CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places

Soyeon Park Shan Lu Yuanyuan Zhou

Department of Computer Science,
University of Illinois at Urbana Champaign, Urbana, IL 61801
{soyeon,shanlu,yyzhou}@illinois.edu

Abstract

Multicore hardware is making concurrent programs pervasive. Unfortunately, concurrent programs are prone to bugs. Among different types of concurrency bugs, atomicity violation bugs are common and important. Existing techniques to detect atomicity violation bugs suffer from one limitation: requiring bugs to manifest during monitored runs, which is an open problem in concurrent program testing.

This paper makes two contributions. First, it studies the interleaving characteristics of the common practice in concurrent program testing (i.e., running a program over and over) to understand why atomicity violation bugs are hard to expose. Second, it proposes CTrigger to effectively and efficiently expose atomicity violation bugs in large programs. CTrigger focuses on a special type of interleavings (i.e., unserializable interleavings) that are inherently correlated to atomicity violation bugs, and uses trace analysis to systematically identify (likely) feasible unserializable interleavings with low occurrence-probability. CTrigger then uses minimum execution perturbation to exercise low-probability interleavings and expose difficult-to-catch atomicity violation.

We evaluate CTrigger with real-world atomicity violation bugs from four sever/desktop applications (Apache, MySQL, Mozilla, and PBZIP2) and three SPLASH2 applications on 8-core machines. CTrigger efficiently exposes the tested 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 7 full days of stress testing. In addition, without deterministic replay support, once a bug is exposed, CTrigger can help programmers reliably reproduce it for diagnosis. Our tested bugs are reproduced by CTrigger mostly within 5 seconds, 300 to over 60000 times faster than stress testing.

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

General Terms Languages, Reliability

Keywords Software testing, concurrency bug

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 hard to detect and diagnose because of their notorious non-deterministic characteristic. Many concurrency bugs skip the in-house checking, escape into production runs and cause catastrophic disasters in real world (e.g., the Northeastern Electricity Blackout Incident [21]).

Among different types of concurrency bugs, atomicity violation bugs are one of the most common and important [13, 12, 14, 25, 5]. Atomicity violation bugs (a real-world example is shown in Figure 1) widely exist because many programmers are used to sequential thinking and frequently assume code regions to be atomic without appropriate enforcement. Our recent concurrency bug characteristic study [12] shows that about 70% of non-deadlock concurrency bugs in the studied large server and desktop applications are caused by atomicity violations. In addition, atomicity violation bugs will remain even with advanced synchronization primitives such as transactional memory, because programmers might mistakenly separate a group of indivisible operations into different transactions [13, 14]. Therefore, techniques to help eliminate atomicity violation bugs are highly desired.

Recently, much effort has been made to help detect atomicity violation bugs [27, 13, 5, 7, 25]. Almost all of these work would significantly benefit from an effective way to expose atomicity violations during monitored runs (testing runs). For example, dynamic atomicity violation checkers like AVIO [13], SVD [27], and other approaches [7] require atomicity violations to manifest during monitored runs in order to catch them. Although static approaches [6] do not

ASPLOS'09 March 7–11, 2009, Washington, DC, USA.

have such requirements, their power is limited due to the complexity of analyzing concurrency and pointer aliasing, especially for C/C++ programs. As a result, static tools can introduce many false positives. An effective way to examine bug suspects via testing runs can also help static tools to separate false positives from true bugs [22].

Bug-exposing techniques have been studied for a long time and many good techniques have been widely adopted to test *sequential* programs [2]. In general, a good bug-exposing technique needs to have three properties:

- *Effectiveness*: how many hidden bugs can be exposed.
- *Efficiency*: how fast hidden bugs can be exposed. Although the performance issue of testing is not as critical as that of production runs, the bug exposing process cannot take months or years because programmers have constant pressure to release software.
- *Reproducibility*: if a hidden bug is exposed, how likely this bug can be reproduced for diagnosis. If the bug takes another 20 hours to reproduce, it might be very pain-taking for programmers to examine the problem.

Unlike sequential bugs, a concurrency bug usually requires 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. Much work has conducted in the past to generate comprehensive sets of inputs to cover code segments and specification space [2]. The majority of these work are still applicable to concurrent programs, although some extensions specific to concurrent programs are needed to further increase the code coverage [23].

The second condition, unique to concurrency bugs, is a *bug-triggering interleaving*. Without this condition, a bug-triggering input alone may not expose the hidden concurrency bug. Figure 1 shows a real world bug example from the Apache HTTPd Server, a widely-used open-source web server. In this example, programmers forget to protect the pair of accesses to `buf_index`, namely $\{S1, S2\}$, into the same atomic region using locks or transactions and introduce an atomicity violation bug. Unfortunately, this bug is hard to expose during testing because it manifests *only* when $S3$ is executed between $S1$ and $S2$. The probability for this particular interleaving to happen is very small. Actually, when we ran Apache with a *bug-triggering input* (an input that can potentially trigger the bug) on an 8-core machine, it took 22 hours for this bug to manifest.

In comparison to the first condition, the second condition is significantly understudied. Therefore, similar to recent concurrency testing efforts [4, 16, 17, 22], *this paper focuses on the bug-triggering interleaving issue* and relies on prior work to cover the first condition.

1.2 State of the Art

The common practice to expose concurrency bugs is to run a program with each input test case for a long time (for servers) or for many times (for other types of applications). We refer to this as *stress testing*. Intuitively it makes some sense, since the non-deterministic nature of concurrent programs will help exercise different interleavings in different runs. Unfortunately, practice has shown that stress testing is neither efficient nor reproducible [16]. The first part of this paper will dig deeper into the reason for the deficiency of stress testing.

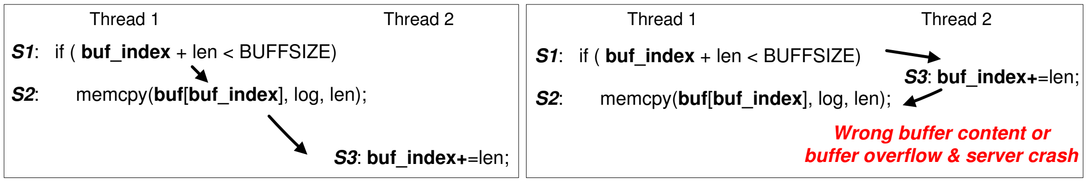
Recently, several inspiring works [3, 4, 16, 17, 22] were proposed to improve stress testing. All of these works target at selecting certain interleavings from the exponential size interleaving space for practical testing to focus on.

ConTest [3] injects artificial delays at synchronization points (e.g., lock acquisition & release) in order to intensify the contention for synchronization resources. This would help expose deadlocks, but not data races or atomicity violation bugs that are usually caused by programmers forgetting to use synchronizations.

CHESS [16, 17] cleverly reduces the interleaving testing space by bounding the number of preempting context switches to small numbers (e.g., 1 – 4). However, even with a small number of context switches allowed, CHESS' testing space still increases polynomially with the program execution length. As a result, CHESS has to make a hard trade-off between coverage and testing time. For example, CHESS often limits context switches only at synchronization points in order to test big concurrent programs in practice. Such a constraint will make the method less effective for exposing atomicity violation and data race bugs, just like that in ConTest as discussed above. Our ideas presented in this paper well complement CHESS by systematically picking out interleavings that have low occurrence probabilities and high association with atomicity violation bugs.

Based on the same motivation, RaceFuzzer [22] focuses on potential data races reported by race detectors. It attempts to force all the reported race interleavings during testing in order to separate false positives from true race bugs. While it is definitely useful to help users automatically filter out false positives in race bug detection, its bug exposing capability significantly relies on the underlying data race detectors: if the detector does not have a good coverage, RaceFuzzer would miss many bugs. Unfortunately, due to the inherent complexity of concurrent programs, there are still few race bug detectors that can achieve high coverage, especially for C/C++ programs and for atomicity violation bugs.

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 address this limitation and allow each test run to use multiple processors, just like that in stress testing.



```
S1:  if ( buf_index + len < BUFFSIZE)
S2:      memcpy(buf[buf_index], log, len);
                                            S3: buf_index+=len;
```

```
S1:  if ( buf_index + len < BUFFSIZE)
                                            S3: buf_index+=len;
S2:      memcpy(buf[buf_index], log, len);
```

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

Figure 1. An example simplified from an Apache atomicity violation bug (It manifests only when S3 is executed between S1 and S2.)

1.3 Contributions of this paper

This paper studies the interleaving characteristics of stress testing and proposes a method called CTrigger to efficiently expose atomicity violation bugs in large programs.

First, to reduce the interleaving space in exploration, we propose to focus on a special type of interleavings called unserializable interleavings that are inherently correlated to atomicity violation bugs [27, 13]. An *unserializable interleaving* is an interleaving that is not equivalent to any sequential execution of the involved operations. As atomicity is equivalent to serializability in the context of concurrency bugs, focusing on unserializable interleavings can provide a good coverage of exposing atomicity violation bugs and allow us to substantially reduce the interleaving testing space.

Second, using three large server programs, three SPLASH2 programs, and one utility program, we examine why stress testing is insufficient in 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 be covered without external control and are also hard to reproduce for bug diagnosis. The primary factors that affect interleaving probabilities are synchronizations, memory access distances, etc.

Third, based on our above observations, we design a testing framework called CTrigger to effectively, efficiently and reproducibly expose atomicity violation bugs in concurrent programs. CTrigger achieves these goals by incorporating the following new ideas step by step as shown in Figure 2.

- Focusing on unserializable interleavings. From a few profiling runs, CTrigger identifies a large set of potential unserializable interleavings.
- Pruning infeasible interleavings. Some potential unserializable interleavings can never happen during execution due to synchronizations. 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% of potential unserializable interleavings in the seven tested applications are pruned.
- Ranking and identifying low-probability interleavings. As different interleavings have different probabilities to be exposed, we propose a simple metric to estimate interleaving probability and rank all unpruned 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 the simple stress testing mechanism. Our experimental results show that our ranking mechanism is effective. It ranks bug-triggering interleavings high, mostly within top 10%, and achieves speedup of bug exposing time by up to 457 times. Besides our work, the ranking metric may also be useful to other concurrency testing frameworks such as CHESS to improve testing efficiency.
- Minimum external control to force low-probability interleavings during testing on multicores. 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 multicores, and avoids slowing down execution periods that are unrelated to the target interleavings.

We evaluate CTrigger with real world bugs from four server/desktop open-source programs, MySQL, Apache, Mozilla, and PBZIP2, and three SPLASH2 benchmarks on 8-core machines. Among these applications, 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 10–1000 times faster than stress testing and previous methods (both synchronization-based or race-based techniques described in Section 1.2). For example, CTrigger takes 63 seconds and 235 seconds, respectively, to expose the two real world Apache server 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 10–1000 would be very beneficial. For example, 100 different 1-hour-long (under CTrigger) input test cases and 10 different configurations would take CTrigger 2 days to finish on

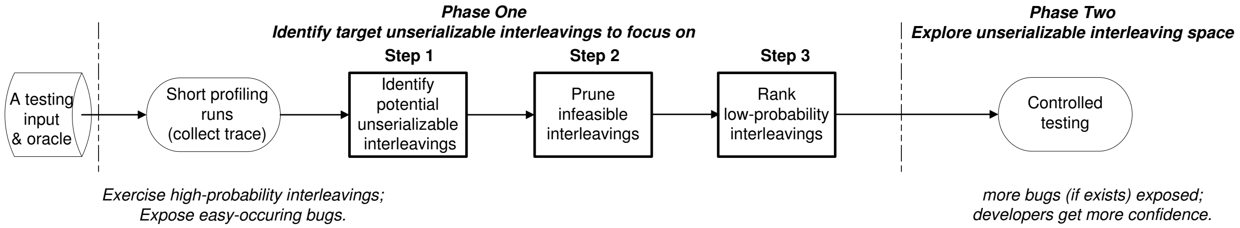

Phase One
Identify target unserializable interleavings to focus on
Phase Two
Explore unserializable interleaving space
A testing input & oracle
Short profiling runs (collect trace)
Step 1
Identify potential unserializable interleavings
Step 2
Prune infeasible interleavings
Step 3
Rank low-probability interleavings
Controlled testing
Exercise high-probability interleavings;
Expose easy-occuring bugs.
more bugs (if exists) exposed;
developers get more confidence.

Figure 2. CTrigger testing framework (The phases one and two are conducted for each test input.)

20 machines, whereas they will take stress testing 20-2000 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 re-expose the same bug for diagnosis without any deterministic replay support. For the tested bugs, CTrigger re-exposes them mostly within 5 seconds, 300 to more than 60000 times faster than stress testing.

2. Background: Atomicity Violation Bugs and Unserializable Interleavings

Atomicity, also called as **serializability**, is a property for the concurrent execution of several operations when their data manipulation effect is equivalent to that of a serial execution of them [13]. Programmers often assume some code regions to be atomic. Unfortunately, their implementation may not guarantee the atomicity. Consequently, the assumed atomicity can be broken when the code region is unserializably interleaved by accesses from another thread, which leads to an atomicity violation bug.

As discussed in details in some recent work [13, 25], the basic type of unserializable interleavings is composed of three memory accesses (shown in Figure 3). Two of them, referred to as p(receding)-access and c(urrent)-access, consecutively access a shared location from the same thread. The third one, referred to as r(emote)-access, accesses the same memory location in the middle of the previous two from a different thread. For example, the key part of the bug shown in Figure 1 is such a basic type of unserializable interleaving. The bug manifests when r-access S3 unserializably interleaves the p-access S1 and c-access S2.

Due to the inherent connection between atomicity violation bugs and unserializable interleavings, it is natural to focus on unserializable interleavings in order to expose atomicity violation bugs. Furthermore, for simplicity and efficiency, we can start with the basic type of unserializable interleavings described above. Specifically, for every shared memory access instruction C, we can try to exercise at least one unserializable interleaving associated with C, i.e., *interleaving-C*, short for an unserializable interleaving with instruction C as the current access. We accordingly define the exploration space as $\{$interleaving-$C|C$ is a shared-memory access instruction$\}$. Within this space, some unserializable interleavings may never happen due to synchronization. We will discuss how to prune out these *infeasible interleavings* in later sections.

The unserializable interleaving space defined above is linear to the static size of the program. It is much smaller than the entire interleaving space and is therefore practical to thoroughly explore. In the mean time, 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.

3. Why Stress Testing is Not Good: An Interleaving Characteristic Study

Stress testing (defined in Section 1.2) is the current dominant practice. To understand why it is ineffective at exposing atomicity violation bugs, we quantitatively study its characteristics from the perspective of unserializable interleaving space. The understanding will guide our design of CTrigger.

3.1 Methodology

We use four widely-used open-source server/desktop applications, Apache HTTPd, MySQL, Mozilla and PBZIP2, and three applications from the SPLASH2 [26] benchmark-suite. These applications cover different types of functionalities and synchronization models, as shown in table 1. The experiments use a dual quad-core (totally eight processors) Intel Xeon machine, and each application is configured to have eight worker threads.

To collect the interleaving information, we use Pin binary instrumentation tool [15] to monitor the execution. To make sure that our study can reflect the real non-perturbed execution environment, we carefully design our instrumentation to give minimum perturbation in a thread-balanced way.

App.	LOC	Description	Synch.	Workload
Apache	302K	Web server	lock	SURGE [1]
MySQL	1.9M	Database server	lock	MySQL-test*
Mozilla	3.4M	Web browser suite	lock	JavaScript test suite*
PBZIP2	2.0K	File compressor	lock & queue	a random file
FFT	1.0K	FFT transformation	barrier	default setting with 8 processors
LU	1.0K	Matrix factorization	barrier	
Barnes	3.0K	N-body problem	lock & queue	

Table 1. Applications and workloads (*:MySQL-test and JavaScript test suite are designed by the application developers.)

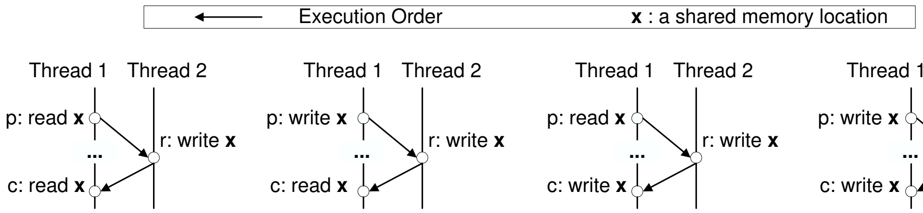


Figure 3. **Unserializable interleavings** (A static instruction C's unserializable interleaving is exercised iff at least one of its dynamic instances follows above pattern during execution.)

3.2 Observations

Our experimental results reveal the following observations:

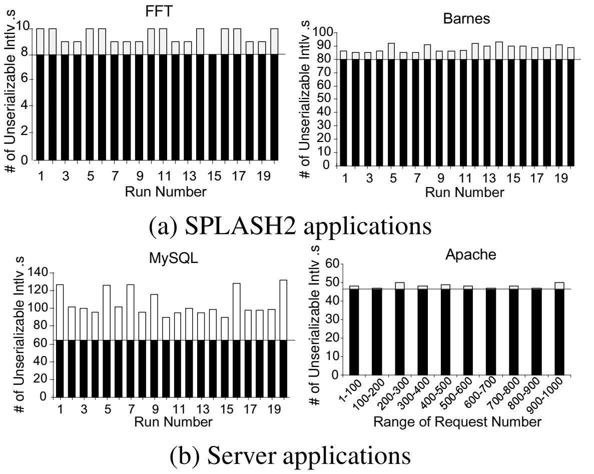
(1) *Is stress testing non-deterministic in a random way?* From the perspective of covering unserializable interleavings, the answer is *no*. As shown in Figure 4, the majority of unserializable interleavings exercised by different runs (or different iterations for server programs) are the same.

(2) *Can we rely on stress testing to cover the whole unserializable interleaving space?* The answer is *no*. As shown in Figure 5, stress testing hardly exercises 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 to be exercised?* Different interleavings have completely different occurrence probabilities. For example, Figure 4 shows that some interleavings are exercised in all stress testing runs, i.e., about 100% occurrence probability. On the contrary, some interleavings are never exercised during days of experiment, i.e., almost 0% probability. Further examination reveals the following major factors determining the probability: (i) program synchronizations, such as lock and barrier, make some interleavings always happen and some never happen; (ii) 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; (iii) 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.

3.3 Implications to exposing atomicity violation bugs

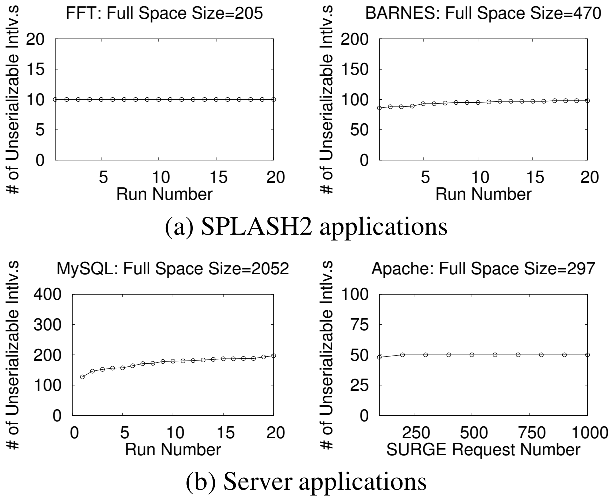
In summary, we can see that stress testing is not good at exposing atomicity violation bugs because it cannot effectively exercise the unserializable interleaving space. Without perturbation to the execution, stress testing repeatedly tests those high-probability unserializable interleavings. *Atomicity 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) [11, 12, 13, 27].



(a) SPLASH2 applications

(b) Server applications

Figure 4. **Similarity between runs:** each bar shows the number of unserializable interleavings covered in each run. The dark part shows the number of interleavings that are exercised by *all* runs, i.e., having 100% occurrence frequency. The remaining (those with less than 100% frequency) are shown on the white part.



(a) SPLASH2 applications

(b) Server applications

Figure 5. **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 4.1).

4. CTrigger Phase One: Identify Target Unserializable Interleavings to Focus on

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 Figure 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 through 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 Step 1, 2 and 3 in Figure 2). We will discuss the second phase in the next section.

Please note that we take similar assumptions with recent work on concurrency testing [16, 22, 4]. We assume that programmers have a test case suite and they go through CTrigger's phase one and two for each test input. We also assume that, for one input, the code statements executed at different runs are mostly, maybe not completely, the same.

4.1 Step 1: profiling and 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 be all feasible) unserializable interleavings.

In a program, not every memory access instruction has its corresponding unserializable interleaving. Since an unserializable interleaving is composed of three accesses, a p(receding)-access, a c(urrent)-access and an r(emote)-access (refer to Section 2), 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.

4.2 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 Figure 6). Additionally, CTrigger also prunes other types of infeasible interleavings such as those caused by memory recycling.

4.2.1 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 Figure 6(b).

In our implementation, CTrigger records all barrier and thread-create/join operations into the trace. In trace analysis, CTrigger uses vector timestamps to maintain and compare the order relationship between accesses. Note that vector timestamps used in CTrigger are similar but different from those used in conventional happens-before race detection algorithms [19]: CTrigger does **not** push ahead vector timestamps at lock/unlock operations, because lock/unlock does not force absolute orders.

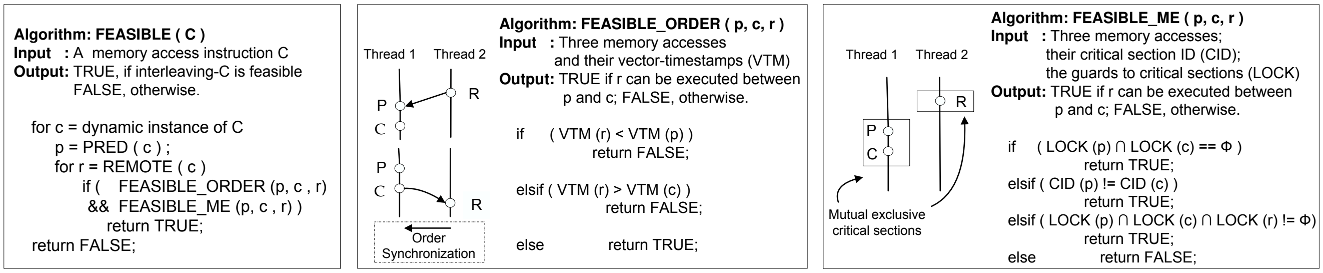
Infeasible interleavings caused by mutual exclusion Synchronization primitives like locks and transactions provide mutual exclusion in concurrent programs. Considering this type of synchronization, an r-access cannot interleave a p- and a c-access iff there exist two mutual exclusive critical regions 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 (Figure 6(c)).

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. Different from the lock-set race detection algorithm [20], the lock-sets maintained by CTrigger record *dynamic*, rather than static, lock instances that protect each access. In this way, CTrigger can tell whether two accesses are inside the same critical section.

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.

4.2.2 Discussions

CTrigger works well for real-world server programs written in C, as we will see in the experiments (Section 6). Most infeasible interleavings can be correctly identified. However, a small number of infeasible interleavings may be missed due to un-identified 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. Note that our current prototype can also be extended to consider other synchronization operations, e.g. *transactions*.



```
Algorithm: FEASIBLE ( C )
Input   : A  memory access instruction C
Output: TRUE, if interleaving-C is feasible
            FALSE, otherwise.

   for c = dynamic instance of C
        p = PRED ( c ) ;
        for r = REMOTE ( c )
              if (   FEASIBLE_ORDER (p, c , r)
               &&  FEASIBLE_ME (p, c , r) )
                    return TRUE;
   return FALSE;
```

(a) algorithm to distinguish feasible unserializable interleaving

```
Algorithm: FEASIBLE_ORDER ( p, c, r )
Input   : Three memory accesses
              and their vector-timestamps (VTM)
Output: TRUE if r can be executed between
              p and c; FALSE, otherwise.

   if      ( VTM (r) < VTM (p) )
                  return FALSE;

   elsif ( VTM (r) > VTM (c) )
                       return FALSE;

   else             return TRUE;
```

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

```
Algorithm: FEASIBLE_ME ( p, c, r )
Input   : Three memory accesses;
              their critical section ID (CID);
              the guards to critical sections (LOCK)
Output: TRUE if r can be executed between
              p and c; FALSE, otherwise.

   if     ( LOCK (p) ∩ LOCK (c) == Φ )
                   return TRUE;
   elsif ( CID (p) != CID (c) )
                   return TRUE;
   elsif ( LOCK (p) ∩ LOCK (c) ∩ LOCK (r) != Φ)
                   return TRUE;
   else            return FALSE;
```

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

Figure 6. CTrigger feasible interleaving analysis algorithm (PRED and REMOTE denote preceding access(es) and remote accesses(es), respectively. They are collected in step 1.)

4.3 Step 3: ranking low-probability interleavings

As discussed in Section 3.2, different interleavings have different occurrence probabilities during stress testing. Some interleavings rarely occur but have high likelihood to hide atomicity violation bugs, especially those bugs that are hard to reproduce for diagnosis. Therefore, it is 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.

4.3.1 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-access) are, and how far away a remote access (r-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 (Figure 7).

- **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.

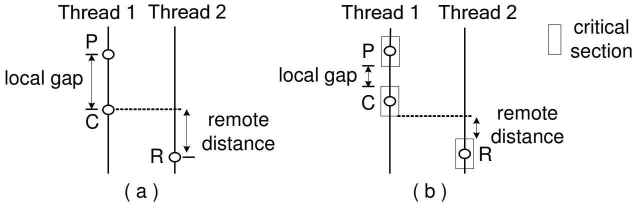


Figure 7. Local gap and remote distance

There is a big difference between the two metrics: 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.

4.3.2 How to compute the metrics?

The main idea of 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:

(1) 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 few instructions will be affected.

(2) 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* (Figure 7(b)), 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.

(3) 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.

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

5. 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 the phase one, starting from the ones with the lowest (estimated) occurrence-probabilities. As CTrigger records the execution control that it makes during testing, once CTrigger succeeds to expose an atomicity violation bug, it can reliably reproduce the bug by retrying the same control, which helps diagnosis.

Execution control for one interleaving Unlike previous work such as CHESS [16] and RaceFuzzer [22] that control thread schedule and execute only one thread at a time, CTrigger controls execution by suspending a thread's execution at appropriate places to increase the occurrence probability of the targeting unserializable interleaving. The period of suspension is carefully controlled to avoid significant performance degradation.

Specifically, for an unserializable interleaving, CTrigger suspends corresponding threads before its c-access C or r-access R whenever necessary during the execution (Figure 8 (a)). This can help increase the *local gap* and decrease the *remote distance* of the target unserializable interleaving, and therefore increase its occurrence probability.

Although above ideas are intuitive, there are several efficiency and effectiveness issues that we need to address:

(1) How long should the suspension be? An intuitive answer is to suspend the execution until the interleaving occurs, i.e., suspend c's thread until r executes or suspend r's thread until c is ready to execute. Unfortunately, the unserializable interleaving may never occur, as shown in Figure 8 (b). To avoid such endless suspension (deadlock), CTrigger sets a time-out threshold for each suspension point.

(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 more than one thread, e.g., two, execute the c instruction (Figure 8 (c)), suspending both threads may decrease the interleaving probability. Therefore, CTrigger only suspends one thread at a time. Secondly, a static instruction might have many dynamic instances. Suspending before every instances can result in huge slow-downs. 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 targeting interleavings from happening. CTrigger can address this issue by suspending the execution right before the outermost critical section that holds the targeting instruction.

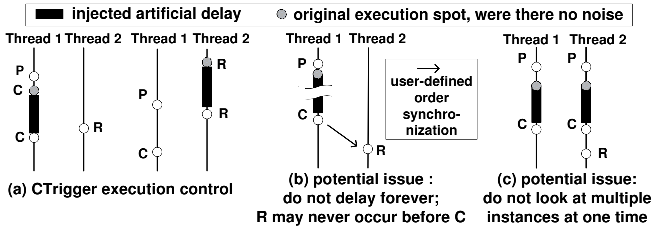
(4) Context sensitivity The occurrence of some unserializable interleavings depends on the program context or thread context. That is, they only happen when the involved instructions are executed upon certain stack frame or by certain threads. CTrigger provides the option to collect callstack and thread information from trace analysis, and use such information in execution control.

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 targeting 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 to 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 sever applications).

Implementation CTrigger controls execution via binary instrumentation using Pin [15]. CTrigger takes the list of unserializable interleavings provided by the CTrigger analysis, and instruments every instruction that involves in at least one unserializable interleaving. At run time, CTrigger intercepts every dynamic instances of these instructions and injects delay according to the above strategies.

Outcome Interpretation If a target unserializable interleaving is successfully forced by CTrigger and the software misbehaves (e.g., crashes, different results from testing oracles, errors detected by bug detectors), a bug is then exposed.


injected artificial delay
original execution spot, were there no noise
Thread 1
Thread 2
Thread 1
Thread 2
P
C
C
R
P
C
R
R
Thread 1
Thread 2
P
C
R
user-defined order synchro-nization
Thread 1
Thread 2
P
C
P
C
R

(a) CTrigger execution control

(b) potential issue : do not delay forever; R may never occur before C

(c) potential issue: do not look at multiple instances at one time

Figure 8. CTrigger's execution control and design issues

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 external macro providers) leading to atomicity violation bugs that generate wrong outputs
LU	LU	
Barnes	Barnes	

Table 2. Evaluated applications and atomicity violation bugs (*: Mozilla code is slightly modified to help compare the execution result with the oracle.)

Once a bug is exposed, CTrigger can reliably reproduce it 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 also gain more confidence about the software quality. If the targeting 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.

6. Evaluation

6.1 Methodology

To evaluate our ideas and CTrigger framework, we apply CTrigger on seven applications and evaluate how well it can expose the tested atomicity violation bugs inside these applications. These applications include three large open-source server/client applications, i.e., Apache, MySQL and Mozilla, one utility application, PBZIP2, and three SPLASH2 [26] benchmarks. We evaluate one or two *real world* atomicity violation bugs in each application [1]. The details are described in Table 1 (in Section 3.1) and Table 2.

The platform setting is the same as that in Section 3.1. The selection of testing inputs for the server/client applications are based on the original bug reports on 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 the existence of the SPLASH2 macro bugs in advance. They are exposed by CTrigger under testing with the default inputs.

We evaluate the effectiveness, efficiency and reproducibility of CTrigger: whether the bugs can be exposed, how quickly the bugs can be exposed, and how reliably the bugs can be reproduced after their first manifestation. We compare CTrigger with four other bug exposing mechanisms on the same platform as shown in Table 3.

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 [3]. The released version of CHESS [16] 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 [22]. Our implementation is based on Pin and the state-of-the-art open-source Valgrind-lockset race detection tool [18]. 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

Table 3. Evaluated concurrency testing methods

6.2 Efficiency and effectiveness

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 1000 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 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 ineffective as stress testing. Actually, since Pin framework slows down each testing run, it takes 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 and the program crashes at lock acquisition. However, it cannot help expose the other seven bugs. These seven bugs, like most real-world atomicity violation bugs, were introduced when programmers forgot to do synchronization. As a result, Synch-based testing slows down each testing run without improving the chance of exposing these bugs.

As regards to 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-

[1] CTrigger exposed one *previously unknown* bug in the macro library of SPLASH2 introduced by external macro providers. CTrigger also found four new buggy code regions in Apache (Figure 9), which have never been reported.

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

Table 4. The time (unit: second) spent to expose the tested atomicity violation bugs (**NO**: the bug was not exposed in our maximum testing time, which is one day for Apache, MySQL and Mozilla, half day for other small applications. *: compared with stress testing.)

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

Table 5. Breakdown of CTrigger bug exposing time (unit: sec) (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.)

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 between the reported racing instructions does not always lead to atomicity violation. Enforcing the race is insufficient to expose the bug.

CTrigger bug exposing time breakdown Table 5 shows the time spent in every step of CTrigger for exposing above bugs. CTrigger trace collection and analysis take about 1 to 60 seconds. The tracing 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 6.6 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.

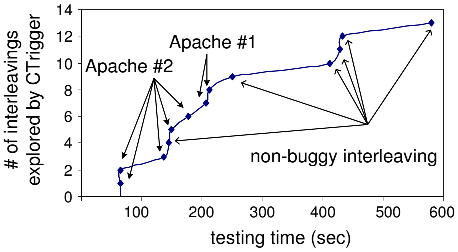


Figure 9. Unserializable interleavings additionally explored by CTrigger (The base line is the unserializable interleavings covered in profiling runs. The first 60 seconds are devoted to profiling runs and have no additional coverage).

6.3 Unserializable interleaving coverage

CTrigger can effectively explore low-probability unserializable interleavings, and improve the coverage within the unserializable interleaving space. Figure 9 shows the unserializable interleavings *additionally* explored by CTrigger for Apache compared with the stress testing (profiling runs). These additionally covered interleavings include both bug-triggering ones and non-bug-related ones, as denoted by Figure 9. Covering bug-triggering ones helps CTrigger to expose the two Apache bugs; covering non-bug-related ones validates the correctness of these low-probability interleavings. In contrast, the number of interleavings explored in stress testing is saturated after around 60–70 seconds.

6.4 Reproducing a previously-exposed bug

As shown in Table 6, 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

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

Table 6. The time (unit: second) spent to reproduce an exposed bug (**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 expose the corresponding bug even once.)

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.

6.5 CTrigger infeasible interleaving pruning

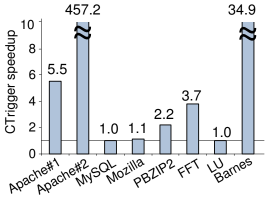
Identifying infeasible interleavings is critical for CTrigger to set a reachable testing goal. Table 7 shows that CTrigger feasibility analysis is very effective: 37–96% of the potential unserializable interleavings are successfully identified as infeasible. In order to examine the stability of the feasibility 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.

6.6 CTrigger low-probability interleaving ranking

We evaluate how 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.

BugId.	# of Mem-Acc Instructions	# of Potential UI*	# of Feasible UI	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

Table 7. Effectiveness of infeasible interleaving pruning (* : The two Apache bugs can be triggered using the same input, so we just put one result here. MySQL is using a different input with that in Section 3. The pruning percentage is based on the number of potential unserializable interleavings.)



(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	4th out of 143

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

Figure 10. Efficient CTrigger ranking

As shown in Figure 10(a), 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 Figure 10(b), using its local-gap based probability estimation.

7. Related Work

Concurrent program testing Several recent works [3, 4, 16, 22] on exposing concurrency bugs are closely related to CTrigger. We have already discussed them in detail in Section 1.2. There have also been many inspiring works [9, 24, 8, 28, 11] on designing interleaving coverage criteria to evaluate/measure the coverage of concurrency testing. Due to complexity concerns, these work have not guided practical testing to expose concurrency bugs in large programs. CTrigger closely follows the guidance of the most recent work [11] in this direction and has reasonably high coverage in the unserializable interleaving space that is inherently connected with atomicity violation bugs.

How to generate input test cases for concurrent programs has also been studied [23]. CTrigger can work together with testing input generation techniques to improve the effectiveness of concurrent program testing.

Concurrency bug detection Much research has been conducted on detecting different types of concurrency bugs [19, 20, 29, 5, 13, 27, 7]. In general, exposing software bugs is complementary to bug detection, as most dynamic bug detectors require bugs to manifest during the bug detection runs. Concurrency testing tools like CHESS, RaceFuzzer and also our CTrigger can make hidden bugs manifest for bug detectors to catch.

Concurrency bug avoidance and surviving Our work is also related to concurrent programming model design like transactional memory [10]. Even with transactional memory, atomicity violations can still happen. Therefore, the effectiveness and benefits of CTrigger still apply.

In recent Atom-Aid work [14], transactional memory is cleverly leveraged to help survive atomicity violation bugs that have escaped the in-house testing. Such production-run surviving techniques and development-site exposing techniques like CTrigger can well complement each other.

8. Conclusions and Future Work

This paper has presented a study of the interleaving characteristics in stress testing and proposed 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 program execution to force them to occur.

Our experiments with seven real-world server/desktop and scientific applications show that CTrigger is effective at exposing atomicity violation bugs: it achieves 2 – 4 orders of magnitude speedup in bug exposing and 2 – 5 orders of magnitude speedup in bug reproducing (for diagnosis purpose) 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 well complements the existing techniques on improving the quality of concurrent programs: bug detectors can detect bugs more quickly and accurately; and developers can save 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., multi-variable involved bugs). Future work can also combine it with test input generation and other interleaving testing mechanisms.

Acknowledgments

We thank the anonymous reviewers for useful feedback, the Opera groups for useful discussions and paper proofreading. This research is supported by NSF CCF-0325603 grant, NSF CNS-0615372 grant, NSF CNS-0347854 (career award), DOE Early Career Award DE-FG02-05ER25688, and Intel gift grants.

References

[1] P. Barford, and M. Crovella. Generating representative Web Workloads for network and server performance evaluation. In *ACM SIGMETRICS*, June 1998

[2] B. Beizer. Software testing techniques, 2nd edition. *New York: Van Nostrand Reinhold*, 1990

[3] A. Bron, E. Farchi, Y. Magid, Y. Nir and S. Ur. Applications of synchronization coverage. In *PPoPP*, 2005

[4] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multi-threaded Java program test generation. In *IBM Systems Journal*, 2002

[5] C. Flanagan, and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, 2004

[6] C. Flanagan, and S. Qadeer. A type and effect system for atomicity. In *PLDI*, 2003

[7] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, 2008

[8] M. J. Harrold, and B. A. Malloy. Data flow testing of parallelized code. In *ICSM*, 1992

[9] P. V. Koppol, and K.-C. Tai. An incremental approach to structural testing of concurrent software. In *ISSTA*, 1996

[10] J. R. Larus, and R. Rajwar. Transactional memory. *Morgan & Claypool*, 2006

[11] S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *FSE*, 2007

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – A comprehensive study of real world concurrency bug characteristics. In *ASPLOS*, 2008

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006

[14] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA*, 2008

[15] 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. In *PLDI*, 2005

[16] M. Musuvathi, and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007

[17] M. Musuvathi, S. Qadeer, T. Ball, and G. Basler. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008

[18] N. Nethercote, and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007

[19] R. H. B. Netzer, and B. P. Miller. Improving the accuracy of data race detection. In P̂PoPP, 1991

[20] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson Eraser: A dynamic data race detector for multithreaded programs. In *ACM TOCS*, 1997

[21] Software Bug Contributed to Blackout. SecurityFocus. *http://www.securityfocus.com/news/8016*

[22] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008

[23] K. Sen, and G. Agha. Automated systematic testing of open distributed programs. In *FSE*, 2006

[24] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. In *IEEE Transactions on Software Engineering*, 1992

[25] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006

[26] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995

[27] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005

[28] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *ISSTA*, 1998

[29] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *SOSP*, 2005