Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing

Shan Lu, Soyeon Park, and Yuanyuan Zhou, *Member*, *IEEE*

Abstract—Multicore hardware is making concurrent programs pervasive. Unfortunately, concurrent programs are prone to bugs. Among different types of concurrency bugs, atomicity violations are common and important. How to test the interleaving space and expose atomicity-violation bugs is an open problem. This paper makes three contributions. First, it designs and evaluates a hierarchy of four interleaving coverage criteria using 105 real-world concurrency bugs. This study finds a coverage criterion (Unserializable Interleaving Coverage) that balances the complexity and the capability of exposing atomicity-violation bugs well. Second, it studies stress testing to understand why this common practice cannot effectively expose atomicity-violation bugs from the perspective of unserializable interleaving coverage. Third, it designs CTrigger following the unserializable interleaving coverage criterion. CTrigger uses trace analysis to identify feasible unserializable interleavings, and then exercises low-probability interleavings to expose atomicity-violation bugs. We evaluate CTrigger with real-world atomicity-violation bugs from seven applications. CTrigger efficiently exposes these bugs within 1-235 seconds, two to four orders of magnitude faster than stress testing. Without CTrigger, some of these bugs do not manifest even after seven days of stress testing. Furthermore, once a bug is exposed, CTrigger can reliably reproduce it, usually within 5 seconds, for diagnosis.

Index Terms—Testing and debugging, debugging aids, diagnostics, testing strategies, test coverage of code, concurrent programming, bug characteristics.

1 Introduction

1.1 Motivations

The reality of multicore hardware is making concurrent programs pervasive. Unfortunately, concurrent programs are prone to bugs due to the inherent complexity of concurrency. These bugs are difficult to detect and diagnose because of their unique nondeterminism. Many concurrency bugs are missed during in-house testing. They escape into production runs and cause catastrophic disasters in the real world (e.g., the Northeastern Electricity Blackout Incident [34]).

Among different causes of concurrency bugs, an atomicity violation is one of the most common ones [7], [21], [22], [23], [39]. Simply speaking, an atomicity violation occurs when there is unexpected interference between the execution of a block of code by one thread and operations concurrently performed by other threads (a stricter definition will be presented in Section 2). An example of a real-world atomicity-violation bug is illustrated in Fig. 1. As shown in the figure, S1 and S2 both access `buf_index`. They were not protected by any lock or transaction. As a result, S3 from thread 2 could violate the atomicity of code region S1-S2 in thread 1 and cause the program to crash.

- *S. Lu is with the Computer Science Department, University of Wisconsin–Madison, 1210 W. Dayton Street, Madison, WI 53706. E-mail: shanlu@cs.wisc.edu.*
- *S. Park and Y. Zhou are with the Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Dr. M/C 0404, La Jolla, CA 92093. E-mail: {soyeon, yyzhou}@cs.ucsd.edu.*

Manuscript received 6 Feb. 2010; revised 15 July 2010; accepted 4 Mar. 2011; published online 22 Mar. 2011.
Recommended for acceptance by P. Frankl.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-02-0037.
Digital Object Identifier no. 10.1109/TSE.2011.35.

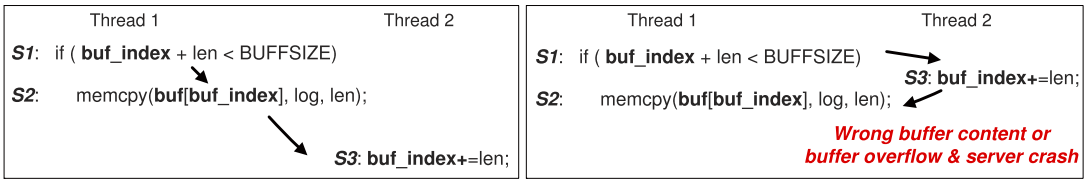
Atomicity-violation bugs widely exist because many programmers are used to sequential thinking and frequently assume code regions to be atomic without proper enforcement. A recent characteristic study [21] on concurrency bugs in open-source applications shows that about 70 percent of nondeadlock concurrency bugs are caused by atomicity violations. Therefore, techniques for eliminating atomicity-violation bugs are highly desired.

A lot of progress has been made recently on detecting atomicity-violation bugs [7], [8], [10], [22], [39], [42], [46]. However, many dynamic bug detection tools [8], [10], [22], [46] cannot accurately report a bug until they observe a buggy execution where the bug manifests. Therefore, we still need effective approaches to exploring multithreaded programs' state space and forcing atomicity-violation bugs to manifest during in-house testing (i.e., **exposing** bugs).

A good bug-exposing approach needs to have three properties:

- *Effectiveness*. Exposing as many hidden bugs as possible.
- *Efficiency*. Exposing hidden bugs as quickly as possible. Although performance during testing is not as critical as it is during production runs, the bug-exposing process cannot take months or years because programmers are under constant pressure to release software.
- *Reproducibility*. Reliably reproducing a bug for diagnosis. If the bug takes another 20 hours to reproduce, it will be painful for programmers to examine the problem.

Compared to sequential bugs, exposing concurrency bugs is much more difficult while meeting the above three requirements. Unlike sequential bugs, concurrency bugs



(a) A non-bug triggering interleaving, which almost always occurs (b) A bug triggering interleaving, which rarely occurs

Fig. 1. An example simplified from an Apache atomicity-violation bug. (It is triggered by S3 executed between S1 and S2.)

usually require at least two conditions to manifest. The first condition, similar to that of sequential bugs, is a *bug-triggering input*. An appropriate input is needed to execute a faulty code segment with a bug-triggering state. Most of the techniques developed for sequential programs are still applicable to concurrent programs, with small extensions [36].

The second condition, unique to concurrency bugs, is a *bug-triggering interleaving*. Without this condition, a bug-triggering input alone cannot expose the hidden concurrency bug. For example, the bug in Fig. 1 is difficult to expose during in-house testing because it manifests *only* when $S3$ is executed between $S1$ and $S2$. The probability for this particular execution order is small. Actually, when we ran Apache with a *bug-triggering input* on an eight-core machine, it took 22 hours for this bug to manifest.

In comparison to the first condition, the second condition is significantly understudied. Therefore, similarly to recent work on concurrent program testing [6], [26], [27], [35], *this paper focuses on the bug-triggering interleaving issue* and relies on prior techniques to cover the first condition.

1.2 State of the Art

A common practice to expose concurrency bugs is to run a program with each input for many times, which is usually referred to as *stress testing* [26]. The rationale behind this practice is that the nondeterministic nature of concurrent programs will help exercise different interleavings in different runs. Unfortunately, the reality has shown that stress testing is neither efficient nor reproducible [26]. The first part of this paper will dig deeper into this phenomenon.

Recently, several inspiring works [4], [6], [26], [27], [35], [37] have improved stress testing. They all try to select a small number of interleavings from the exponential-size interleaving space, and make testing focus on these representative interleavings.

ConTest [4] injects artificial delays at synchronization points (e.g., lock acquisition and release) to intensify the contention for synchronization resources. It can help expose deadlocks, but is not effective in exposing atomicity-violation bugs because the vast majority of atomicity-violation bugs are caused by accesses not separated by any synchronization operations (e.g., the bug in Fig. 1).

CHESS [26], [27] focuses testing on interleavings with small numbers (e.g., 1-4) of context switches. In order to scale to large real-world applications, CHESS limits context switches to occur only at synchronization points. This constraint makes CHESS less effective for atomicity-violation and data-race bugs, just like ConTest. This paper will complement CHESS by systematically picking out interleavings that have low occurrence probabilities and a high likelihood of exposing atomicity-violation bugs.

RaceFuzzer [35] focuses on interleavings that can exercise potential data races reported by race detectors. It manipulates the execution order between race instructions during testing so that false positives can be separated from true bugs. Its bug-exposing capability relies on the underlying race detectors. RaceFuzzer cannot expose a bug unless the underlying detector reports it first. Unfortunately, almost no detector can achieve high coverage for data races in C/C++ programs without introducing a huge number of false positives due to benign race issues [29]. More importantly, many concurrency bugs, including many atomicity-violation bugs, are data-race free [22].

In addition, both CHESS and RaceFuzzer select only one thread to execute at a time, which can significantly slow down each test run and cannot take advantage of multicore machines in testing. While it is possible to conduct multiple test runs on the same machine, the contention for disk and network makes it impractical for I/O-intensive applications, such as server programs. In this paper, we propose an approach to addressing this limitation and allow each test run to use multiple processors, just like that in stress testing.

1.3 Contributions of This Paper

This paper first studies the manifestation conditions of many real-world concurrency bugs and the characteristics of stress testing. Guided by these studies, a framework, called CTrigger, is proposed to efficiently expose atomicity-violation bugs in large programs.

First, we design a hierarchy of four interleaving coverage criteria and evaluate them using 105 real-world concurrency bugs. These criteria are designed based on different fault models. They demonstrate different tradeoffs between complexity and bug-exposing capability.

Second, guided by the above study, we design a linear-sized testing space, called an *unserializable interleaving space*, to expose atomicity-violation bugs. This testing space is designed based on the most common type of atomicity-violation bugs: atomicity violations to two instructions that consecutively access the same memory location from the same thread [22], [46].

Third, using seven representative open-source programs, we examine why stress testing is insufficient in covering the unserializable interleaving space and exposing atomicity-violation bugs. Our evaluation shows that different unserializable interleavings have different probabilities to occur; different runs in stress testing usually cover similar interleavings (i.e., high-probability ones); low-probability ones, which usually hide atomicity-violation bugs, are hard to cover without external control and are also hard to reproduce for diagnosis. The primary factors that affect interleaving probabilities are synchronizations, memory access distances, etc.

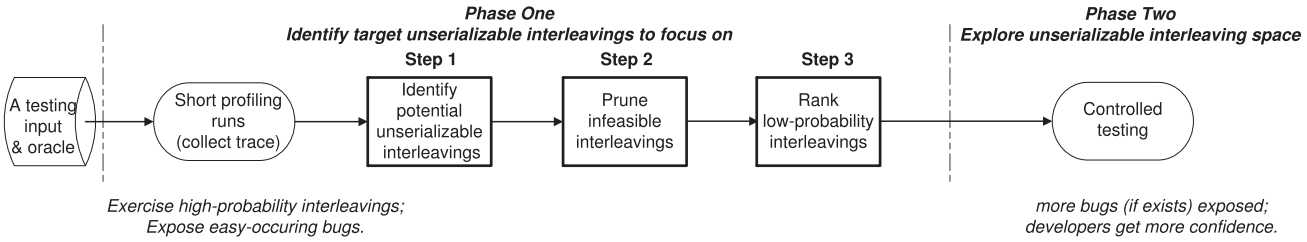


Fig. 2. CTrigger testing framework. (Phases one and two are conducted for each test input.)

Fourth, based on the above observation, we design a testing framework called CTrigger to effectively cover the unserializable interleaving space and expose atomicity-violation bugs with efficiency and reproducibility. CTrigger achieves these goals by incorporating the following new ideas step by step, as shown in Fig. 2.

- Focusing on unserializable interleavings. From a few, mostly one, profiling runs, CTrigger identifies a large set of potential unserializable interleavings.
- Pruning infeasible interleavings. Some potential unserializable interleavings can never happen due to synchronization operations. For example, two accesses protected by a lock cannot be unserializably interleaved by another access protected by the same lock. We have designed an algorithm to prune these infeasible interleavings by considering both order synchronizations and mutual exclusion synchronizations. Our pruning significantly reduces the number of vain attempts to force infeasible interleavings. Our experimental results show that 37-96 percent of potential unserializable interleavings in the seven tested applications are pruned.
- Ranking and identifying low-probability interleavings. As different interleavings have different probabilities to manifest, we propose a simple metric to estimate interleaving probability and rank unserializable interleavings. This ranking mechanism allows us to focus on low-probability interleavings during controlled testing, and leaves high-probability ones to be covered by stress testing. Our experimental results show that our ranking mechanism is effective. It ranks bug-triggering interleavings high, mostly within the top 10 percent, and speeds up the bug exposing by up to 457 times. Besides our work, the ranking metric may also help other concurrency testing frameworks such as CHESS to improve testing efficiency.
- Minimum external control to force low-probability interleavings during testing on multicore machines. Unlike CHESS and RaceFuzzer that control execution by scheduling one thread at a time, CTrigger inserts artificial synchronizations (with an expiration time) in only a small set of execution points corresponding to the target interleavings of interests. This allows the tested program to leverage multiple cores. It also avoids slowing down execution periods that are unrelated to the target interleavings.

We evaluate CTrigger on eight-core machines with real-world bugs from three SPLASH2 benchmarks and four server/desktop open-source programs, including MySQL, Apache, Mozilla, and PBZIP2. Among them, MySQL, Apache, and Mozilla are widely used large open-source programs with up to 3.4 million lines of code. CTrigger exposes the tested atomicity-violation bugs 10-1,000 times faster than stress testing and previous methods (both synchronization-based and race-based techniques described in Section 1.2). For example, CTrigger takes 63 and 235 seconds, respectively, to expose the two real-world Apache bugs, whereas the stress testing requires more than 20 hours to expose them, and one of the bugs never manifests after *one week* of stress testing!

As explained before, testing efficiency is very important due to the time pressure in software release. A speedup of 10-1,000 would be very beneficial. For example, 100 different 1-hour-long (under CTrigger) input test cases and 10 different configurations would take CTrigger two days to finish on 20 machines, whereas it will take stress testing 20-2,000 days to achieve similar exposing capability for atomicity-violation bugs, which is definitely too long to be acceptable.

In addition, since CTrigger records the execution control that exposes a bug, it can perform the same control to reliably reexpose the same bug for diagnosis without any deterministic replay support. For the tested bugs, CTrigger reexposes them mostly within 5 seconds, which is 300 to more than 60,000 times faster than stress testing.

The remainder of this paper is organized as follows: Section 2 shows the background of this paper. Section 3 designs and evaluates a hierarchy of interleaving coverage criteria, from which we select the best one (unserializable interleaving coverage criterion) to guide our CTrigger design. In Section 4, we present why stress testing cannot effectively satisfy the unserializable interleaving coverage criterion and expose atomicity-violation bugs. Sections 5 and 6 propose CTrigger framework. Section 8 presents experimental results. Section 9 discusses related work, followed by Section 10 concluding this paper.

2 Background

2.1 Atomicity-Violation Bugs

Atomicity, also called **serializability**, is a property for a sequence of instructions S in multithreaded software. If the outcome of S in concurrent execution is equivalent to the outcome of any serial execution of S, the atomicity of S is preserved. Otherwise, the atomicity is violated [7], [22], [46].

Atomicity-violation bugs are introduced when programmers expect some code regions to always behave atomically without enforcing the atomicity in their implementation. As a result, the expected atomicity can be broken at runtime and make software misbehave. We will refer to the code region that is expected to be atomic as an **expected atomicity unit**, or **atomicity unit** for short.

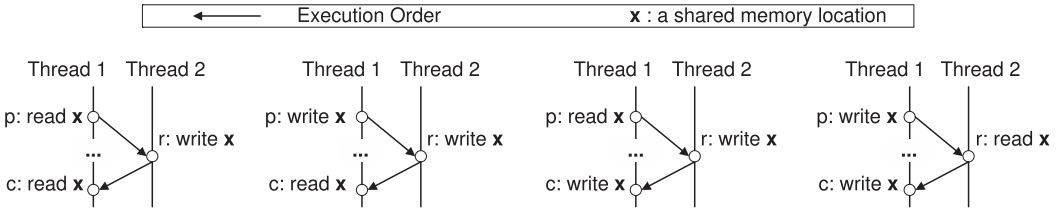


Fig. 3. Unserializable interleavings. (A static instruction C's unserializable interleaving is exercised iff at least one of its dynamic instances follows above pattern during execution.)

The atomicity units in most real-world atomicity-violation bugs simply include two accesses [22], [39]. They are two consecutive accesses to the same variable from one thread, such as the accesses to `buf_index` on S1 and S2 in Fig. 1. For ease of discussion, we will refer to the earlier one in the atomicity unit as preceding-access, p-access for short, and the later one as current-access, c-access for short.

There are, in total, four scenarios where the atomicity of p-c unit can be violated [22]. All four scenarios are depicted in Fig. 3. To simplify our discussion, we will refer to the access from another thread that violates the atomicity of p-c unit as remote-access, r-access for short. For example, the write to `buf_index` on $S3$ in Fig. 1 is the r-access.

2.2 Test Coverage Criteria

The number of possible test cases for a program is huge. Therefore, practical testing has to select representative test cases to focus on. This is usually guided by a *test coverage criterion*. Different coverage criteria target different program spaces, e.g., the input space, the interleaving space, etc. A criterion includes two parts. One is a set of program properties, which could be program statements, program branches, etc. The other is a property-satisfaction function f, indicating which test cases satisfy (exercise) a certain program property. The adequacy of testing is measured by how many program properties are satisfied. If all the properties are satisfied, testing achieves *complete coverage* and is called a *complete testing* under this coverage criterion.

Complexity and *bug-exposing capability* are the two most important metrics for a coverage criterion. A coverage criterion's complexity can be measured by the number of test cases (inputs, interleavings, etc.) that are needed to exercise all the properties [44]. Bug-exposing capability can be experimentally measured by fault coverage [25], i.e., the percentage of bugs that can be reliably exposed by a complete testing under the coverage criterion.

It is difficult to reach a good balance between complexity and bug-exposing capability. People usually compose hierarchical families of coverage criteria [3], [9] to acquire a thorough understanding of the design tradeoffs. In general, good criteria should be based on valid bug models. For example, structural coverage criteria are based on the bug model that most sequential bugs are related to certain program structures and control flows.

3 Interleaving Coverage Criteria

How to select representative interleavings from the huge interleaving space is an open problem. This section first designs and evaluates a hierarchy of interleaving coverage criteria. Based on the evaluation results, it then presents a practical and well-balanced coverage criterion for CTrigger to expose atomicity-violation bugs.

3.1 A Hierarchy of Interleaving Coverage Criteria

This section presents a hierarchy of four interleaving coverage criteria. These criteria are designed based on different concurrency bug models and cover a wide spectrum of testing complexity and bug-exposing capabilities. It will help us understand the whole design space of interleaving testing.

In the following discussion, we consider a concurrent program P, executed under an input I, consisting of M threads $(1, 2, \ldots, M)$. Similarly to previous work [43], we model the concurrent execution of P by a total ordered sequence of shared variable access events. At any moment, only one thread i is active and executes one event. Each thread i in total executes N_i events. The event order within each thread is fixed, and the order among different threads might change. Each total order among all the events is called an **interleaving**.

Note that the set of events executed by P under I could vary slightly from one run to another. Since it does not change the complexity comparison among different coverage criteria presented below, we ignore it for the simplicity of the discussion. We will consider this issue when we design and implement CTrigger (Section 5).

- **Criterion 1: all-interleavings (ALL).** Each interleaving (i.e., the total order among all accesses) is one testing property in ALL, as shown in Fig. 4. An interleaving testing is "complete" iff all feasible interleavings under input I are covered.

  ALL clearly has high coverage in exposing concurrency bugs, but it is also clearly impractical for real-world applications—its testing space is exponential in the number of accesses in each thread and exponential to the number of threads. For example, even for the toy program execution shown in Fig. 4, the ALL testing space is as large as $34{,}650 = \binom{12}{4}\binom{8}{4}\binom{4}{4}$. ALL is similar to early proposed

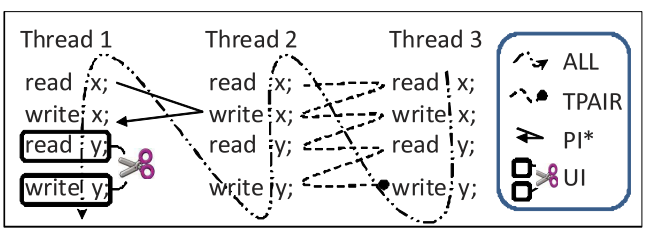


Fig. 4. Different interleaving coverage criteria and their testing properties. (*: The PI coverage criterion shown above is the one customized for atomicity-violation bugs.)

coverage criteria that attempted to cover all possible interleavings or all possible states of a concurrent program [12], [14], [16], [38].

- **Criterion 2: thread-pair-interleavings (TPAIR).** A total-order among two threads is one testing property, as shown in Fig. 4. Interleaving space gets "complete coverage" under TPAIR iff all feasible interleavings of all shared memory accesses from any pair of threads are covered.

  The assumption behind TPAIR is that most concurrency bugs are caused by the interaction between two threads, instead of an arbitrarily large number of threads. This is a widely believed concurrency bug model in previous bug detection and testing work [17], [22], [33].

  The testing space of TPAIR can be calculated as $\sum_{1\le i<j\le M}\binom{N_i+N_j}{N_i}$. The complexity of TPAIR is polynomial in the number of threads and exponential in the number of memory accesses from each thread. For the example shown in Fig. 4, the size of its TPAIR testig space is $210 = \binom{3}{2}\binom{8}{4}\binom{4}{4}$. Obviously, TPAIR is much simpler than ALL, but still unscalable for large real-world programs.
- **Criterion 3: partial-interleavings (PI).** PI further reduces the complexity by defining each testing property to be the execution order among a small number (constant K) of accesses. The interleaving is "complete" iff all feasible orders among all groups of K accesses are covered.

  The design of PI is based on the observation that the manifestation of many concurrency bugs, such as data races and atomicity-violation bugs, can be guaranteed by wrong execution orders among only two or three accesses. For example, as discussed in Section 2.1, many atomicity-violation bugs are caused by the wrong order among three accesses [22], [39]. We can easily customize PI for atomicity-violation bugs by testing the order of three accesses from any pair of threads to the same memory location, as shown in Fig. 4.

  The size of PI's testing space is at most $\binom{\sum_i N_i}{K}$, polynomial in the number of memory accesses. Therefore, PI is much easier to use in practice than TPAIR and ALL. For the example shown in Fig. 4, its atomicity-customized PI space has only 12 testing properties, much simpler than TPAIR and ALL. Further shrinking the testing space is difficult for general concurrency bugs, but is feasible for atomicity bugs.
- **Criterion 4: unserializability interleavings (UI).** Each testing property of UI is associated with one memory access c, as shown in Fig. 4. A property is covered iff c and its preceding access p to the same memory location from the same thread is unserializably interleaved. A testing is "complete" under UI iff every memory access' corresponding testing property is covered, if feasible.

Obviously, the testing space of UI is linear in the number of memory accesses. For example, the UI space only has six testing properties in Fig. 4. UI is clearly the simplest among all coverage criteria discussed in this section. It also fits atomicity-violation bugs wel, because it captures the essence of atomicity violation.

3.2 Evaluating the Coverage Criteria

3.2.1 Methodology and Caveat

In order to evaluate the bug-exposing capabilities of coverage criteria, the most convincing method is to check against real-world buggy applications, as done in previous data-flow coverage criteria evaluation [44]. Unfortunately, so far there have been no representative real-world concurrency bug benchmarks. Facing this challenge, we conducted a *best effort* real-world concurrency bug characteristic study and evaluated coverage criteria accordingly.

We selected four representative open-source applications in our study: MySQL, Apache, Mozilla, and OpenOffice. These are all mature (with 9-13 years development history) large concurrent applications (with one to four million lines of code). From their well-maintained bug databases, we first used keyword ("race(s)," "deadlock(s)," "synchronization(s)," "concurrency," "lock(s)," "mutex(es)," "atomic," "compete(s)," and their variations) search to *randomly* collect about 500 reports of fixed bugs. Then, we manually checked every reports and finally got **105** bugs (71 non-deadlock and 34 deadlock bugs) that were really caused by programmers' wrong assumptions about concurrent execution. Finally, for each of these 105 concurrency bugs, we manually checked what the conditions for them to manifest were when the program was executed under a bug-triggering input. We made our best effort to make sure to fully understand these bugs based on the developers' discussion, source code, the final patches.

We should note that the characteristics study results presented below should be interpreted with the above methodology in mind, and may not generalize to all programs. However, we believe that our study does capture the characteristics of concurrency bugs in two large important classes of concurrent applications: server-based and client-based applications. In addition, most of these characteristics are consistent across all four examined applications. Furthermore, we do not emphasize any quantitative characteristic results.

3.2.2 Characteristics and the Implications to Testing

> **Characteristic 1:** The manifestation of most (101 out of 105) examined concurrency bugs involves no more than two threads.

Implications. TPAIR coverage criterion can guide interleaving testing to expose most concurrency bugs.

The underlying reason for this characteristic is that a thread usually does *not* closely interact with many others, and most collaboration is conducted between two or a small group of threads. As a result, the manifestation of most concurrency bugs only involves a few (mostly two) threads.

> **Characteristic 2.1:** 90% (67 out of 74) of the examined non-deadlock bugs can deterministically manifest, if certain orders among at most four memory accesses are enforced.
>
> **Characteristic 2.2:** 97% (30 out of 31) of the examined deadlock bugs can deterministically manifest, if certain orders among at most four resource acquisition/release are enforced.

Implication. Interleaving testing can focus on different execution orders among small number of memory

TABLE 1
Applications and Workloads

App.	LOC	Description	Synch.(#)	Workload
Apache	302K	Web server	lock(224)	SURGE [2]
MySQL	1.9M	Database server	lock(1020)	MySQL-test*
Mozilla -js	270K	Web browser suite	lock(37)	JavaScript test suite*
PBZIP2	2.0K	File compressor	lock(21) & queue	a random file
FFT	1.0K	FFT transformation	barrier(7)	default setting with 8 processors
LU	1.0K	Matrix factorization	barrier(5)	
Barnes	3.0K	N-body problem	lock(6) & queue	

*: MySQL-test and JavaScript test suite are designed by the application developers. The Sync column shows the number of static instances of synchronizations in each application in parentheses.

accesses. That is, PI is an effective simplification from ALL and TPAIR.

The characteristics 2.1 and 2.2 are consistent with previous concurrency bug studies: Data races [31], [33] only involve two conflicting accesses, atomicity-violation bugs [22], [39] frequently only involve three accesses, the most common types of deadlocks are lock order inversion between two locks.

Characteristic 3: For most (44 out of 51) of the examined atomicity-violation bugs, software failures are guaranteed once the atomicity of certain code unit from one thread is violated. How many threads and how many accesses participate in violating the atomicity do not matter.

Implication. *UI reaches a nice balance between simplicity and exposing capability for atomicity-violation bugs.*

In summary, assuming the representativeness of our randomly selected concurrency bugs and the existence of potential bug-triggering inputs, the hierarchy of ALL, TPAIR, PI, and UI provides a rich set of testing choices, with decreasing complexity and decreasing bug-exposing capability. If we focus on atomicity-violation bugs, UI provides the best complexity-capability trade-off and is most suitable for practical use.

3.3 Unserializably Interleaving Space for CTrigger

Following the above coverage criteria study, we pick UI interleaving coverage criterion to guide CTrigger testing.

Specifically, for every shared memory access instruction C, CTrigger testing will try to exercise at least one interleaving, where one dynamic instance of C and its preceding access are unserializably interleaved (shown in Fig. 3). We refer to such an interleaving as *interleaving-C*, short for an unserializable interleaving with instruction C as the current access. We accordingly define CTrigger's testing space as {interleaving-$C \mid C$ is a shared memory access instruction}. Within this space, some unserializable interleavings may never happen during actual execution due to synchronization constraints in the program. We will discuss how to prune out these *infeasible interleavings* in later sections.

The unserializable interleaving space defined above is linear in the static size of the program. It is much smaller than the entire interleaving space and is therefore practical to thoroughly explore. In the meantime, unserializable interleaving space gives a good coverage for all potential atomicity-violation bugs. Covering this space during testing would give developers at least some level of confidence on their software quality against atomicity violations.

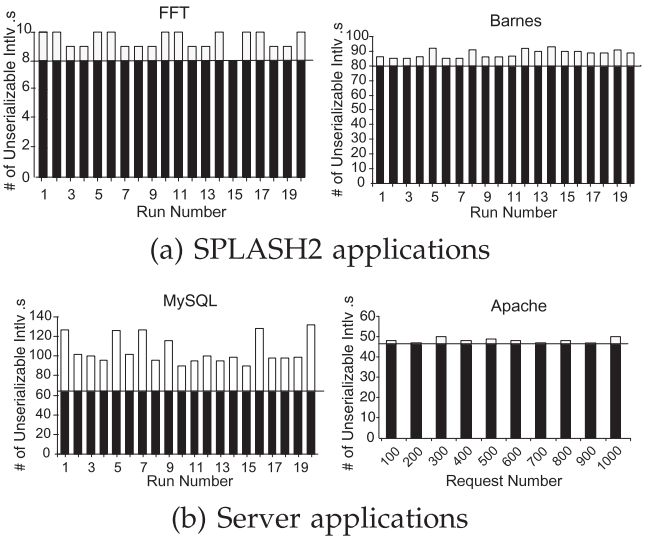


Fig. 5. Similarity between runs: Each bar shows the number of unserializable interleavings covered in each run. The dark part shows the number of interleavings exercised by *all* runs, i.e., having 100 percent occurrence frequency. The remainder (those with less than 100 percent frequency) are shown in the white part.

4 Why Stress Testing is Not Good: An Interleaving Characteristic Study

Stress testing is the current dominant practice. Relying on the nondeterministic behavior of concurrent programs, it executes the program with the same input many times without any execution control. To understand why it is ineffective at exposing atomicity-violation bugs, we quantitatively study its characteristics from the perspective of an unserializable interleaving space. This understanding will guide our design of CTrigger.

4.1 Methodology of the Characteristics Study

We used four widely used open-source server/desktop applications, Apache, MySQL, Mozilla, and PBZIP2, and three SPLASH2 [45] benchmarks. These applications cover different types of functionalities and synchronization models, as shown in Table 1. The experiments used a dual quad-core (eight processors in total) Intel Xeon machine, and each application was configured to have eight worker threads. For server applications, we took common configuration options and used a realistic workload generator or the test cases designed by developers.

To collect the interleaving information, we used the Pin binary instrumentation tool [24] to monitor the execution. To make sure that our study can reflect the real non-perturbed execution environment, we carefully designed our instrumentation to give minimum perturbation in a thread-balanced way.

4.2 Observations

Our experimental results reveal the following observations:

1. *Is stress testing nondeterministic in a random way?* From the perspective of covering unserializable interleavings, the answer is *no*. As shown in Fig. 5, the majority of unserializable interleavings exercised by different runs (or different iterations for server programs) are the same.

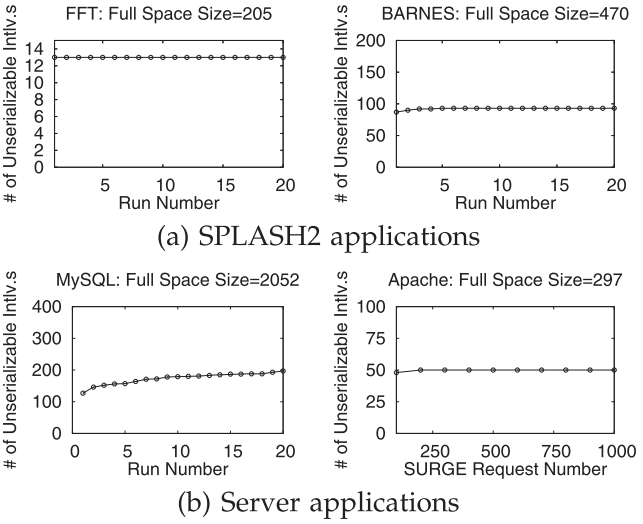


Fig. 6. The accumulative set of exercised unserializable interleavings grows slowly after the first few runs. (Other applications that are not shown here also have similar behaviors. The *full space* includes all potential unserializable interleavings. We will discuss how to calculate that in Section 5.1.)

2. *Can we rely on stress testing to cover the whole unserializable interleaving space?* The answer is *no*. As shown in Fig. 6, stress testing exercises hardly any new unserializable interleaving after the first few runs and leaves some feasible unserializable interleavings uncovered in every application. Actually, some feasible interleavings are never exercised in days of stress testing. These interleavings are exactly the most obnoxious ones that usually hide difficult-to-detect and tough-to-diagnose atomicity-violation bugs.
3. *Why do some unserializable interleavings have low probability of being exercised?* Different interleavings have completely different occurrence probabilities. For example, Fig. 5 shows that some interleavings are exercised in all stress testing runs, i.e., about 100 percent occurrence probability. On the contrary, some interleavings are never exercised during days of experiment, i.e., almost 0 percent probability. Further examination reveals the following major factors determining the probability: a) program synchronizations, such as lock and barrier, make some interleavings always happen and some never happen; b) distances between related instructions: when two memory accesses from a thread are close to each other, the chance is small for them to be unserializably interleaved by a remote conflicting access; and c) the number of dynamic instances of a static instruction: the more dynamic instances a static instruction has, the more likely that one of them will be unserializably interleaved.

4.3 Implications for Exposing Atomicity Bugs

In summary, we can see that stress testing is not good at exposing atomicity-violation bugs because it cannot effectively exercise the unserializable interleaving space. Without perturbation to the execution, stress testing repeatedly tests those high-probability unserializable interleavings. *Atomicity-violation bugs can easily hide in those low-probability unserializable interleavings and escape into production runs.* Such bugs are usually the most obnoxious, difficult-to-catch, and tough-to-diagnose concurrency bugs due to their rare occurrences (without external control) [19], [21], [22], [46].

Recently several projects have studied ways to improve stress testing based on simple heuristic-based execution perturbation, e.g., random delay [18] or thread scheduling at synchronization points [4], [26], [27], [37]. Unfortunately, however, those also have limitation in exposing atomicity violation, as we discussed in Section 1.2.

5 CTrigger Phase One: Identify Target Unserializable Interleavings

Based on the observations described in previous sections, we design a framework called CTrigger to expose hidden atomicity-violation bugs in concurrent programs. CTrigger testing includes two phases for each concurrent program and each test input (shown in Fig. 2): At the first phase, it conducts trace analysis to obtain a list of unserializable interleavings for exploration; at the second phase, it explores these unserializable interleavings by controlled testing and exposes hidden atomicity-violation bugs.

In this section, we discuss how CTrigger obtains the target unserializable interleaving list through three steps (marked as Steps 1, 2, and 3 in Fig. 2). We will discuss the second phase in the next section.

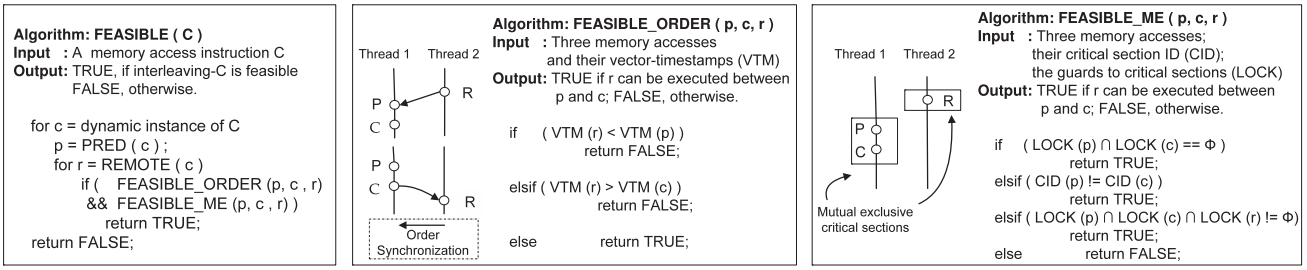
Please note that we take similar assumptions with recent work on concurrency testing [6], [26], [35]. We assume that programmers have a test-case suite. Each test case includes a *test input* and a *test oracle* that specifies the correct execution result. Programmers will go through CTrigger's phases one and two for each test case. We also assume that, for one input, the set of code statements executed at different runs is mostly the same. During our experiments, we measured each application's statement coverage of different runs under the same input. The results have confirmed our assumption.

5.1 Step 1: Identifying Potential Unserializable Interleavings

In CTrigger, we use a few profiling runs with a given test input to collect memory access information and conduct trace analysis to build the initial list of unserializable interleavings, which consists of potential (may not all be feasible) unserializable interleavings.

Each unserializable interleaving is composed of three accesses, a p(receding)-access, a c(urrent)-access, and an r(emote)-access (refer to Section 2). Therefore, as a first step, CTrigger goes through every memory access instruction C and checks whether C has a p-access and an r-access. If so, we identify interleaving-C as a *potential* unserializable interleaving. In our study, this step is based on profiling. Potentially, it can also be done via static analysis.

Usually, only a couple of runs are needed for this profiling stage since the trace analysis can infer what unserializable interleavings might occur in other runs. In our experiments in Section 8, we run each application only once for profiling.



(a) algorithm to distinguish feasible unserializable interleaving

(b) infeasible interleavings caused by order synchronization and algorithm to prune them

(c) infeasible interleavings caused by mutual exclusion and algorithm to prune them

Fig. 7. CTrigger feasible interleaving analysis algorithm. (PRED and REMOTE denote preceding access(es) and remote accesses(es), respectively. They are collected in step 1. VTM stands for vector-time-stamps; LOCK are lock sets protecting each memory access; CID stands for critical-section-ID.)

5.2 Step 2: Pruning Infeasible Interleavings

Among the potential unserializable interleavings, some can never happen (i.e., r-access cannot execute between p and c) due to synchronizations. It is important to prune them to avoid the vain attempt to force them. In this section, we categorize all synchronization operations into two types, *order synchronization* and *mutual exclusion*, and design pruning algorithms accordingly (shown in Fig. 7). CTrigger also prunes other types of infeasible interleavings, such as those caused by memory recycling.

5.2.1 Algorithms and Implementations

Infeasible interleavings caused by order synchronization. An order synchronization operation, e.g., a barrier and a thread create/join, forces certain order between events from different threads. Therefore, if an r-access is separated from p and c-accesses by order synchronizations, it can never be executed between them. By checking this condition, CTrigger can prune out such infeasible interleavings. The process is shown on Fig. 7b.

In our implementation, CTrigger records all barrier and thread-create/join operations into the trace. In trace analysis, CTrigger uses vector time stamps to maintain and compare the order relationship between accesses. Note that some conventional happens-before race detection algorithms [31] update the vector time stamps at not only order synchronization operations but also mutual exclusion synchronization operations such as lock and unlock. CTrigger ignores the latter because lock/unlock does not force absolute orders.

Infeasible interleavings caused by mutual exclusion. Synchronization primitives like locks and transactions provide mutual exclusion in concurrent programs. Considering this type of synchronization, an r-access cannot interleave a p and a c-access iff there exist two mutual exclusive critical regions that one holds the r and the other holds both the p and c. Following this, we can prune infeasible interleavings caused by mutual exclusions (Fig. 7c).

Specifically, CTrigger records all lock/unlock operations into the trace. During trace analysis, CTrigger maintains a lock set for each shared memory access and uses that to determine which critical section(s) the access belongs to. Note that traditional lock-set race-detection algorithm [33] is not sufficient for CTrigger because two accesses protected by the same lock variable may not be inside the same critical section (e.g., `lock(L)` $S1$ `unlock(L)... lock(L)` $S1$ `unlock(L)`). To handle this, CTrigger maintains a global counter that is incremented at every lock operation during tracing. It can uniquely identify each critical section and help CTrigger analysis.

Memory recycling issue. CTrigger also considers infeasible interleavings caused by memory address recycling. Specifically, two different program variables may be assigned to one memory address during the course of execution due to memory recycling. The instructions using such variables actually can never conflict with each other. CTrigger prunes this type of infeasible interleavings by intercepting memory allocation and deallocation operations and differentiating memory locations allocated at different time.

5.2.2 Discussions

CTrigger works well for real-world server programs written in C, as we will see in the experiments (Section 8). Most infeasible interleavings can be correctly identified. However, a small number of infeasible interleavings may be missed due to unidentified customized synchronization operations. This is handled at CTrigger's second phase: When trying to force an interleaving, CTrigger sets an expiration time for each artificial delay. Once the time expires, CTrigger gives up and continues exploring other interleavings. Since most infeasible interleavings are pruned, the wasted effort is tiny.

Our current prototype can also be extended to consider other synchronization operations. *Transactions* can be used for mutual exclusion and can be analyzed similarly as *locks*. *Semaphores* can be used for both mutual exclusion and order synchronization. We will need to first differentiate these two usage scenarios and then apply the above algorithms accordingly. *While-loops* (i.e., busy-waiting loops) are frequently used for order synchronization. If developers or static analysis tools can annotate those loops, CTrigger can apply the above order-synchronization-based pruning algorithm to them—we simply need to set up a happens-before order between the memory write access that provides the loop-terminating value and the read access in the last iteration of the loop that gets that value.

5.3 Step 3: Ranking Low-Probability Interleavings

As discussed in Section 4.2, different interleavings have different occurrence probabilities during stress testing. Some interleavings rarely occur but have high likelihood of exposing atomicity-violation bugs, especially those that are hard to reproduce for diagnosis. Therefore, it is

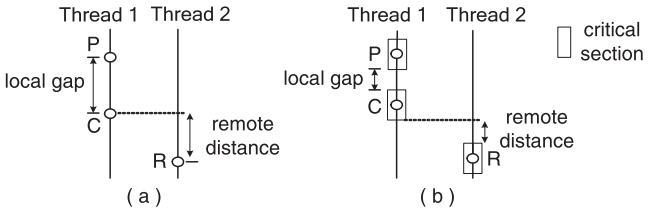


Fig. 8. Local gap and remote distance.

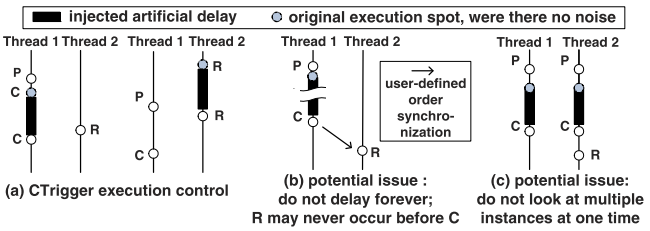


Fig. 9. CTrigger's execution control and design issues.

desirable to identify and prioritize low-probability interleavings during testing in order to effectively expose bugs.

In this section, we first discuss the major factors that affect the probability of interleavings. We will then introduce our probability ranking metrics and explain the detailed ranking algorithms. Note that accurately calculating the interleaving probability is difficult and also unnecessary. CTrigger aims at using simple and yet effective metrics to select low-probability interleavings.

5.3.1 Two Major Factors for Occurrence Probability

The occurrence probability of an unserializable interleaving is affected by many factors. Among them, two factors are most important: how close the two local accesses (p and c-accesses) are, and how far away a remote access is from the local accesses. Intuitively, when a p and a c-access are close to each other, the time window can be too small for a remote access (to the same memory location) to interleave in between. Similarly, when a remote access is far away from the local accesses, the chance of an interleaving is small.

Based on the intuition above, we define the following two simple metrics to estimate the probabilities and to rank the unserializable interleavings (Fig. 8).

- **Local gap** is the execution time distance between a p-access and a c-access for an unserializable interleaving (p, c, r) as defined in Section 2. This metric represents the size of an *interleavable window*, i.e., the period where an r-access can interleave between the p and c-accesses.
- **Remote distance** is the time difference between an interleavable window and an r-access. As remote distance increases, the r-access gets farther from the interleavable window and is less likely to interleave the p and c.

There is a big difference between the two metrics: The local gap is stable across runs as it only involves one thread; the stability of remote distance highly depends on the nature of applications. Currently, CTrigger uses the local gap as the primary ranking metric, and refers to the remote distance only when multiple interleavings have similar local gaps.

5.3.2 How to Compute the Metrics?

The main idea of the CTrigger ranking mechanism is straightforward. CTrigger first analyzes the profiling-run traces and gets the local gap for every unserializable interleaving. It then generates a ranking based on the local gaps: the smaller a local gap is, the higher an interleaving is ranked. Although the basic idea is simple, there are several issues we need to address:

How to measure the distance? We use CPU performance counter (accessible through `RDTSC` x86 assembly instruction) to measure local gaps. This scheme can include the different latencies of different operations, such as disk I/O, into gap information. Currently, we do not consider the effect of context switches in local gap measurement. Fortunately, the time slice for preemptive context switches is very large, so only a few instructions will be affected.

How to deal with synchronizations between local accesses? Synchronization operations would affect the effective interleavable windows and thereby should be considered when calculating local gaps. For example, when each of p, c, and r accesses is protected by a same lock *separately* (Fig. 8b), the local gap should be the execution period starting from the end of p's critical section to the beginning of c's critical section because r cannot be concurrently executed with critical sections that contains p or c.

How to deal with multiple instances of the same static instruction? The more dynamic instances a static instruction has, the more likely an interleaving would occur. Therefore, CTrigger takes *the summation* of all local gaps from all the dynamic instances of an unserializable interleaving.

In the end, CTrigger gets a list of likely feasible unserializable interleavings ranked based on estimated occurrence probability. It further excludes the interleavings that are already exercised during the profiling runs, and delivers the remaining list to its next phase.

6 CTrigger Phase Two: Explore Unserializable Interleaving Space

In this phase, CTrigger systematically controls the concurrent execution in order to exercise the unserializable interleavings identified and ranked in phase one, starting from the ones with the lowest (estimated) occurrence-probabilities.

6.1 Execution Control for One Interleaving

Unlike previous work, such as CHESS [26] and RaceFuzzer [35], that controls thread schedule and executes only one thread at a time, CTrigger suspends a thread's execution at selected places to increase the occurrence probability of the target unserializable interleaving. The duration of the suspension is carefully controlled to avoid significant performance degradation.

Specifically, for an unserializable interleaving, CTrigger suspends a thread before it executes c-access C or r-access R (Fig. 9a) whenever necessary to increase the *local gap*, decrease the *remote distance*, and therefore increase the occurrence probability of the target unserializable interleaving.

Although the above ideas are intuitive, there are several efficiency and effectiveness issues to address:

1. *How long is the suspension?* An intuitive answer is to suspend the execution until the atomicity violation occurs, which means suspending C's thread until R executes or suspending R's thread until C is ready to execute. Unfortunately, the unserializable interleaving may never occur, as shown in Fig. 9b. To avoid such endless suspension (deadlock), CTrigger sets a time-out threshold for each suspension point. In our experiments in Section 8, the threshold is set to 200 microseconds.
2. *When should a thread be suspended?* An intuitive answer is to suspend a thread when it is about to perform the c or r-access. This intuitive solution has problems. First, when there are two or more threads that can execute C, suspending all of them may decrease the interleaving probability (Fig. 9c). Therefore, CTrigger only suspends one thread at a time. Second, a static instruction might have many dynamic instances. Suspending before every instance can result in huge slowdowns. For efficiency, CTrigger sets a threshold for the number of times that threads are suspended for each unserializable interleaving.
3. *The danger of waiting inside a critical region.* Suspending a thread inside critical sections might also block other threads that are waiting to enter critical sections. Although it will not lead to a deadlock, as CTrigger has an expiration time for each suspension, it may prevent the target interleavings from happening. CTrigger can address this issue by suspending the execution right before the outermost critical section that holds the target instruction.
4. *Context sensitivity.* The occurrence of some unserializable interleavings depends on the program context. For example, p and c may be well protected by the same lock under one calling context, but not protected under another calling context. In order to more efficiently expose the atomicity violation that can happen only in the latter context, CTrigger's first phase provides the option to record the entire call stack and thread information for each access in the trace. Therefore, using the detailed context information, CTrigger's execution control can explore the interleavings only in the bug-triggering context. In our experiments, however, we have found that CTrigger can still effectively expose bugs without any call-stack information.

6.2 Execution Control for a List of Interleavings

Controlled testing for a ranked list of unserializable interleavings is a complex planning problem because exploring one interleaving might interfere with the exploration of another interleaving. Facing this problem, CTrigger follows a simple principle—one interleaving at a time. After the target interleaving occurs or the time expires, it moves on to the next interleaving. Note that it does not mean one interleaving per run. Each run can still explore multiple target interleavings.

As regards which one to explore first, CTrigger provides two options. The first option is to simply go down the ranked list and explore unserializable interleavings one by one. While simple, it may be inefficient if a high-ranking interleaving appears late during the execution. The second option is to consider a set of interleavings with similar ranks at a time. CTrigger suspends execution for whichever interleavings whose involving instructions appear first. In our experiments, we use the first option for a short list of unserializable interleavings (such as those in SPLASH2 applications) and the second option for a long list of unserializable interleavings (such as those in server applications).

6.3 Implementation

CTrigger controls execution via binary instrumentation using Pin [24]. It takes the list of unserializable interleavings provided by the CTrigger analysis and instruments every instruction that involves in at least one unserializable interleaving. At runtime, CTrigger intercepts every dynamic instance of these instructions and injects delay following the above strategies.

6.4 Outcome Interpretation

During the controlled execution, CTrigger monitors shared memory accesses to check whether an atomicity violation has happened.

If a target unserializable interleaving is successfully forced by CTrigger and the software misbehaves (e.g., crashes, making different results from test oracles, errors detected by bug detectors), an atomicity-violation bug is exposed. Since CTrigger records the execution control added by it during every testing run, once a bug is exposed, CTrigger can reliably reproduce the bug by retrying the execution control that it added during the previous bug-triggering run.

If a target unserializable interleaving is successfully forced but the software does not misbehave, benign atomicity violations are identified. In this case, programmers gain more confidence about the software quality.

If the target interleaving does not happen after the controlled execution, most likely the interleaving is actually infeasible due to customized synchronization operations that are not identified in our trace analysis. Such information is still useful as it can help identify customized synchronization operations, which will help concurrent program analysis.

7 Methodology

To evaluate our ideas and CTrigger framework, we applied CTrigger on seven applications, including three large open-source server/client applications. The details about applications used are shown in Table 1. We evaluated one or two *real-world* atomicity-violation bugs in each application which are described in Table 2.[1] Meanwhile, CTrigger could find four new buggy code regions in Apache (Fig. 10a), which have never been reported.

The platform setting is the same as that in Section 4.1. The selection of testing inputs for the server/client applications was based on the original bug reports on

1. CTrigger exposed one *previously unknown* bug in the macro library of SPLASH2 introduced by external macro providers.

TABLE 2
Evaluated Applications and Atomicity-Violation Bugs

App.	Bug Id.	Bug description
Apache	Apache#1	Server crash during cache management
	Apache#2	Log-file corruption
MySQL	MySQL	DB log missing database actions
Mozilla	Mozilla*	Wrong results of JavaScript execution
PBZIP2	PBZIP2	Crash during file decompression
FFT	FFT	A problem in platform-dependent macro (introduced by macro providers) leading to atomicity-violation bugs that generate wrong outputs
LU	LU	
Barnes	Barnes	

**: Mozilla code Is slightly modified to help compare the execution result with the test oracle.*

TABLE 3
Evaluated Concurrency Testing Methods

Stress	Stress testing
Pure-Pin	Stress testing running upon the Pin binary instrumentation framework. This is the baseline for the next three schemes, which are all implemented by us upon Pin.
Sync-based	A bug exposing mechanism that injects delay at synchronization operations just like ConTest [4]. The released version of CHESS [26] is similar, i.e. also sync-based.
Race-based	A bug exposing mechanism that forces suspect data races reported by a race detector. This is similar to RaceFuzzer [35]. Our implementation is based on Pin and the state-of-the-art open-source Valgrind-lockset race detection tool [30]. It is extended by our execution control to run multi-threads concurrently instead of one thread at a time like in the original RaceFuzzer.
CTrigger	Our method presented in this paper

corresponding forums (since CTrigger focuses on testing the interleaving space, not the inputs, figuring out the bug-triggering inputs is out of our scope).

Note that, for all bugs, CTrigger does not assume any prior knowledge about the bug-triggering interleavings. It strictly follows the process described in previous sections to systematically identify and exercise low-probability unserializable interleavings. For instance, we did **not** know about the existence of the SPLASH2 macro bugs in advance. They were exposed by CTrigger under testing with the default inputs.

8 Evaluation

This section presents the following experimental results about the effectiveness, efficiency, and reproducibility of CTrigger:

- We evaluate whether tested bugs can be exposed and how quickly they can be exposed in Section 8.1. In this part of evaluation, we compare CTrigger with four other bug-exposing mechanisms on the same platform as shown in Table 3.
- We evaluate how CTrigger improves the unserializable interleaving coverage during its testing process in Section 8.2.
- We evaluate how reliably a bug can be reproduced after its first manifestation in Section 8.3.
- We evaluate the impact of CTrigger infeasible interleaving pruning and CTrigger low-probability interleaving ranking in Sections 8.4 and 8.5.

8.1 Efficiency and Effectiveness

8.1.1 Bug-Exposing Time

Overall, as shown in Table 4, CTrigger can expose all the tested atomicity-violation bugs efficiently, within 1-235 seconds. It is about 10 to over 1,000 times faster than all alternative testing methods for all tested bugs, except for Apache#2, MySQL, and PBZIP2 bugs, where its efficiency is comparable with Race-based testing. CTrigger is especially effective for large server/client applications. For example, CTrigger needs just 4 minutes to expose Apache bug#1, which can**not** be exposed by any alternative testing schemes

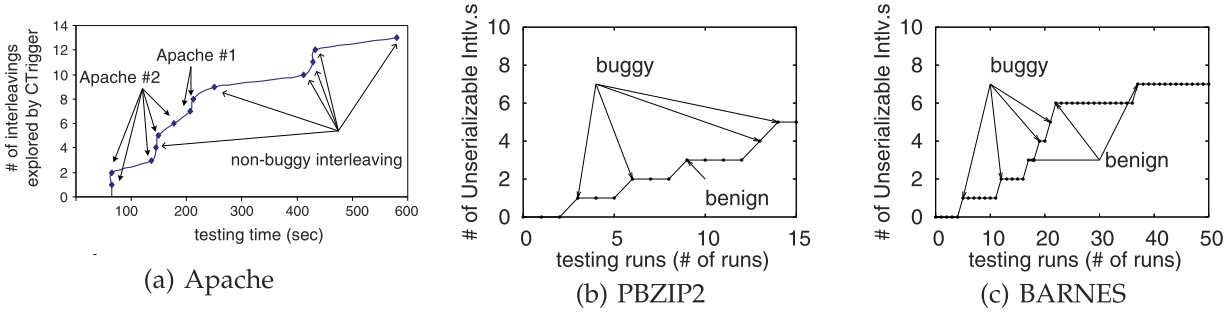


Fig. 10. Unserializable interleavings additionally explored by CTrigger. (The base line is the unserializable interleavings covered in profiling runs. The first 60 seconds in Apache are devoted to profiling runs and have no additional coverage.)

TABLE 4
The Time (Unit: Second) Spent to Expose the Tested Atomicity-Violation Bugs

BugId.	Stress	Pure-Pin	Synch-based	Race-based	**CTrigger**	CTrigger Speedup*
Apache#1	> **1 week**	NO	NO	NO	235.0	> 2573.6 X
Apache#2	80604.0	NO	14976.0	126.0	63.6	1267.4 X
MySQL	287.0	5431.0	3796.0	3.5	2.0	143.5 X
Mozilla	NO	NO	NO	65759.6	66.2	> 1305.1 X
PBZIP2	NO	NO	32.0	2.6	2.6	> 9391.3 X
FFT	673.0	2284	NO	NO	0.94	716.0 X
LU	188.6	3459	NO	NO	4.2	44.9 X
Barnes	248.7	NO	NO	NO	17.6	14.1 X

***NO**: the bug was not exposed in our maximum testing time, i.e., one day for Apache, MySQL, and Mozilla, half day for other small applications.*
**: compared with stress testing.*

TABLE 5
Breakdown of CTrigger Bug-Exposing Time
(Unit: Second)

BugId.	Profiling Runs	CTrigger Analysis	Controlled Testing
Apache#1	61.4	1.1	172.5
Apache#2	61.4	1.1	1.1
MySQL	0.90	0.10	0.90
Mozilla	8.0	1.0	57.2
PBZIP2	0.56	0.0006	2.01
FFT	0.52	0.23	0.19
LU	1.40	2.58	0.18
Barnes	4.88	7.81	4.94

*CTrigger analysis includes the three steps of setting up unserializable interleaving space. The profiling and controlled testing are conducted for every testing input in a systematic way with **NO** manual effort and **NO** knowledge of the contained bug or used inputs.*

within one full day. Actually, even after one week, the bug was still not exposed with stress testing (Note that this bug did appear during production runs and bothered the Apache server users. That is why it was reported in Apache's bugzilla database and was later fixed by developers.) All these results indicate that CTrigger can greatly reduce the testing time and make atomicity-violation bug detection and diagnosis more efficient.

Not surprisingly, Pure-Pin is similarly as ineffective as stress testing. Actually, since the Pin framework slows down each testing run, it takes a longer time than stress testing to expose the tested atomicity-violation bugs.

Synch-based testing perturbs the execution at synchronization points. It can help expose the PBZIP2 bug because this bug is caused by an unserializable access to a lock variable. The main thread could delete the lock variable too early, and the program crashes when a worker thread tries to acquire an already-deleted lock. However, it cannot help expose the other seven bugs. These seven bugs, like most real-world atomicity-violation bugs, were introduced when programmers did *not* put synchronization operations (locks or barriers or others) at places that need synchronization. As a result, without a synchronization point around the buggy code, Synch-based testing slows down each testing run without improving the chance of exposing these bugs.

As regards Race-based testing, the eight tested bugs can be divided into three categories. The first category includes Apache#2, MySQL, and PBZIP2. These bugs are successfully caught by Valgrind as race suspects. Leveraging the race detection results, Race-based testing can expose these bugs in similar amount of time with CTrigger. It is still slower than CTrigger for Apache#2 because the rareness-based ranking mechanism enables CTrigger to focus on buggy interleavings earlier than Race-based testing. The second category includes Apache#1 and the three SPLASH2 bugs. Since Valgrind fails to detect these bugs, Race-based testing cannot help expose them. This indicates that the bug-exposing capability of Race-based testing greatly relies on the underlying race detector's coverage. The last category is the Mozilla bug. Interestingly, it is reported by Valgrind as race suspects. However, the race pair reported is only a part of the p, r, c-accesses and the other pairs among them are protected by the same lock. Therefore, race-based testing based only on the race pair is insufficient to expose the atomicity violation.

8.1.2 CTrigger Bug-Exposing Time Breakdown

Table 5 shows the time spent in every step of CTrigger for exposing the above bugs. CTrigger trace collection and analysis take about 1 to 60 seconds. The profiling time mainly depends on how fast the set of unserializable interleavings exercised by stress testing becomes stable, and the analysis time is affected by the execution's memory footprint size.

CTrigger needs less than 5 seconds of controlled testing to expose most of the tested atomicity-violation bugs. Such efficiency is the combined effects of CTrigger infeasible interleaving pruning, ranking, and execution control strategies. In almost all cases, the bug-triggering interleavings are ranked very high in the low-probability interleaving list (refer to Section 8.5 for detailed ranking results). As a result, the bugs are exposed very quickly in few seconds of controlled testing. However, in Apache#1 and Mozilla, the bug-triggering interleavings are ranked relatively low and thus take longer testing time. In both cases, multiple benign atomicity violations are exercised and validated to be benign before the bugs get exposed.

8.2 Unserializable Interleaving Coverage

CTrigger effectively exposes atomicity-violation bugs through systematically exploring low-probability unserializable interleavings and improving the coverage within the unserializable interleaving space.

Table 6 shows how CTrigger improves the total unserializable interleaving space coverage. As we can see, Stress can cover around 60 percent of unserializable interleaving space. CTrigger can improve this number to larger than 90 percent. By exercising those hard-to-cover unserializable interleaving spaces, buggy unserializable interleavings are exposed. Of course, there are still some

TABLE 6
UI Space Coverage Increment

App.	# ALL feasible* UIs	Stress		CTrigger		
		# covered UIs	Coverage	# covered UIs	Coverage (%)	# Exposed Buggy UIs
Apache#1,#2	65	52	80%	65	100%	7
MySQL	25	24	96%	25	100%	1
Mozilla	31	11	35%	30	97%	2
PBZIP2	18	10	56%	15	83%	4
FFT	21	13	62%	21	100%	5
LU	7	4	57%	7	100%	3
Barnes	100	93	93%	100	100%	4

**: # of All feasible UIs Is obtained through automatic analysis PLUS manual code inspection.*

TABLE 7
The Time (Unit: Second) Spent to Reproduce an Exposed Bug

BugId.	Stress	Pure-Pin	Sync-based	Race-based	**CTrigger**	Speedup* (X)
Apache#1	–	–	–	–	76.2	**
Apache#2	NO	–	11664	0.70	1.3	> 66461.5
MySQL	348.0	5239.7	10054	0.90	0.90	386.7
Mozilla	–	–	–	5.44	4.39	**
PBZIP2	–	–	0.43	0.52	0.44	**
FFT	1658	5633	–	–	0.18	9211
LU	562.3	NO	–	–	0.18	3124
Barnes	165.4	–	–	–	0.45	367.6

*NO: The bug was not reproduced within one day. *: Speedup is calculated based on stress testing. -: We do not measure reproducing time when the bug cannot be exposed even once, as shown in Table 4. **: We cannot compute speedup as the stress testing never exposes the corresponding bug even once.*

unserializable interleavings that CTrigger failed to cover. These unserializable interleavings usually involve subtle customized synchronization and need more sophisticated perturbation to get exercised. For example, one unserializable interleaving in PBZIP2 requires well designed artificial delay in four distinct places to get exercised.

Fig. 10 uses Apache, PBZIP2, and BARNES as examples to show the detailed processes of CTrigger's UI-space exploration. As we can see, CTrigger tries to exercise the testing targets provided by CTrigger analysis one by one, gradually explores new unserializable interleavings that cannot be effectively touched by stress testing, and exposes atomicity-violation bugs.

These additionally covered interleavings include both bug-triggering ones and non-bug-related ones, as denoted in Fig. 10. Covering bug-triggering ones helps CTrigger to expose concurrency bugs; covering non-bug-related ones validates the correctness of these low-probability interleavings. In contrast, the number of interleavings explored in stress testing quickly saturates.

We can also see from the figure that CTrigger does not gain new coverage for every testing run. The main reason is that some infeasible interleaving skips CTrigger's interleaving feasibility analysis due to custom synchronization in the program. Fortunately, most infeasible interleavings have already been pruned.

8.3 Reproducing a Previously Exposed Bug

As shown in Table 7, CTrigger can efficiently reproduce all tested atomicity-violation bugs, mostly within 5 seconds. This high bug reproducibility provided by CTrigger can greatly help programmers' bug diagnosis. CTrigger achieves the high reproducibility by recording and replaying its execution control. After an atomicity-violation bug is exposed, CTrigger immediately knows which unserializable interleaving causes the manifestation of this bug. By repeating the same execution control and enforcing the same unserializable interleaving, CTrigger can easily repeat the bug.

Race and Synch-based testing also record and repeat the perturbation they inject during the bug-exposing runs. However, the perturbation record-and-replay scheme helps the bug reproducing only when the original bug exposing is *directly* caused by the perturbation (e.g., Race-based testing for the Apache#2, MySQL, and PBZIP2 race bugs), instead of by random effects. If the perturbation is not the root cause of the bug exposing, repeating the perturbation cannot help bug reproducing. For example, it still takes hours for Sync-based testing to reproduce Apache#2 and MySQL bugs.

Finally, as we can see in the table, for stress testing and Pure-Pin, reproducing a bug is always as difficult as exposing it at the first time because neither mechanism records any interleaving information when a bug is exposed.

TABLE 8
Effectiveness of Infeasible Interleaving Pruning

BugId.	# of Mem-Acc Instructions	# of Potential UI	# of CTrigger feasible UIs	Pruning (%)
Apache#1,#2	2551	297	157	47.1
MySQL	2257	113	25	77.9
Mozilla	2376	76	48	36.8
PBZIP2	149	93	25	73.1
FFT	311	205	21	89.8
LU	377	177	7	96.0
Barnes	716	470	143	69.6

MySQL uses a different input than that in Section 4. The pruning percentage is based on the number of potential UIs.

8.4 CTrigger Infeasible Interleaving Pruning

8.4.1 How Many Infeasible Interleavings Are Pruned?

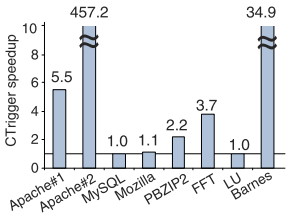
Identifying infeasible interleavings is critical for CTrigger to set a reachable testing goal. Table 8 shows that CTrigger feasibility analysis is very effective: 37-96 percent of the potential unserializable interleavings are successfully identified as infeasible. Here, we use the number of static instructions that touch shared memory locations as the number of potential UIs. In order to examine the stability of the feasibility analysis results, we execute each SPLASH2 application for 20 times. The sets of feasible interleavings generated from each of these 20 runs are exactly the same.

8.4.2 How Many Infeasible Interleavings Are Not Detected by Ctrigger Analysis?

As we can see in Table 9, CTrigger has effectively pruned most infeasible unserializable interleavings. Some infeasible interleavings were not detected by CTrigger analysis because they are prohibited by customized synchronization operations that are not considered by CTrigger, such as while-loop and producer-consumer queues. For example, in BARNES, among the 43 infeasible interleavings that went undetected by CTrigger analysis, 5 are caused by while-loop and 38 are caused by producer-consumer queue synchronization.

TABLE 9
How Many Infeasible UIs Skip the CTrigger Analysis

BugId.	CTrigger Analysis # Feasible UI	Manual Inspection # of Feasible UI
Apache#1,#2	157	65
MySQL	25	25
Mozilla	48	31
PBZIP2	25	18
FFT	21	21
LU	7	7
Barnes	143	100



(a) Speedups of CTrigger over the alternative ranking mechanism

Apache#1	7th out of 157
Apache#2	1st out of 157
MySQL	1st out of 25
Mozilla	14th out of 48
PBZIP2	3rd out of 25
FFT	2nd out of 21
LU	1st out of 7
Barnes	5th out of 143

(b) CTrigger rank of a bug-triggering interleaving in all feasible interleavings

Fig. 11. Efficient CTrigger ranking.

8.5 CTrigger Low-Probability Interleaving Ranking

The effectiveness of ranking can be seen from the unserializable interleaving space exploration curve (Fig. 10). As we can see, in most cases, the buggy interleavings are exercised earlier than benign interleavings and infeasible interleavings.

We also evaluate how the CTrigger ranking mechanism helps improve the efficiency of exposing hidden atomicity-violation bugs. For comparison, we applied an alternative scheme to decide the order of controlled testing: first come first serve. Specifically, we rank the unserializable interleavings based on their occurrence order, rather than estimated occurrence probability, during the execution. Using this ranking, we similarly apply the controlled testing and measure how long it takes to expose the tested atomicity-violation bugs.

As shown in Fig. 11a, CTrigger speeds up the alternative ranking method by up to 457.2 times in terms of the interleaving exploration time to expose the tested bugs. This shows that CTrigger's ranking method is effective: The bug-triggering interleavings are ranked high, as shown in Fig. 11b, using its local-gap-based probability estimation.

9 Related Work

9.1 Concurrent Program Testing

There are many previous works on testing concurrent programs. These works have laid out a solid background for interleaving testing. However, most of them have exponential or polynomial complexity and are impractical for large C/C++ software in real world.

Data-flow testing [11], [47] aims to exercise all possible data-flow relationships, such as define-use pairs, in a concurrent program. The strength of data-flow testing is that many bugs are truly associated with abnormal data-flow patterns, no matter whether in sequential software or multithreaded software. The limitation of data-flow testing is that it is difficult, if not completely impossible, to figure out the data-flow testing space of a multithreaded application. The testing space is also huge, at least polynomial in the program size. Due to complexity concerns, these works have not guided practical testing to expose concurrency bugs in large programs.

Reachability testing [12], [14], [16], [38] is an approach that combines nondeterministic and deterministic testing. It starts from one run of the program (i.e., one interleaving) and discovers all races in this run dynamically. It then keeps exploring new interleavings that exercise new combinations of race outcomes and keeps discovering new races until all possible partial orders of synchronization events are exhausted. The strength of reachability testing is that it does not rely on any static model of the program. Instead, it easily derives all testing targets on the fly by flipping race outcomes. The limitation is its exponential testing complexity. This is especially a concern in practice, when the testing budget is limited and large applications are involved.

Simplifying interleaving testing and making it practical for real-world large applications have become urgent recently. Actually, the experience of the CHESS project [27] by Microsoft Research indicates that an interleaving testing is likely to be impractical for many large applications in the real world if its complexity is polynomial, not to mention exponential, in the number of shared memory accesses in software.

Several recent works are designed on providing practical interleaving testing based on different heuristics [4], [17], [18], [26], [27], [37]. Most of these works were already discussed in Section 1.2. CHESS [27], by default, allows context switches to occur only at synchronization operations in order to make its testing affordable for most real-world applications. T-way reachability testing [17] smartly simplifies the basic reachability testing. It explores possible combinations of race outcomes for every groups of t races, instead of all combinations for all races (t is a small constant and can be as small as 1).

The main strength of these works is that they target simple testing schemes that are feasible to conduct in practice, a common goal shared with CTrigger. CTrigger can complement these works well because CTrigger simplifies the testing space from a new perspective. Specifically, CTrigger is guided by the characteristics of real-world atomicity-violation bugs and is especially good at exposing atomicity-violation bugs. Since we have already compared CTrigger with most of these works in detail in Section 1.2, here we use a simple example to demonstrate the difference between CTrigger and t-way reachability testing [17] as follows:

Suppose a toy program P has two threads. Thread 1 writes a shared variable v twice and thread 2 reads v once without synchronization. The interleaving space of P contains three interleavings and only one, denoted by I, can potentially trigger an atomicity-violation bug (i.e., thread 2 reading v in between the two writes from thread 1). The testing space of CTrigger only includes one interleaving that is exactly I. CTrigger testing can focus on exposing atomicity-violation bugs under limited testing budget. The testing space of t-way reachability testing includes more than one interleaving because any single interleaving alone cannot cover all possible outcomes of all the races in P. Of course, reachability testing could have better chances to expose non-atomicity-violation concurrency bugs. CTrigger and t-way reachability testing complement each other, as they use different perspectives to simplify the testing space.

Comparing with previous works, the main contribution of CTrigger lies in two aspects. First, CTrigger has a linear-sized testing space and is practical for real-world large

applications. Second, CTrigger design is guided by the characteristics of atomicity-violation bugs, an important type of concurrency bugs. CTrigger's linear-sized testing space is not randomly decided. It is composed of carefully selected interleavings that are most likely to expose hard-to-manifest atomicity-violation bugs.

CTrigger also has its limitations. Since it is designed for atomicity-violation bugs, CTrigger may not expose other types of concurrency bugs that some previous works can. Of course, we believe atomicity-violation bugs are worth our attention because they contribute to about 70 percent of real-world nondeadlock concurrency bugs according to previous empirical studies [21].

Overall, CTrigger is a practical testing framework and can well complement previous techniques in testing multithreaded software.

9.2 Concurrency Bug Detection

Much research has been conducted on detecting various types of concurrency bugs, especially atomicity-violation bugs and data races.

As discussed in Section 2, atomicity-violation bugs occur when concurrent execution violates the atomicity of certain code region. Many tools have been built to detect these bugs [7], [8], [10], [22], [41], [42], [46].

Traditional data-race detection [31], [33], [48] focuses on conflicting accesses to a single variable. Recent works discovered that concurrency bugs could also occur when accesses to multiple correlated variables are not synchronized. These bugs are called *high-level data races* [1] or *multivariable concurrency bugs* [20].

Differently from all these bug detection works, CTrigger focuses on testing. It systematically exercises representative interleavings in a concurrent program's huge interleaving space, and effectively makes single-variable atomicity-violation bugs manifest during in-house testing. Just like traditional input testing conducted in all software companies, CTrigger testing has no false positive and provides reliable bug reexposing for diagnosis. In addition, by forcing hidden bugs to manifest, CTrigger well complements many bug detectors [8], [10], [22], [46] that only catch bugs when they manifest.

Some previous works, such as [41], [42], can analyze the execution trace and report potential atomicity violations that have not occurred yet. CTrigger is different from these works mainly at two aspects. First, CTrigger targets at proactive exposing bugs during in-house testing. To achieve this goal, CTrigger not only detects potential bugs, but also perturbs the execution to make bug-triggering interleavings happen so that false positives can be automatically pruned and true bugs can be reliably reproduced for diagnosis. Second, CTrigger is carefully designed to be scalable for large C/C++ multithreaded applications. For example, CTrigger designs a testing space that is linear in the number of static instructions in the program. CTrigger also uses ranking to improve the testing efficiency. In comparison, the block-based atomicity-violation algorithm [41], [42] is polynomial in the number of dynamic memory accesses in the execution. It is only evaluated on small applications with thousands of lines of code and is difficult to apply to large applications.

9.3 Other Related Works

Deterministic replay is a common technique that can help developers to reproduce a software bug that has already occurred [5], [13], [15], [28], [40]. CTrigger is different from these replay tools because it does not wait for a bug to manifest, which may never happen during in-house development. Instead, it perturbs the execution to expose bugs that have never manifested before.

Some recent works [23] target surviving atomicity-violation bugs that have escaped the in-house testing. Such production-run surviving techniques and development-site exposing techniques like CTrigger can complement each other well.

10 Conclusions and Future Work

This paper proposes a new method, called CTrigger, to expose difficult-to-detect and tough-to-diagnose atomicity-violation bugs that are often hidden in low-probability unserializable interleavings. CTrigger achieves this by selecting representative interleavings, pruning infeasible ones, identifying low-probability ones, and controlling execution to force them to occur.

Our experiments with seven real-world applications show that CTrigger is effective: It achieves two to four orders of magnitude speedup in bug exposing and two to five orders of magnitude speedup in bug reproducing (for diagnosis) over stress testing. For some server application bugs that need several days of stress testing to manifest, CTrigger can expose them within 4 minutes. With the significantly improved efficiency and reproducibility of bug exposing, CTrigger can help bug detectors locate bugs more quickly and accurately and save developers a lot of efforts in bug diagnosis.

Our work is only the beginning on addressing the important problem of exposing atomicity-violation bugs. It can be improved by more accurate infeasible interleaving pruning, better selection of rare interleavings, better planning in exercising a group of interleavings, and extension to expose more complicated atomicity-violation bugs (e.g., multivariable involved bugs). Future work can also combine it with test input generation and other interleaving testing mechanisms.

Acknowledgments

The authors thank the anonymous reviewers whose comments and suggestions have greatly improved our paper. This work is supported by US National Science Foundation (NSF) grants CNS-0720743, CCF-0325603, CNS-0615372, CNS-0347854 (career award), CCF-1018180 grant, a NetApp Gift grant, and a Claire Boothe Luce faculty fellowship. An earlier version of this paper [32] appeared in the *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems*.

References

[1] C. Artho, K. Havelund, and A. Bierre, "High-Level Data Races," *Proc. First Int'l Workshop Verification and Validation of Enterprise Information Systems*, 2003.

[2] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," *Proc. ACM SIGMETRICS Joint Int'l Conf. Measurement and Modeling of Computer Systems,* June 1998.

[3] B. Beizer, *Software Testing Techniques,* second ed. Van Nostrand Reinhold, 1990.

[4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of Synchronization Coverage," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* 2005.

[5] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen, "Execution Replay of Multiprocessor Virtual Machines," *Proc. ACM SIGPLAN/SIGOPS Int'l Conf. Virtual Execution Environment,* 2008.

[6] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multi-Threaded Java Program Test Generation," *IBM Systems J.,* vol. 41, pp. 111-125, 2002.

[7] C. Flanagan and S.N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 2004.

[8] C. Flanagan, S.N. Freund, and J. Yi, "Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2008.

[9] P.G. Frankl and E.J. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1483-1498, Oct. 1988.

[10] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic Detection of Atomic-Set-Serializability Violations," *Proc. Int'l Conf. Software Eng.,* 2008.

[11] M.J. Harrold and B.A. Malloy, "Data Flow Testing of Parallelized Code," *Proc. Int'l Conf. Software Maintenance,* 1992.

[12] G.-H. Hwang, K.C. Tai, and T.L. Huang, "Reachability Testing: An Approach to Testing Concurrent Software," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 5, pp. 493-510, 1995.

[13] S.T. King, G.W. Dunlap, and P.M. Chen, "Operating Systems with Time-Traveling Virtual Machines," *Proc. USENIX Ann. Technical Conf.,* 2005.

[14] P.V. Koppol and K.-C. Tai, "An Incremental Approach to Structural Testing of Concurrent Software," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* 1996.

[15] T.J. LeBlanc and J.M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Trans. Computers,* vol. 36, no. 4, pp. 471-482, Apr. 1987.

[16] Y. Lei and R.H. Carver, "Reachability Testing of Concurrent Programs," *IEEE Trans. Software Eng.,* vol. 32, no. 6, pp. 382-403, June 2006.

[17] Y. Lei, R.H. Carver, R. Kacker, and D. Kung, "A Combinatorial Testing Strategy for Concurrent Programs," *Software Testing, Verification and Reliability,* vol. 17, no. 4, pp. 207-225, 2007.

[18] Y. Lei and E. Wong, "A Novel Framework for Non-Deterministic Testing of Message-Passing Programs," *Proc. Ninth IEEE Int'l Symp. High-Assurance Systems Eng.,* pp. 66-75, 2005.

[19] S. Lu, W. Jiang, and Y. Zhou, "A Study of Interleaving Coverage Criteria," *Proc. Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.,* 2007.

[20] 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," *Proc. ACM SIGOPS Symp. Operating Systems Principles,* 2007.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes—A Comprehensive Study of Real World Concurrency Bug Characteristics," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2008.

[22] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting Atomicity Violations via Access Interleaving Invariants," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2006.

[23] B. Lucia, J. Devietti, K. Strauss, and L. Ceze, "Atom-Aid: Detecting and Surviving Atomicity Violations," *Proc. Ann. Int'l Symp. Computer Architecture,* 2008.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2005.

[25] Y. Malaiya, N. Li, R. Karcich, and B. Skbbe, "The Relationship between Test Coverage and Reliability," *Proc. Int'l Symp. Software Reliability Eng.,* 1994.

[26] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2007.

[27] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler, "Finding and Reproducing Heisenbugs in Concurrent Programs," *Proc. Conf. Operating Systems Design and Implementation,* 2008.

[28] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder, "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems,* 2006.

[29] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically Classifying Benign and Harmful Data Racesallusing Replay Analysis," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2007.

[30] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2007.

[31] R.H.B. Netzer and B.P. Miller, "Improving the Accuracy of Data Race Detection," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* 1991.

[32] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* 2009.

[33] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs," *ACM Trans. Computer Systems,* vol. 15, pp. 391-411, 1997.

[34] SecurityFocus "Software Bug Contributed to Blackout," http://www.securityfocus.com/news/8016, 2011.

[35] K. Sen, "Race Directed Random Testing of Concurrent Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2008.

[36] K. Sen and G. Agha, "Automated Systematic Testing of Open Distributed Programs," *Proc. Fundamental Approaches to Software Eng.,* 2006.

[37] S.D. Stoller, "Testing Concurrent Java Programs Using Randomized Scheduling," *Electronic Notes in Theoretical Computer Science,* vol. 70, no. 4, pp. 142-157, July 2002.

[38] R.N. Taylor, D.L. Levine, and C.D. Kelly, "Structural Testing of Concurrent Programs," *IEEE Trans. Software Eng.,* vol. 18, no. 3, pp. 206-215, Mar. 1992.

[39] M. Vaziri, F. Tip, and J. Dolby, "Associating Synchronization Constraints with Data in an Object-Oriented Language," *Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages,* 2006.

[40] VMware, "(Appendix c) Using the Integrated Virtual Debugger for Visual Studio," http://www.vmware.com/pdf/ws65_manual.pdf, 2011.

[41] L. Wang and S.D. Stoller, "Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming,* 2006.

[42] L. Wang and S.D. Stoller, "Runtime Analysis of Atomicity for Multithreaded Programs," *IEEE Trans. Software Eng.,* vol. 32, no. 2, pp. 93-110, Feb. 2006.

[43] S.N. Weiss, "A Formal Framework for the Study of Concurrent Program Testing," *Proc. Second Workshop Software Testing, Verification, and Analysis,* 1988.

[44] E.J. Weyuker, "More Experience with Data Flow Testing," *IEEE Trans. Software Eng.,* vol. 19, no. 9, pp. 912-919, Sept. 1993.

[45] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. Ann. Int'l Symp. Computer Architecture,* 1995.

[46] M. Xu, R. Bodík, and M.D. Hill, "A Serializability Violation Detector for Shared-Memory Server Programs," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 2005.

[47] C.-S.D. Yang, A.L. Souter, and L.L. Pollock, "All-Du-Path Coverage for Parallel Programs," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* 1998.

[48] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking," *Proc. ACM Symp. Operating Systems Principles*, 2005.

Shan Lu received the PhD degree from the University of Illinois at Urbana-Champaign. She is an assistant professor in the Computer Science Department at the University of Wisconsin–Madison. Her main research interests are software reliability and concurrent software systems.

Soyeon Park received the PhD degree from the Korea Advanced Institute of Science and Technology. She is a project scientist in the Department of Computer Science and Engineering at the University of California, San Diego. Her research interests include software reliability, operating system, and computer architecture.

Yuanyuan Zhou received the PhD degree in computer science from Princeton University. She is a Qualcomm Chair Professor in the Department of Computer Science and Engineering at the University of California, San Diego. Her research interests include software reliability, operating systems, and storage systems. She is a member of the IEEE.