0x804fa14.
[Thread−−Lock] Tid: 1, time: 0xcb9734cb99a4, lock_callsite_v: 0x804878e, lock_entry_v: 0x804fa14.
[ThreadUnLock] Tid: 1, time: 0xcb9734d0d661, unlock_callsite_v: 0x80487a8, unlock_entry_v: 0x804fa14.
[BufferFull]pc:0x804889d,addr:0x804a044,time:0xcb972c3cf131,tid:0x2,read:0
[BufferFull]pc:0x804875d,addr:0x804fa10,time:0xcb9731c8e311,tid:0x2,read:0
[BufferFull]pc:0x8048797,addr:0x804fa10,time:0xcb97324999d4,tid:0x2,read:1
[ThreadFini] Thread 2 joined, total number is 2
[BufferFull]pc:0x8048850,addr:0x804a044,time:0xcb9724214b03,tid:0x1,read:0
[BufferFull]pc:0x8048866,addr:0x804a044,time:0xcb97317dc0fd,tid:0x1,read:0
[BufferFull]pc:0x804875d,addr:0x804fa10,time:0xcb97327e6ee2,tid:0x1,read:0
[BufferFull]pc:0x8048797,addr:0x804fa10,time:0xcb9734d0d018,tid:0x1,read:1
[ThreadFini] Thread 1 joined, total number is 1
[ThreadFini] stop logging.
[Thread−−Lock] Tid: 0, time:0xcb973b63eb24, lock_callsite_v: 0xb7fea93d, lock_entry_v: 0xb7fff4e4.
[ThreadUnLock] Tid: 0, time:0xcb973cdac2c9, unlock_callsite_v: 0xb7feaa03, unlock_entry_v: 0xb7fff4e4.
[BufferFull]pc:0x804853f,addr:0x8049ffc,time:0xcb971cdf24fd,tid:0x0,read:0
[BufferFull]pc:0x8048715,addr:0x8049f00,time:0xcb971cfd513d,tid:0x0,read:0
[BufferFull]pc:0x8048827,addr:0x804fa10,time:0xcb971ece9099,tid:0x0,read:1
[BufferFull]pc:0x8048952,addr:0x804fa30,time:0xcb972a5d7ae8,tid:0x0,read:0
[ThreadFini] Main Thread joined, total number is 0
countMemRef = 11
countLockRef = 12
countSyncRef = 0
```


DOI: 10.1002/cpe.5160

Table VII. Predictor Output

```
>>>>>>>>>>>>>>>>>>[loaded mem_trace]<<<<<<<<<<<<<<<<<<<<<<<
addr: 134520576 thread_num 1 tid_list [0]
=>tid 0 optag 1
{'tid':0, 'ip':134514453, 'op':0, 'addr':134520576, 'time':300854382745500L}
addr: 134543920 thread_num 1 tid_list [0]
=>tid 0 optag 1
{'tid':0, 'ip':134515026, 'op':0, 'addr':134543920, 'time':300854593387006L}
addr: 134520900 thread_num 2 tid_list [1, 2]
=>tid 1 optag 1
{'tid':1, 'ip':134514768, 'op':0, 'addr':134520900, 'time':300854503377701L}
{'tid':1, 'ip':134514790, 'op':0, 'addr':134520900, 'time':300854731142805L}
=>tid 2 optag 1
{'tid':2, 'ip':134514845, 'op':0, 'addr':134520900, 'time':300854623855132L}
addr: 134543888 thread_num 3 tid_list [1, 2, 0]
=>tid 0 optag 0
{'tid':0, 'ip':134514727, 'op':1, 'addr':134543888, 'time':300854414244820L}
=>tid 1 optag 2
{'tid':1, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854741888865L}
{'tid':1, 'ip':134514583, 'op':1, 'addr':134543888, 'time':300854761068501L}
=>tid 2 optag 2
{'tid':2, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854770372774L}
{'tid':2, 'ip':134514583, 'op':1, 'addr':134543888, 'time':300854863837481L}
addr: 134543924 thread_num 1 tid_list [0]
=>tid 0 optag 1
{'tid':0, 'ip':134515048, 'op':0, 'addr':134543924, 'time':300854880252394L}
addr: 134520828 thread_num 1 tid_list [0]
=>tid 0 optag 1
{'tid':0, 'ip':134513983, 'op':0, 'addr':134520828, 'time':300854379934800L}
>>>>>>>>>>>>>>>>>>[loaded cs list]<<<<<<<<<<<<<<<<<<<<<<<
{'l_cv':3021282230L, 'l_ev':3087004900L, 'ftime':300854231871793L, 'stime':
300854216155510L, 'tid': 0, 'u_cv': 3021282649L, 'u_ev': 3087004900L}
{'l_cv': 134514519, 'l_ev': 134543892, 'ftime': 300854751945481L, 'stime':
 300854739271508L, 'tid': 1, 'u_cv': 134514545, 'u_ev': 134543892}
{'l_cv': 134514574, 'l_ev': 134543892, 'ftime': 300854763368480L, 'stime':
 300854755321374L, 'tid': 1, 'u_cv': 134514600, 'u_ev': 134543892}
{'l_cv': 134514519, 'l_ev': 134543892, 'ftime': 300854770378229L, 'stime':
 300854758009206L, 'tid': 2, 'u_cv': 134514545, 'u_ev': 134543892}
{'l_cv': 134514574, 'l_ev': 134543892, 'ftime': 300854863838852L, 'stime':
 300854863715532L, 'tid': 2, 'u_cv': 134514600, 'u_ev': 134543892}
{'l_cv':3086919997L, 'l_ev':3087004900L, 'ftime':300854937005159L, 'stime':
300854908579384L, 'tid': 0, 'u_cv': 3086920195L, 'u_ev': 3087004900L}
>>>>>>>>>>>>>>>>>>[calculated CI list]<<<<<<<<<<<<<<<<<<<<<<<
first: {'tid':1, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854741888865L}
second: {'tid':1, 'ip':134514583, 'op':1, 'addr':134543888, 'time':300854761068501L}
inter: {'tid':2, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854770372774L}
-----------------------
first: {'tid':2, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854770372774L}
second: {'tid':2, 'ip':134514583, 'op':1, 'addr':134543888, 'time':300854863837481L}
inter: {'tid':1, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854741888865L}
```


DOI: 10.1002/cpe.5160

Table VIII. Pruner and Grouper Output

```
>>>>>>>>>>>>>>>>>>[result list after prunning]<<<<<<<<<<<<<<<<<<<<<<<
first access: {'tid':1, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854741888865L}
second access: {'tid':1, 'ip':134514583, 'op':1, 'addr':134543888, 'time':300854761068501L}
inter access: {'tid':2, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854770372774L}
first_cs: {'l_cv': 134514519, 'l_ev': 134543892, 'ftime': 300854751945481L,
 'stime': 300854739271508L, 'tid': 1, 'u_cv': 134514545, 'u_ev': 134543892}
second_cs: {'l_cv': 134514574, 'l_ev': 134543892, 'ftime': 300854763368480L,
 'stime': 300854755321374L, 'tid': 1, 'u_cv': 134514600, 'u_ev': 134543892}
inter_cs: {'l_cv': 134514519, 'l_ev': 134543892, 'ftime': 300854770378229L,
 'stime': 300854758009206L, 'tid': 2, 'u_cv': 134514545, 'u_ev': 134543892}
-----------------------
first access: {'tid':2, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854770372774L}
second access: {'tid':2, 'ip':134514583, 'op':1, 'addr':134543888, 'time':300854863837481L}
inter access: {'tid':1, 'ip':134514525, 'op':0, 'addr':134543888, 'time':300854741888865L}
first_cs: {'l_cv': 134514519, 'l_ev': 134543892, 'ftime': 300854770378229L,
 'stime': 300854758009206L, 'tid': 2, 'u_cv': 134514545, 'u_ev': 134543892}
second_cs: {'l_cv': 134514574, 'l_ev': 134543892, 'ftime': 300854863838852L,
 'stime': 300854863715532L, 'tid': 2, 'u_cv': 134514600, 'u_ev': 134543892}
inter_cs: {'l_cv': 134514519, 'l_ev': 134543892, 'ftime': 300854751945481L,
 'stime': 300854739271508L, 'tid': 1, 'u_cv': 134514545, 'u_ev': 134543892}
-----------------------
>>>>>>>>>>>>>>>>>>[Grouping]<<<<<<<<<<<<<<<<<<<<<<<
sorted lists in group 0 :
{'first_interval': [300854739271508L, 300854751945481L],
 'second_interval': [300854755321374L, 300854763368480L],
 'inter_interval': [300854758009206L, 300854770378229L],
 'msg': '1[1]134514519,134514545; [1]134514574,134514600; [2]134514519,134514545\n'}
sorted lists in group 1 :
{'first_interval': [300854758009206L, 300854770378229L],
 'second_interval': [300854863715532L, 300854863838852L],
 'inter_interval': [300854739271508L, 300854751945481L],
 'msg': '1[2]134514519,134514545; [2]134514574,134514600; [1]134514519,134514545\n'}
>>>>>>>>>>>>>>>>>>[statistic result]<<<<<<<<<<<<<<<<<<<<<<<
total accesses: 12
visited addresses num: 6
total CI num: 2
result CI num after prunning: 2
group num: 2
```


DOI: 10.1002/cpe.5160

Table IX. Controller Output

```
work_dir/groupset/group_0.log
=========================Load CI 0 ==========================
[load CI from file]:1[2]0x8048757,0x8048771; [2]0x804878e,0x80487a8; [1]8048757,8048771
Loaded 1 CI(s) for replay
[beforeThreadLock][enter First] callsite_v:0x8048757, tid:2
[beforeThreadLock][delay Inter] callsite_v:0x8048757, tid:1
[beforeThreadUnLock][executed First] callsite_v:0x8048771, tid:2
[beforeThreadLock][delay Second] callsite_v:0x804878e, tid:2
[beforeThreadLock][enter Inter after delay] callsite_v:0x8048757, tid:1
[beforeThreadUnLock][executed Inter] callsite_v:0x8048771, tid:1
[beforeThreadLock][enter Second after delay] callsite_v:0x804878e, tid:2
[beforeThreadUnLock][executed Second] callsite_v:0x80487a8, tid:2

main_bank_lock: main.cc:103: int main(int, char**):
Assertion 'account->balance == 0' failed.
./start_test.sh: line 85: 16702 Aborted (core dumped)
  pin-3.7-97619-gcc-linux/pin -t work_dir/obj-ia32/controller.so --
test_dir/main_bank_lock

work_dir/groupset/group_1.log
=========================Load CI 0 ==========================
[load CI from file]:1[1]0x8048757,0x8048771; [1]0x804878e,0x80487a8; [2]8048757,8048771
Loaded 1 CI(s) for replay
[beforeThreadLock][delay Inter] callsite_v:0x8048757, tid:2
[beforeThreadLock][enter First] callsite_v:0x8048757, tid:1
[beforeThreadUnLock][executed First] callsite_v:0x8048771, tid:1
[beforeThreadLock][enter Inter after delay] callsite_v:0x8048757, tid:2
[beforeThreadLock][delay Second] callsite_v:0x804878e, tid:1
[beforeThreadUnLock][executed Inter] callsite_v:0x8048771, tid:2
[beforeThreadLock][enter Second after delay] callsite_v:0x804878e, tid:1
[beforeThreadUnLock][executed Second] callsite_v:0x80487a8, tid:1

main_bank_lock: main.cc:103: int main(int, char**):
Assertion 'account->balance == 0' failed.
./start_test.sh: line 85: 17408 Aborted (core dumped)
pin-3.7-97619-gcc-linux/pin -t work_dir/obj-ia32/controller.so --
test_dir/main_bank_lock
```


DOI: 10.1002/cpe.5160