2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs

Qi Gao [*]
Facebook Inc.
gao@fb.com

Wenbin Zhang
The Ohio State University
zhangwen@cse.ohio-state.edu

Zhezhe Chen
The Ohio State University
chenzhe@cse.ohio-state.edu

Mai Zheng
The Ohio State University
zhengm@cse.ohio-state.edu

Feng Qin
The Ohio State University
qin@cse.ohio-state.edu

Abstract

Concurrency bugs are becoming increasingly prevalent in the multi-core era. Recently, much research has focused on data races and atomicity violation bugs, which are related to low-level memory accesses. However, a large number of concurrency typestate bugs such as "invalid reads to a closed file from a different thread" are under-studied. These concurrency typestate bugs are important yet challenging to study since they are mostly relevant to high-level program semantics.

This paper presents 2ndStrike, a method to manifest hidden concurrency typestate bugs in software testing. Given a state machine describing correct program behavior on certain object typestates, 2ndStrike profiles runtime events related to the typestates and thread synchronization. Based on the profiling results, 2ndStrike then identifies bug candidates, each of which is a pair of runtime events that would cause typestate violation if the event order is reversed. Finally, 2ndStrike re-executes the program with controlled thread interleaving to manifest bug candidates.

We have implemented a prototype of 2ndStrike on Linux and have illustrated our idea using three types of concurrency typestate bugs, including invalid file operation, invalid pointer dereference, and invalid lock operation. We have evaluated 2ndStrike with six real world bugs (including one previously unknown bug) from three open-source server and desktop programs (i.e., MySQL, Mozilla, pbzip2). Our experimental results show that 2ndStrike can effectively and efficiently manifest all six software bugs, most of which are difficult or impossible to manifest using stress testing or active testing techniques that are based on data race/atomicity violation. Additionally, 2ndStrike reports no false positives, provides detailed bug reports for each manifested bug, and can consistently reproduce the bug after manifesting it once.

[*] This work was done while the author was a Ph.D. student at The Ohio State University

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Testing Tools

General Terms Reliability

Keywords Software Testing, Typestate Bugs, Concurrency Bugs

1. Introduction

Concurrency bugs are becoming increasingly prevalent in the multi-core era as more programs are written or rewritten in a multi-threaded fashion for better utilizing multi-core systems. However, it is extremely difficult for developers to reason about all possible orders of accessing shared objects from multiple threads. This is especially true for multi-threaded programs with complex semantics. Consequently, concurrency bugs hidden in corner cases are inevitable and cause runtime errors.

Unfortunately, concurrency bugs are notoriously difficult to pinpoint in testing due to their non-deterministic nature. The *de facto* way to expose and detect concurrency bugs is stress testing, i.e., running a multi-threaded program repetitively for a long time with test inputs. While easy to conduct, stress testing is inefficient due to large consumption of computing resources. Furthermore, stress testing is usually ineffective in exposing concurrency bugs because it exercises only a fraction of possible thread interleavings [39, 47].

Much research has been conducted on detecting two types of concurrency bugs, i.e., data races [14, 56, 72] and atomicity violations [24, 33, 67]. These approaches focus on detecting memory accesses to shared variables that are not protected by common locks or not serializable under exercised thread interleavings. While these methods are effective in detecting concurrency bugs that are triggered, they cannot handle concurrency bugs hidden in un-exercised thread interleavings. To address this issue, several active testing techniques [46, 47, 57] are recently proposed. Based on information from external detectors or runtime profilers, these testing techniques actively control the thread scheduling to favor the interleavings that may trigger data races [57] or atomicity violations [46, 47]. As a result, these tools help expose data races or atomicity violations hidden in uncommon thread interleavings.

However, previous studies have shown that large system programs contain a significant number of concurrency bugs that involve neither data races nor atomicity violations [36]. Figure 1 shows a real-world concurrency bug of this type in MySQL, a popular database server [1]. In this example, Thread 1 reads data from a file at statement A and Thread 2 closes the same file at statement B. As shown in Figure 1(a), the correct thread interleaving

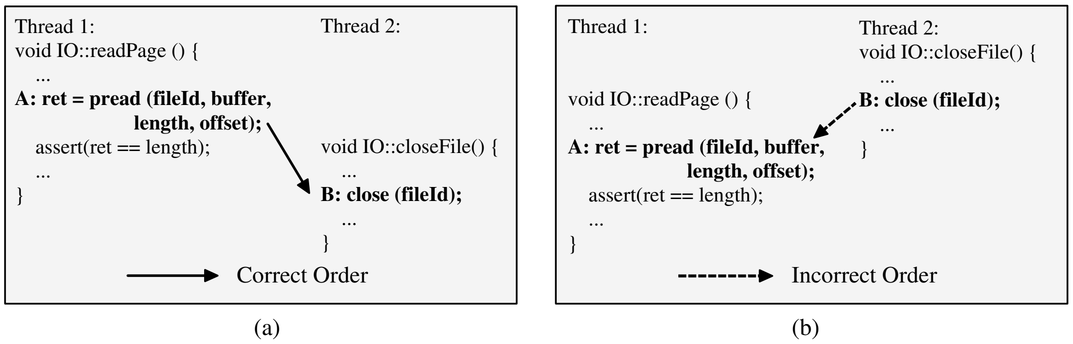

Figure 1. A real-world concurrency typestate bug extracted from MySQL

is reading the file from Thread 1 followed by closing the file from Thread 2, which is the common case for program execution. However, due to improper synchronization, Thread 2 can potentially, although rarely, close the file before Thread 1 accesses the same file, which is shown in Figure 1(b). This synchronization error violates file semantics instead of causing data races or atomicity violation.

This type of concurrency bugs are related to high-level program semantics, and can be captured by *typestates* [62]. A typestate is the state associated to an object of a given type (e.g., a file descriptor, a lock, or a memory object). At each typestate, an object is permitted to be applied with a subset of operations. Among the permitted operations, some may alter the typestate of the object, while others preserve the typestate of the object. *Concurrency typestate bugs* occur when typestate-altering operations from one thread are not well synchronized with other operations from different threads, causing the object being applied with un-permitted operations in some typestate, i.e., *typestate violation*. Figure 1(b) demonstrates such concurrency typestate bug. The operation `close` in Thread 2 changes the typestate of the file object to `CLOSED`. Then, the operation `pread` in Thread 1 is performed in the same file object with the typestate of `CLOSED`, which is not permitted.

Concurrency typestate bugs are important yet challenging to address. Similar to typestate bugs in sequential programs, which have been studied extensively [6, 7, 11, 12, 17, 18, 25, 62], concurrency typestate bugs are common yet under-studied mistakes in large multi-threaded system programs [36]. In these programs, many stateful objects, such as connections for Internet servers and tables for databases, are shared among multiple threads. With complex business logic, it is easy for programmers to make mistakes in reasoning about object states with thread synchronization, often leading to concurrency typestate bugs. Furthermore, existing approaches that detect or expose data races or atomicity violations [14, 24, 33, 46, 47, 56, 57, 67, 72] cannot deal with this type of bugs. The main reason is that these methods focus on the synchronization at the memory access level without taking program semantics into consideration. Even if data races or atomicity violations may co-exist in some concurrency typestate bugs, existing methods can provide few insights for understanding the root causes of bugs due to lack of high-level program semantics. Recently-proposed approaches Falcon [49] and Bugaboo [37] can detect order violation bugs, which may be relevant to concurrency typestates. However, these approaches focus on detecting instead of exposing concurrency bugs.

Our Contributions: In this paper, we propose a general and extensible testing framework called *2ndStrike* to manifest potential concurrency typestate bugs hidden in multi-threaded programs. By using user-defined state machines to capture semantic-related typestate information in the program, 2ndStrike performs runtime profiling specific to typestate properties. Based on the profiling results, 2ndStrike identifies the potential synchronization errors that could lead to typestate violation. Then 2ndStrike re-executes the program with controlled thread scheduling for manifesting the potential errors. This work flow can be fully automated and easily integrated with existing test harnesses.

The brief process of applying 2ndStrike to the previous bug example goes as follows. First, 2ndStrike profiles runtime events that are related to file operations and thread synchronization. The event orders in the log are most likely generated by common scenarios as shown in Figure 1(a). By analyzing the log, 2ndStrike identifies the pair of events, i.e., `pread` from Thread 1 and `close` from Thread 2, as a candidate of potential typestate violation. This is because `pread` from Thread 1 is not permitted to be applied to a file object with the typestate of `CLOSED`, which is the result of applying `close` from Thread 2 to the same file object with the typestate of `OPEN`. Then, 2ndStrike attempts to manifest this potential concurrency typestate bug by re-executing the program with a different thread scheduling, i.e., `close` from Thread 2 is scheduled before `pread` from Thread 1.

Based on the above idea, we have implemented a prototype of 2ndStrike on Linux for manifesting three common types of concurrency typestate bugs, including invalid file operation, invalid pointer dereference, and invalid lock operation. We chose these three typestate properties for illustrating the idea since they are easy to understand without requiring much domain knowledge. We do not see any particular technical difficulties to extend 2ndStrike for application-specific typestates.

We have evaluated 2ndStrike with six real-world concurrency typestate bugs of three types from three open-source server and desktop programs, including MySQL [1], Mozilla [2], and pbzip2 [3]. Our experimental results show that 2ndStrike can effectively and efficiently manifest all six concurrency typestate bugs. Additionally, it provides detailed bug reports and can consistently reproduce the bugs after manifesting them once. Compared to existing approaches, 2ndStrike has one or more of the following advantages:

- 2ndStrike is effective. To the best of our knowledge, 2ndStrike is the first offline testing method to *manifest concurrency typestate bugs* by exploiting semantics-related information and thread scheduling. Our experimental results show that it can manifest all six tested concurrency typestate bugs. These bugs are difficult to trigger by stress testing and five of them will be missed by data race or atomicity violation directed active testing techniques [47, 57].
- 2ndStrike is efficient. With typestate specific instrumentation, users can instrument a small set of semantics-related runtime events. This incurs much smaller runtime overhead than many existing tools that instrument every memory accesses (by a

factor of 3-50). Furthermore, 2ndStrike's stand-alone analyzer runs in parallel with the program being tested and analyzes the runtime logs on-the-fly without saving them to disk, which reduces both storage and runtime overhead.

- 2ndStrike is extensible. 2ndStrike is a general testing framework to perform typestate specific profiling, analysis, and scheduling control for manifesting concurrency typestate bugs. As examples, three 2ndStrike modules have been built for manifesting three common types of concurrency typestate bugs.
- 2ndStrike is easy to use. It does not require source code changes and can be easily integrated with existing test harnesses packaged with software programs. Additionally, it does not report false positives. Moreover, it provides reproducibility and bug reports to facilitate developers to diagnose the manifested bugs.

The rest of this paper is organized as follows. Section 2 presents the background of concurrency typestate bugs and compares them with data race/atomicity violation bugs. Section 3 provides an overview of 2ndStrike, followed by detailed description on each step in the work flow. Section 4 discusses design issues. After that, Section 5 presents the evaluation methodology, followed by the experimental results in Section 6. Then Section 7 discusses related work and Section 8 concludes the paper.

2. Concurrency Typestate Bugs

2.1 Typestates, State Machines, and Concurrency Typestate Bugs

First introduced in [62], typestate is a temporal extension of the concepts of types in programming languages. Each type has an associated set of typestates. At each program point, an object of a given type is in one of the typestates associated with its type. In essence, typestate determines a set of operations that are permitted to be applied to the corresponding objects.

In 2ndStrike, we use a finite state machine to model typestates for objects of a given type, defining which operations are permitted to be applied to the objects in which typestates. There are several ways to provide state machines for a object type. Generally, programmers can provide such state machines since they know high-level program semantics better than others. To reduce programmers' manual effort, researchers have proposed static methods [59] as well as dynamic methods [5] for automatically inferring the likely typestate property of objects. Furthermore, we can derive certain state machines based on common programming rules, such as "cannot read on a closed file handler" and "cannot dereference an invalid pointer".

Figure 2(a), (b), and (c) show the state machines for objects of the types as files, pointers, and locks, respectively. As shown in Figure 2(a), a file has two states: `OPEN` and `CLOSED`. A file in `OPEN` state indicates that its content is ready for being accessed, while the file content cannot be accessed when it is in `CLOSED` state. More specifically, the operations such as `Read`, `Write`, `Seek`, and `Close` are permitted on the file in `OPEN` state, and the operation `Open` is permitted on the `CLOSED` state. Among these operations, `Read`, `Write`, and `Seek` do not change the file's typestate, while `Open` changes the file state to `OPEN` and `Close` changes the file state to `CLOSED`. Similarly, Figure 2(b) and (c) show the state machines for pointers and locks, respectively.

A *typestate violation* occurs when a non-permitted operation is applied to an object in certain states. For example, when the operation `Read` is performed on a file object with the state of `CLOSED`, it incurs a typestate violation.

In this paper, a *concurrency typestate bug* refers to a synchronization error among multiple threads where one thread applies a state-changing operation to an object followed by another thread

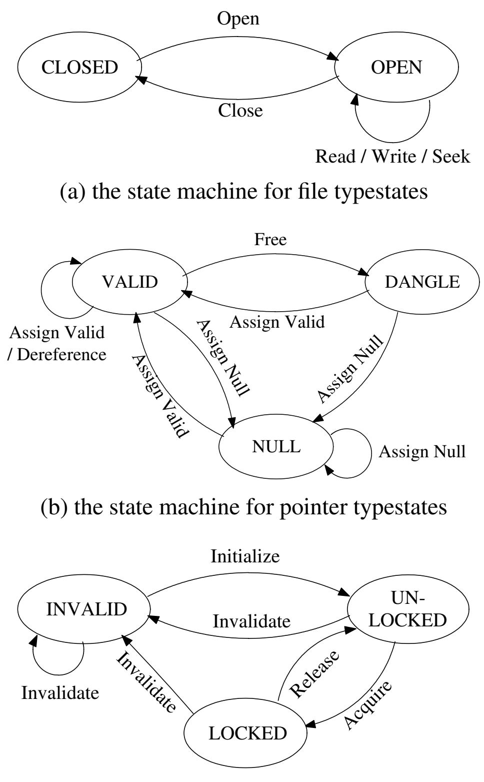

Figure 2. The state machines for files, pointers, and locks

apply operations to the same object that causes a typestate violation. As shown in Figure 1, the order between the operation `pread` by Thread 1 and the operation `close` by Thread 2 is not strictly enforced. Therefore in some rare cases, the file state could first change to `CLOSED` before the operation `pread` is applied, resulting in a typestate violation.

To illustrate how 2ndStrike works, we use three state machines as shown in Figure 2(a), (b), and (c) for three types of concurrency typestate bugs, including invalid file operation, invalid pointer dereference, and invalid lock operation, respectively. We chose them because they are simple state machines suitable for illustrating the idea.

2.2 Concurrency Typestate Bugs versus Data Race/Atomicity Violation Bugs

Concurrency typestate bugs are relevant to, yet quite different from data race/atomicity violation bugs. They are relevant because the synchronization errors causing typestate violations can be data races and/or atomicity violations in some cases. However, the key difference between them is that data race and atomicity violation bugs focus on low-level memory accesses, while concurrency typestate bugs focus on high-level object typestates, which are related to program semantics.

The philosophy behind data race and atomicity violation bugs is that programmers should protect the accesses to shared variables via locks and achieve atomicity in order to guarantee the correctness. Indeed, many concurrency bugs occur because mutual exclusion property and/or atomicity property are not enforced by the code. Ultimately, mutual exclusion and atomicity are properties to distinguish between correct and incorrect execution orders in multithreaded programs. For example, the atomicity property for two

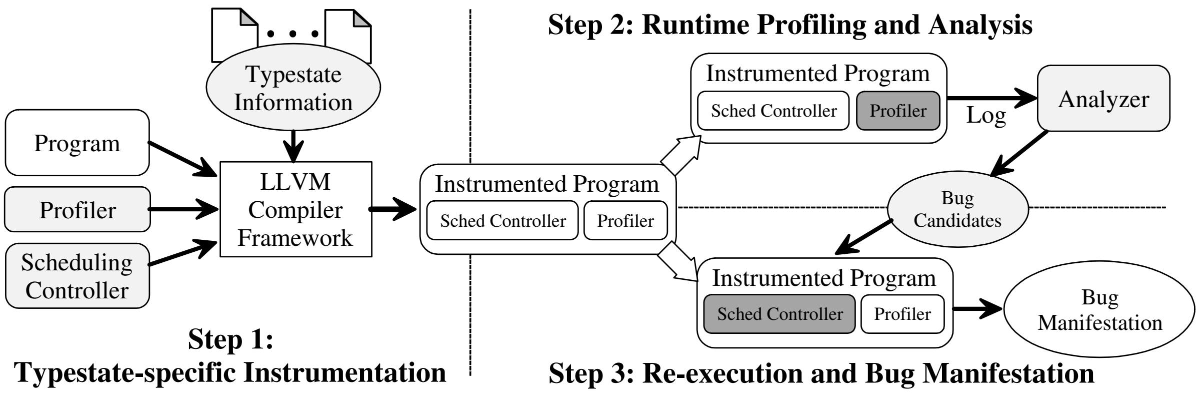

Figure 3. 2ndStrike Overview

consecutive reads on an object a from one thread (R_a^{l1}, R_a^{l2}) with a remote write on the same object from another thread (W_a^r) is to distinguish the incorrect order, i.e., R_a^{l1} W_a^r R_a^{l2}, from the correct orders, i.e., W_a^r R_a^{l1} R_a^{l2} and R_a^{l1} R_a^{l2} W_a^r.

Concurrency typestate bugs are directly related to the orders of runtime operations on objects and thus can capture the general order violation bugs [36], which cannot be considered as atomicity violations or simple data races. By associating semantic information to runtime variables, the notion of concurrency typestate bugs is helpful for detecting concurrency bugs that are related to program semantics.

3. 2ndStrike Design and Implementation

3.1 Overview

2ndStrike performs three steps to manifest hidden concurrency typestate bugs in testing scenarios. As shown in Figure 3, it first performs typestate-specific instrumentation to the program, then executes the program with runtime profiling and analysis, and lastly attempts to manifest the potential bugs with controlled scheduling during program re-execution.

The first step is to instrument the program being tested so that the events related to the typestates can be monitored and further actions can be performed. Different concurrency typestate bugs are related to different aspects of program semantics, users (e.g. programmers or testing engineers) can select one or more defined state machines for a typestate to test in this step. Depending on object types and typestates, the instrumentation can range from very light-weight to somewhat heavy-weight. The idea is to allow users to define the semantics and to control what they want in testing. The profiler and scheduling controller are also linked as shared library in this step.

The second step is to execute the instrumented program with test inputs for profiling typestate related events and to analyze these events for identifying *bug candidates*. A bug candidate is a pair of operations from different threads that can cause a concurrency typestate bug if the order of the operation pair is reversed. For better performance, the analyzer works in parallel with the profiler for processing the log stream.

After finishing the steps of profiling and analysis, 2ndStrike re-executes the program with controlled thread scheduling to manifest the potential bugs indicated by the analyzer. If a bug is successfully manifested, a bug report is generated with the candidate information to help programmers reproduce and diagnose the bug.

3.2 Typestate-Specific Instrumentation

In order to monitor the typestates of runtime objects and to control scheduling, 2ndStrike first instruments the multi-threaded program. The instrumentation is typestate-specific since it is for manifesting a given type of concurrency typestate bugs. For example, for invalid file operation bugs, 2ndStrike instruments the file related library calls such as `open`, `read`, and `close`. Similarly, for invalid pointer dereference bugs, 2ndStrike instruments: a) allocation and de-allocation of runtime objects, such as `malloc` and `free`, to determine the validity of a memory address, b) assignment that involves pointers, and c) dereference of pointers. For invalid lock operation bugs, 2ndStrike instruments lock initialization and invalidation, as well as lock acquire and release operations.

The runtime overhead incurred by the instrumentation largely depends on the targeted typestate. For example, monitoring pointer typestate involves instrumenting a large number of pointer dereference events, and such heavy instrumentation generally results in larger runtime overhead. On the other hand, instrumentation for file typestate is much lighter weight since file operations in a program usually are much less than pointer operations. Users can balance the manifestation benefit and runtime overhead in their own testing environments. Note that since the instrumentation is targeted, even for instrumenting the heavy-weight typestate such as pointer typestate, the runtime overhead is still less than instrumenting each memory access, which is required for data race and atomicity violation detectors (See our evaluation results in Section 6 for details).

Additionally, 2ndStrike keeps track of general synchronization events to prune false bug candidates in the subsequent analysis step (Section 3.3.2). For this purpose, it instruments: a) thread fork/join, b) lock activities, and c) semaphore or condition variable activities.

In our prototype, we use LLVM compiler framework [32] to perform static instrumentation in a code transformation pass. More specifically, the program source code is first fed into the compiler front-end to generate LLVM intermediate representation (IR). Then the 2ndStrike pass is applied to transform the original IR into the instrumented IR. After that, the code generation part of LLVM is used to generate assembly code of the instrumented program. The whole process can be fully automated by modifying the build scripts of the software. Since 2ndStrike simply performs instrumentation, the running time for code transformation is small, ranging from a few seconds for small programs to a few minutes for large programs. We choose static instrumentation over dynamic instrumentation because IR has more semantic information and can make event tracking easier and more accurate.

Our goal for instrumentation is to enable both profiling and scheduling control at program points of interest. To do so, we introduce the concept of *instrumentation point (InstPt)* to perform these two tasks. An InstPt is a specific source code location where 2ndStrike either emits a log entry via profiler or potentially inserts delay via scheduling controller. At each InstPt, 2ndStrike inserts a call to an external library, which performs the actual profiling and scheduling control. In this way, 2ndStrike can easily switch its function between profiling and scheduling without re-instrumenting the code.

3.3 Runtime Profiling and Analysis

3.3.1 Design of Runtime Profiling and On-the-fly Analysis

During the profiling execution, 2ndStrike logs every typestate-related operation and synchronization event. Each log entry includes the following information: a) event type, such as “file close” and “pointer dereference”, b) the thread ID on which the event happens, c) additional event information, e.g. file descriptor that is closed or pointer that is dereferenced, d) the InstPt number corresponding to the event, and e) the callsite that indicates the runtime context of the event.

2ndStrike analyzes the log to generate bug candidates. Each candidate is a pair of operations, from different threads, whose order needs to be flipped for manifesting a potential typestate violation. In each bug candidate, the detailed information on the operations is included for 2ndStrike scheduling controller to accurately identify the runtime operation pair. More specifically, each operation in the bug candidate includes its InstPt and the callsite information. Additionally it can be configured to include the thread ID and the object ID for matching. The algorithm of identifying bug candidates is described in Section 3.3.2.

In 2ndStrike workflow, the analyzer processes log entries in parallel with the profiler generating them. The main reason for the on-the-fly analysis is to reduce storage and runtime overhead. Saving the log to disk may incur non-trivial storage overhead, and also slows down the testing. This is especially true for profiling heavy-weight typestates, e.g., pointer typestate. Therefore, 2ndStrike analyzer runs in a separate process which connects to the program being tested via socket, so that it can consume the generated logs without saving them to disk.

3.3.2 Analysis Algorithm to Identify Bug Candidates

Criteria for Potential Concurrency Typestate Bugs. To manifest concurrency typestate bugs, 2ndStrike needs to identify the potential problems in the synchronization between a state-transition operation (referred to as *OP-transit*) and an operation that is only permitted on a subset of typestates (referred to as *OP-limit*). A bug candidate is a pair of OP-transit and OP-limit in two threads, respectively, that may cause typestate violation if the order of these two operations is reversed.

Figure 4 shows two scenarios where 2ndStrike identifies bug candidates. As shown in Figure 4(a), the first scenario goes as follows. Thread T_1 first performs an operation OP-limit on an object of the state S_p, where OP-limit is not permitted in state S_q. Then another thread T_2 performs an operation OP-transit, which causes the state of the object to transit from S_p to S_q. This is a correct order between these two events in T_1 and T_2, respectively. If the program does not enforce such correct order between the two events, 2ndStrike identifies them as a potential bug. This is because if the order of the two events is reversed, i.e., OP-transit happens before OP-limit, there would be a typestate violation when the OP-limit is performed in the object of the state S_q. Similarly, a bug candidate can also be an OP-transit operation followed by an OP-limit operation that is not permitted in the previous state, as shown in Figure 4(b).

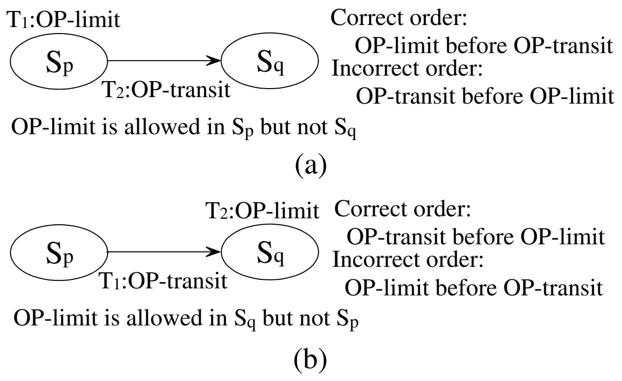
T1:OP-limit
Sp
Sq
T2:OP-transit
Correct order:
OP-limit before OP-transit
Incorrect order:
OP-transit before OP-limit
OP-limit is allowed in Sp but not Sq
(a)
T2:OP-limit
Sp
Sq
T1:OP-transit
Correct order:
OP-transit before OP-limit
Incorrect order:
OP-limit before OP-transit
OP-limit is allowed in Sq but not Sp
(b)

Figure 4. Two scenarios of bug candidates

Algorithm 1 Analysis algorithm

```
1: // For each incoming event $evt$, do the following
2: $candidates \leftarrow \emptyset$
3: if $evt$ is OP-transit then
4:     $set \leftarrow$ QueryOPlimit($evt \rightarrow objId$, $evt \rightarrow tid$,
                    $evt \rightarrow oldState$)
5:     $matchSet \leftarrow \emptyset$
6:     for each OP-limit event $evtPrev$ in $set$ do
7:         if NotPermitted($evtPrev$, $evt \rightarrow newState$) then
8:             $matchSet \leftarrow matchSet \cup \{evtPrev\}$
9:         end if
10:    end for
11:    $cands \leftarrow$ FormCandidates($evt$, $matchSet$)
12: end if
13: $candidates \leftarrow candidates \cup cands$
14: if $evt$ is OP-limit then
15:    $set \leftarrow$ QueryOPtransit($evt \rightarrow objId$, $evt \rightarrow tid$,
                    $evt \rightarrow state$)
16:    $matchSet \leftarrow \emptyset$
17:    for each OP-transit event $evtPrev$ in $set$ do
18:        if NotPermitted($evt$, $evtPrev \rightarrow oldState$) then
19:            $matchSet \leftarrow matchSet \cup \{evtPrev\}$
20:        end if
21:    end for
22:    $cands \leftarrow$ FormCandidates($evt$, $matchSet$)
23: end if
24: $candidates \leftarrow candidates \cup cands$
25: if $evt$ is OP-transit then
26:    InsertOPtransit($evt$)
27: end if
28: if $evt$ is OP-limit then
29:    InsertOPlimit($evt$)
30: end if
```

Identifying Bug Candidates. In order to support the on-the-fly analysis, 2ndStrike analyzer uses a one-pass algorithm that treats the incoming log as a stream. The analyzer keeps track of the typestates of each object and temporarily stores the runtime events in hash tables according to the event type and indexes them by object IDs. For a specific typestate bug, it maintains two hash tables, one for OP-transit and the other for OP-limit.

Algorithm 1 shows how 2ndStrike identifies bug candidates. For each incoming OP-transit event, the analyzer checks the OP-limit table, looking for previous OP-limit events from a different thread on the same object in its old state (line 4). Then, among the resulting OP-limit events, it looks for the ones that are not permitted in the new state to form bug candidates with the OP-transit event (line 5-11). This is for identifying bug candidates in the first scenario as shown in Figure 4(a). Similarly, for each incoming OP-limit event, the analyzer looks up the OP-transit table and identifies the OP-transit events whose previous states do not permit OP-limit to form bug candidates (line 15-22). This is for identifying bug candidates in the second scenario as shown in Figure 4(b). After forming the bug candidates, the event is stored into the corresponding tables (line 25-30). Note that one event can be both OP-transit and OP-limit, e.g., the event `Close` is OP-transit because it causes state transition from the state `OPEN` to the state `CLOSED`, and it is also OP-limit because it is only permitted in the state `OPEN`. In this case, such event will be handled twice–one as OP-transit event and the other as OP-limit event.

Some optimizations can be applied to reduce the number of events in the tables. For example, if the operations permitted in the previous state of an OP-transit include the operations permitted in the new state, the OP-transit is not kept in the table (line 26) because it will never be matched (line 18). In addition, the old events that are outside of a certain window is periodically cleaned up to reduce the memory usage and maintain table lookup efficiency. Therefore, the worst case time complexity for the algorithm is $O(N^2)$, where N is the total number of operations bounded by the window size.

Pruning False Bug Candidates. With identified bug candidates, there could be cases where the two operations cannot be re-ordered. One main reason is that there could be strict order between the operations enforced by thread synchronization events. A technique from previous work [47] is adapted to filter out such strictly-ordered operation pairs. Specifically, 2ndStrike profiles general synchronization events, such as thread fork/join and condition variable wait/signal. Based on this information, it computes a vector clock associated with each operation and forms happens-before relation between operations. In some cases, threads may synchronize each other in customized ways, e.g., using arbitrary shared variables as flags (See Section 4 for details), which results in false bug candidates generated by 2ndStrike. Fortunately, these false candidates will not cause false positives in our final bug reports because if a candidate cannot be manifested by 2ndStrike, it will not be reported to users. Furthermore, a recently-proposed technique [65] can help reduce false candidates by identifying such ad-hoc synchronization in a program.

2ndStrike can prune more false bug candidates by exploiting history information. The observation is that a test suite usually have many inputs and these test inputs tend to cause a similar set of false candidates in the common part of a program such as initialization. To address this issue, 2ndStrike maintains a list of bug candidates that it have attempted but failed to manifest in previous executions. These candidates are likely to be the false ones whose event orders are protected by customized synchronization. For future program execution with different test inputs, 2ndStrike prunes the bug candidates that are in the maintained history list. This design helps increasing the testing efficiency in the long run.

After pruning, 2ndStrike ranks the remaining bug candidates. Similar to previous work [47], our ranking algorithm selects the low-probability candidates since they are more likely to be hidden mistakes. More specifically, our algorithm uses a combination of two metrics, the distance between the two events and the number of occurrence of the event pair at runtime. The candidates with longer distance between two events and smaller number of occurrence are ranked higher.

3.4 Re-execution and Bug Manifestation

After bug candidates being generated, 2ndStrike performs one round of program re-execution for each bug candidate to try to manifest the bug. As each bug candidate consists of two events, we call them *preceding event* and *subsequent event*, reflecting the correct order in normal execution. In the re-execution, 2ndStrike attempts to reorder them by inserting delay before the preceding event.

Specifically, a bug manifesting run uses the same input for the instrumented program with the scheduling controller being activated. At each InstPt, 2ndStrike checks whether the current program location matches the program location of the preceding event in the bug candidate by comparing their InstPt ids and callsites. If they match, 2ndStrike blocks the thread on a semaphore before the operation, allows other threads to proceed as normal, and looks for the matching subsequent event at each InstPt. In addition to matching InstPt ids and callsites with the subsequent event in the bug candidate, the object id in the operation has to match the one in the delayed operation. If any thread has an event matching the subsequent event in the bug candidate, it means the previous thread would commit a typestate violation if it resumes the delayed operation. In this case, 2ndStrike unblocks the previously-blocked thread immediately after the subsequent event occurs in another thread. Additionally, 2ndStrike blocks all other threads so that the previously-blocked thread can get scheduled and commit the typestate violation. At this point, 2ndStrike successfully manifests the bug candidate. Note that 2ndStrike records the runtime information at both preceding event and subsequent event in the bug report.

It is also possible that the matching subsequent event never happens after one thread is blocked before the operation in the preceding event. This is mainly because other synchronization mechanisms such as flag variables are used in the program to prevent such reordering. In this case, 2ndStrike will unblock the previously-blocked thread and keep looking for the preceding event for this bug candidate after a certain time threshold. After a certain number of such timeouts, 2ndStrike will abort the reordering attempt and stop re-execution for this bug candidate.

4. Issues and Discussion

Un-monitored synchronization. Similar to previous active testing tools [47, 57], 2ndStrike does not monitor all possible synchronizations occurring in the program execution, especially the ones based on shared flag variables. This will cause 2ndStrike analyzer to generate some operation pairs that are impossible to be reordered as bug candidates. Although these candidates will be pruned later since 2ndStrike cannot manifest them, they do reduce the test efficiency. To address this issue, we can either ask developers for hints on their customized synchronization mechanisms or derive such information using techniques proposed in [65]. How to more accurately capture the synchronization behavior for exposing concurrency typestate bugs are left for our future work.

Non-determinism. Multi-threaded programs are naturally non-deterministic in their execution, which means the profiling information based on previous executions may not be valid for subsequent executions. Similar to previous active testing tools [47, 57], 2ndStrike assumes that the program execution paths at different runs are mostly the same for a given test input. In practice, unfortunately, there are certain degree of non-determinism involved in program executions especially for large complex software. Therefore, for some program executions that are very different from the profiling execution, 2ndStrike may fail to manifest the concurrency typestate bugs because the object may not be operated in the same way. To address this issue, one way is to integrate 2ndStrike with existing deterministic replay tools [15, 22] for replaying the program execution as much as possible until the scheduling change has to be made.

App.	Bug id	Bug description
MySQL	MySQL-file	A file descriptor was accessed after being closed, causing assertion failure
MySQL	MySQL-ptr1	An invalid pointer dereference causing crash when handling transactions
MySQL	MySQL-ptr2	An invalid pointer dereference causing crash on accessing recovery objects
MySQL	MySQL-ptr3*	An invalid pointer dereference causing crash on accessing a freed table index
Mozilla	Mozilla-ptr	An invalid pointer dereference causing crash when destroying contexts
pbzip2	pbzip2-lck	An invalid lock usage causing crash when decompressing a file

Table 1. Evaluated applications and concurrency typestate bugs. ***MySQL-ptr3 was previously unknown and manifested by 2ndStrike**.

Input coverage. 2ndStrike's capability for manifesting bugs depends on test inputs. Although the ability to integrate with existing test harnesses and use existing test suites can alleviate this issue, 2ndStrike does not provide systematic coverage as model checking tools (e.g., CHESS [40]) do. To address this issue, we hope to integrate 2ndStrike with symbolic execution techniques [10, 26, 58] for generating test inputs to cover interesting program paths in the future.

False negatives. Although 2ndStrike guarantees there is no false positives in final results, it may have false negatives. Two main sources for false negatives are above-mentioned inadequate input coverage and non-determinism. In addition, history-based bug candidate pruning could also cause false negatives since one failed attempt to manifest a bug cannot rule out the possibility of the bug. Furthermore, users may choose to only try a limited number of bug candidates as allowed by resources, in which case users need to balance the risk of potential false negatives and the testing overhead.

5. Evaluation Methodology

We have implemented a prototype of 2ndStrike on Linux and conducted the experiments on a Intel Xeon machine with four 2GHz processing cores (eight logical processors with hyper-threading enabled). The L2 cache is 12MB and the DRAM size is 16GB. The Linux kernel version is 2.6.16.

We have evaluated 2ndStrike with three different types of multithreaded applications, including MySQL [1], a database server, Mozilla [2], a web browser, and pbzip2 [3], a utility program for file compression and decompression, as shown in Table 1. These applications contain six real-world concurrency typestate bugs of three different types, including invalid file operation, invalid pointer dereference, and invalid lock operation. For MySQL, we test its Falcon storage engine, a new storage engine for highly concurrent workloads. For Mozilla, we test its JavaScript engine with concurrent workloads.

To evaluate the effectiveness, we run 2ndStrike with the test inputs that can potentially trigger (but do not trigger) the bug cases. Whenever possible, we use the default testing frameworks for the applications such as `mysql-test`, and the default testing inputs such as JavaScript files provided in the Mozilla JavaScript engine release package. This allows us to evaluate how 2ndStrike integrates with existing test harnesses, which we consider as an important issue for usability. For pbzip2, since it does not have test inputs included in the package, we use a Linux kernel tarball as input (40MB bz2 file decompressed into 224MB tar file).

For comparison, we have performed a stress testing by running the tested programs with the same inputs repetitively. Furthermore, in order to evaluate whether the active testing tools proposed in RaceFuzzer [57] and CTrigger [47] are effective in detecting the bugs in tested programs, we have conducted data race detection and atomicity violation detection with the same inputs. We implement the hybrid race detection technique [45] (used by RaceFuzzer) to detect races and implement the unserializable access detection idea [33] (used by CTrigger) to detect atomicity violation.

Bug id	2ndStrike	Stress	Data race directed	Atomicity violation directed
MySQL-file	20.4	No	No	No
MySQL-ptr1	137.0	45,535	No	No
MySQL-ptr2	103.7	No	No	No
MySQL-ptr3	160.8	No	No	No
Mozilla-ptr	85.0	No	No	No
pbzip2-lck	9.8	No	Yes	Yes

Table 2. Overall effectiveness of 2ndStrike: The '2ndStrike' and 'Stress' columns show the time in seconds needed to manifest the bug for 2ndStrike and stress testing, respectively. 'No' means a bug cannot be manifested within 24 hours. The last two columns show whether the data race and atomicity violation directed tools can effectively manifest the bugs. 'Yes' and 'No' indicate whether they can or cannot manifest the bugs, based on experimental results from data race detection and atomicity violation detection.

To evaluate the efficiency of 2ndStrike, we have conducted experiments for measuring both profiling/analyzing overhead and bug manifesting overhead (i.e., for controlled thread interleavings). For profiling/analyzing overhead, we measure the program execution time under the following four configurations:

- Baseline, executing the programs natively without 2ndStrike.
- 2ndStrike's profiling only, executing the programs with 2ndStrike's profiling enabled, but logs not recorded (fed into `/dev/null`).
- 2ndStrike's profiling and analyzing, executing the programs with 2ndStrike profiling enabled and running the analyzer in parallel for generating bug candidates.
- Memory access profiling, executing the programs with profiling every memory access and logs not recorded. This is for comparing 2ndStrike's profiling overhead with existing active testing tools based on memory accesses.

For bug manifesting overhead, we measure the execution time for a manifesting run with the bug successfully manifested, and the execution time for bug manifesting run without bug being manifested, e.g., the tested bug candidate is not corresponding to a real bug.

Moreover, we have evaluated the reproducibility of 2ndStrike for each bug. With the bug candidate identified by 2ndStrike after initial successful manifestation, we execute the 2ndStrike in manifesting mode for 30 times to measure how well the bug can be reproduced. High reproducibility can greatly improve developers' efficiency of understanding and fixing a bug.

6. Experimental Results

6.1 Effectiveness

Table 2 describes our evaluation results on effectiveness of 2ndStrike. The second column shows the total time in seconds for profiling the program, analyzing the log, and manifesting the bug for each bug case. The third column shows the time in seconds for manifesting the bug via stress testing. 'No' in this column means

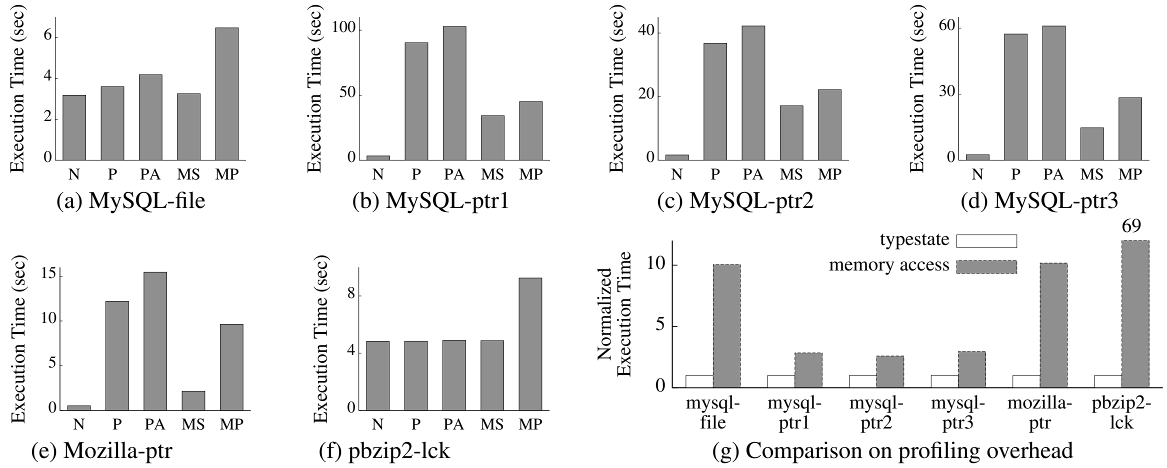

Figure 5. Efficiency of 2ndStrike: (a) to (f) show the performance results for 2ndStrike. (g) compares the profiling overhead for 2ndStrike with profiling every memory access. In (a) to (f), 'N' means normal execution without 2ndStrike being applied; 'P' means execution with 2ndStrike profiling only and logs are fed into `/dev/null`; 'PA' means execution with both 2ndStrike profiling and on-the-fly analysis enabled; 'MS' means bug manifesting run that successfully manifests the bug; 'MP' means bug manifesting run that passes without manifesting any bug.

the bug cannot be manifested by stress testing within 24 hours. The last two columns show whether the data race and atomicity violation directed tools can effectively manifest the bug.

Our results show that 2ndStrike is effective in manifesting the tested concurrency typestate bugs. With 2ndStrike, all six tested bugs are manifested within 160.8 seconds. The main reason of 2ndStrike's effectiveness is that it focuses on the semantic errors instead of only on memory accesses. For example, instead of only observing accesses to file descriptors and pointers, 2ndStrike considers the operations close and deallocation as OP-transit, and the operations read and dereference as OP-limit. This allows 2ndStrike to identify the potential typestate violations and manifest them during re-execution. We expect 2ndStrike enabling programmers to manifest more semantic-related bugs by manually defining or automatically inferring more application-specific typestate models.

In our experiments, 2ndStrike discovers one unknown bug referred to as MySQL-ptr3. The bug occurs when a database front-end thread is altering a table and freeing an old index, immediately followed by an operation execution and the index access from a back-end thread. This concurrency behavior does not involve traditional data races since both critical sections are protected by locks. However, it does violate the typestate property: accessing a freed index. In fact, this type of bugs can be caught more easily if 2ndStrike is based on more application-specific typestate instead of generic ones on pointers.

On the contrary, the stress testing cannot manifest five out of the six bugs, at least within 24 hours. For MySQL-ptr1, stress testing is 331 times slower than 2ndStrike for manifesting the bug. This is mainly because stress testing tend to focus on only a fraction of possible thread interleavings.

Our experimental results from data race detection and atomicity violation detection also show that the active testing tools based on them cannot manifest five of the six bugs. This is because the bug-triggering object is not involved in data races or unserializable accesses from multiple threads for these bugs. For example, in MySQL-file case, both `pread` and `close` read the file descriptor but do not write. Similarly, for the dangling pointer bugs, both dereference and deallocation only read the pointer. In fact, Mozilla-ptr was categorized as 'order violation' in a previous study [36] because it does not involve unserializable accesses to shared variables. Due to this reason, these tools are unable to manifest these five bugs. For pbzip2-lck, the data race detector can catch the error because the lock variable is involved in data race. Similarly, the unserializable access detector can also detect the error in the pbzip2-lck case. However, the profiling on every memory access incurs much larger overhead than 2ndStrike does and there are many benign data races reported by the race detector.

6.2 Efficiency

Figure 5 shows the detailed performance results when using 2ndStrike to manifest the six concurrency typestate bugs.

We can observe that by using typestate specific instrumentation, 2ndStrike incurs tolerable runtime overhead incurred by profiling and analysis for software testing purposes. The profiling overhead highly depends on the typestate being profiled. For example, profiling file typestate for MySQL-file and lock typestate for pbzip2-lck only incur 13% and 1% overhead, respectively. However, profiling pointer typestate incurs 22x to 27x overhead, which is largely due to frequent pointer dereference events. Figure 5 (g) compares 2ndStrike's profiling overhead with the overhead incurred by profiling every memory access. We can see that 2ndStrike profiling incurs 61% to 98% less overhead.

Moreover, the results show that 2ndStrike's on-the-fly analysis design helps reduce the profiling and analysis overhead. Performing analysis along with profiling only makes the execution time a little longer, by 1.4% to 16.1% with an average 13.2% (see column 'PA' versus column 'P'). In comparison, performing analysis alone with the log from disk takes about the same time as profiling and performing on-the-fly analysis. In addition, on-the-fly analysis totally avoids saving the log to disk, which otherwise would cause substantial space overhead for the heavy-weight typestate profiling.

Table 3 shows the detailed results on typestate-specific profiling versus memory access profiling. The second column and the third column show the number of static instrumentation points for 2ndStrike and memory access profiling, respectively. The last two columns show the number of dynamically profiled events for 2ndStrike and memory access profiling, respectively. This table clearly indicates the reason why 2ndStrike incurs small overhead for lightweight typestates such as file and lock. Taking MySQL-file as an example, 2ndStrike only instruments at 654 program locations, which combines file typestate profiling with general synchronization. On the contrary, 58,033 program locations must be instru-

Bug id	2ndStrike InstPt	Memory access InstPt	2ndStrike events	Memory access events
MySQL-file	654	58,033	164,702	12,147,265
MySQL-ptr1	27,525	59,541	26,541,228	78,651,987
MySQL-ptr2	27,379	59,301	9,522,145	28,381,980
MySQL-ptr3	30,378	72,608	16,699,980	51,135,709
Mozilla-ptr	18,194	172,409	3,360,687	29,054,798
pbzip2-lck	163	1,479	11,016	165,005,902

Table 3. Statistics of instrumentation and runtime events by 2ndStrike in comparison with memory access instrumentation and profiling

Bug id	Generated	After pruning	Rank
MySQL-file	9	5	3
MySQL-ptr1	568	36	1
MySQL-ptr2	751	204	3
MySQL-ptr3	2013	16	4
Mozilla-ptr	176	60	8
pbzip2-lck	11	11	1

Table 4. Effectiveness of pruning and ranking bug candidates in 2ndStrike

mented to profile memory access, which also generates orders-of-magnitudes more runtime log entries.

Pruning and ranking bug candidates is also critical to 2nd-Strike's efficiency. Table 4 shows the results of pruning and ranking. The second column shows the total number of bug candidates generated. The third column shows the number of bug candidates after pruning based on the strong synchronization order and history information. The last column shows the rank of the bug-triggering candidate, or the highest rank in the case where multiple candidates can trigger the bug. We can observe that the pruning and ranking algorithm can effectively reduce the number of false bug candidates tried in re-execution, and therefore improve testing efficiency.

6.3 Usability

Integration with testing framework. 2ndStrike can be easily integrated with existing testing frameworks and inputs. It does not require source code change or special dynamic instrumentation environments. All it requires is changing the build scripts for the software component. In addition, it is flexible to be applied only to a subset of components in a large software package. In our experiments with MySQL and Mozilla, we only apply 2ndStrike to MySQL's Falcon storage engine and Mozilla's JavaScript engine. We believe that this flexibility can help developers perform more targeted testing and is very important for large software with many components.

Bug reporting. After manifesting a bug, 2ndStrike generates an accurate and detailed bug report, which includes following pieces of information: a) the test input that can potentially trigger the bug, b) the typestate violation manifested, c) the bug candidate, and d) the runtime 'preceding event' and 'subsequent event' that happened during bug manifesting execution. Note that since the bug is manifested when the order between the 'preceding event' and the 'subsequent event' is flipped, the 'subsequent event' actually happens first in the buggy run while the 'preceding event' happens after. For both events, the bug report includes: a) the InstPt information, which corresponds to a unique source code location, b) the runtime callsite, including multiple levels of function names, c) the thread id, and d) the object id. This information can be very helpful for developers to diagnose the bug and pinpoint the root cause.

Bug id	MySQL-file	MySQL-ptr1	MySQL-ptr2
Reproduce rate	100%	86.67%	93.33%
Bug id	MySQL-ptr3	Mozilla-ptr	pbzip2-lck
Reproduce rate	100%	100%	100%

Table 5. Reproducibility of the manifested concurrency typestate bugs in 2ndStrike

No false positives. In addition, 2ndStrike reports no false positives in its final results. This is because all bug candidates that are not corresponding to real bugs are pruned automatically in the bug manifesting executions. Only the bug candidates that can trigger real concurrency typestate bugs are reported in the final results. This feature is also critical to developers, who are generally very reluctant to spend efforts in vain on spurious errors.

Bug reproducing. Moreover, 2ndStrike helps developers by reproducing a manifested bug easier. As shown in Table 5, a candidate of concurrency typestate bug can be reproduced by 2ndStrike with a high probability after it is being manifested once. The reason why some are not 100% reproducible is the non-deterministic nature of multi-threaded program executions (see Section 4 for details). In some of these multi-threaded program executions, the exercised code path is different from that in the profiling run.

7. Related Work

Our work is related to concurrent program testing, typestate bug detection, concurrency bug detection, concurrency bug diagnosis, and concurrency bug prevention or avoidance.

Concurrent program testing. Researchers have conducted many studies on testing concurrent programs, including generating input test cases to execute certain regions of concurrent programs [58], devising interleaving coverage criteria to measure the coverage of concurrency testing [27, 30, 34, 63, 69], and exposing potential memory level errors including data races or atomicity violation bugs [8, 39, 40, 47, 57]. Recently-proposed ConMem [75] detects concurrency bugs that can cause severe memory errors. Techniques for generating input test cases is complementary to our work for testing concurrent programs because concurrency bug manifestation requires both certain inputs and certain thread/process interleaving. Furthermore, interleaving coverage criteria can be used to guide our work for improving the coverage. As discussed in Section 1 and Section 2.2, existing active testing techniques focus on memory accesses and mainly to expose data races or atomicity violation bugs. Unlike these approaches, our work can manifest concurrency typestate bugs, ranging from the memory level (e.g., NULL pointer dereferences) to high-level program semantics (e.g., file operation errors). PCT [9] is a randomized scheduler for manifesting concurrency bugs. 2ndStrike can improve PCT by guiding the assignment of thread priorities and identifying priority change points.

Typestate bug detection. There are plenty of studies on detecting typestate bugs in programs. Some approaches detect typestate bugs via program analysis [18, 25, 62] or symbolic execution [17], while others checks typestate properties at runtime [6, 7, 11, 12]. These approaches are proposed for detecting typestate bugs in sequential programs.

PRETEX [28] is close to our work. It detects potential race-related typestate bugs in multi-threaded Java programs by analyzing correct program executions combined with other possible execution paths indicated by race conditions. However, it does not attempt to construct an execution to manifest the detected potential typestate bugs. Along with the fact that PRETEX is based on inferred typestate properties, it may cause a large number of false positives reported to programmers. Unlike this tool, our work attempts to manifest the non-exposed concurrency typestate bug candidates via system methods, therefore all false positives are automatically pruned. Furthermore, our work can handle the concurrency typestate bugs that are not related with race conditions. Yang et al. have proposed typestate-guide static analysis for detecting data races and atomicity violations [70]. This method may report false positives since it utilizes static methods to model threading interleavings.

Concurrency bug detection. Many studies have been conducted on detecting concurrency bugs, including static methods [19, 23, 54, 61] and dynamic approaches [14, 20, 43, 44, 50, 51, 56, 72] for race detection, and techniques [33, 35] for atomicity violation detection. Complementary to these detection methods, which depend on manifestation of concurrency bugs, our work helps manifest hidden concurrency bugs, belonging to the same category as previous active testing work [8, 39, 40, 47, 57].

Concurrency bug diagnosis. With the assistance of deterministic replay [21, 22, 41, 42, 55, 60, 66, 68] or execution sketching [48], bug diagnosis techniques such as delta debugging [13, 16, 74] and program slicing [4, 64, 76] can be used for understanding concurrency bugs. Recent work on execution synthesis [73] helps inferring a buggy input and the related thread interleaving by extracting information from coredump/callstack and the source code. Similar to concurrency bug detection tools, these diagnosis methods also depend on bug manifestation, which our work can help.

Concurrency bug avoidance. Recently, several studies have been proposed for avoiding concurrency bugs. Programming on transactional memory [31] prevents data races. Isolator [52] and ToleRace [53] exploit data replication to tolerate data races. Atomaid [38] leverages transactional memory for surviving atomicity violation bugs during production runs. Yu et al. have proposed a shared-memory multi-processor design to avoid untested thread interleavings that may cause concurrency bugs [71]. Dimmunix [29] prevents programs from re-encountering previously-seen deadlocks. Unlike these production-phase tools, our work is applied in testing phase.

8. Conclusions

In summary, this paper presents a general testing framework called 2ndStrike for manifesting concurrency typestate bugs in multi-threaded programs in the testing phase. To do so, 2ndStrike profiles runtime events that are related to object typestates described by a given state machine along with synchronization events. Then 2ndStrike identifies bug candidates by checking the profiled events against the state machine with different orders of possible thread interleavings. Finally 2ndStrike re-executes the program with controlled thread interleaving for manifesting the bug candidates.

Our evaluation of 2ndStrike prototype with six real world bugs from three open-source server and desktop programs (i.e., MySQL, Mozilla, pbzip2) has shown that 2ndStrike can effectively and efficiently manifest concurrency typestate bugs, which are otherwise difficult to be manifested using stress testing or data race/atomicity violation based active testing techniques. Additionally, 2ndStrike provides detailed bug reports and can consistently reproduce the bugs after manifesting the bugs once.

Acknowledgments

We would like to thank David August, our shepherd, and anonymous reviewers for their insightful feedback. We also thank Min Xu for early discussions. This research is supported in part by NSF grant#CCF-0953759 (CAREER Award).

References

[1] Mysql. `http://www.mysql.com/`.

[2] Mozilla. `http://www.mozilla.org/`.

[3] Parallel bzip2. `http://compression.ca/pbzip2/`.

[4] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2002.

[6] M. Arnold, M. Vechev, and E. Yahav. Qvm: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, 2008.

[7] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications*, 2007.

[8] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2005.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS '10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI '08: Proceedings of Operating System Design and Implementation*, 2008.

[11] P. Centonze, G. Naumovich, S. J. Fink, and M. Pistoia. Role-based access control consistency validation. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, 2006.

[12] F. Chen and G. Roşu. Mop: an efficient and generic runtime verification framework. In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications*, 2007.

[13] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA '02: Proceedigns of the International Symposium on Software Testing and Analysis*, 2002.

[14] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[15] J. Chow, D. Lucchetti, T. Garfinkel, G. Lefebvre, R. Gardner, J. Mason, S. Small, and P. M. Chen. Multi-stage replay with crosscut. In *VEE '10: Proceedings of the 2010 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2010.

[16] H. Cleve and A. Zeller. Finding failure causes through automated testing. In *Proceedings of the Fourth International Workshop on Automated Debugging*, 2000.

[17] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[18] R. DeLine and M. Fahndrich. Typestates for objects. In *ECOOP '04: Proceedings of the 18th European Conference on Object-Oriented Programming*, 2004.

[19] D. L. Detlefs, K. R. M. Leino, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical report, TR SRC-159, COMPAQ SRC, 1998.

[20] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPoPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 1990.

[21] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *OSDI '02: Proceedings of the 5th symposium on Operating Systems Design and Implementation*, 2002.

[22] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008.

[23] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[24] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, 2004.

[25] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, 2002.

[26] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS '08: Proceedings of the 16th Annual Network and Distributed System Security Symposium*, 2008.

[27] M. Harrold and B. Malloy. Data flow testing of parallelized code. In *ICSM '92: Proceedings of International Conference on Software Maintenance*, 1992.

[28] P. Joshi and K. Sen. Predictive typestate checking of multithreaded jave programs. In *ASE '08: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[29] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[30] P. V. Koppol and K.-C. Tai. An incremental approach to structural testing of concurrent software. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1996.

[31] J. R. Larus and R. Rajwar. Transactional memory. *Morgan & Claypool*, 1(1):1–226, 2006.

[32] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[33] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[34] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, 2007.

[35] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[36] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS '08: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[37] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *Micro'09: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[38] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, 2008.

[39] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2007.

[40] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler. Finding and reproducing heisenbugs in concurrent programs. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[41] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*, 2005.

[42] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[43] R. H. B. Netzer and B. P. Miller. Improving the accuracy of data race detection. In *PPoPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 1991.

[44] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2003.

[45] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2003.

[46] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE '08: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[47] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[48] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[49] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: Fault localization in concurrent programs. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010.

[50] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI '96: Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996.

[51] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded c++ programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2003.

[52] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. Isolator: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th International Conference on*

Architectural Support for Programming Languages and Operating Systems, 2009.

[53] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, 2009.

[54] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Esc/java users manual. Technical report, Compaq Systems Research Center, 2001.

[55] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proceedings of the sixth International Symposium on Automated Analysis-driven Debugging*, 2005.

[56] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[57] K. Sen. Race directed random testing of concurrent programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2008.

[58] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *FASE '06: Proceedings of Fundamental Approaches to Software Engineering*, 2006.

[59] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, 2007.

[60] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Technical Conference*, 2004.

[61] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the 1993 USENIX Winter Technical Conference*, 1993.

[62] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[63] R. Taylor, D. Levine, and C. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18:206–215, 1992.

[64] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[65] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI '10: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[66] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[67] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, 2005.

[68] M. Xu, M. D. Hill, and R. Bodik. A regulated transitive reduction (rtr) for longer memory race recording. In *ASPLOS '06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[69] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1998.

[70] Y. Yang, A. Gringauze, D. Wu, and H. Rohde. Detecting data race and atomicity violation via typestate-guided static analysis. Technical report, Microsoft Research, 2008.

[71] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA '09: Proceedings of the 36th annual International Symposium on Computer architecture*, 2009.

[72] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, 2005.

[73] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys '10: Proceedings of the 5th European Conference on Computer Systems*, 2010.

[74] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC-FSE '99: The Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, 1999.

[75] W. Zhang, C. Sun, and S. Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010.

[76] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003.