AtexRace: Across Thread and Execution Sampling for In-House Race Detection

Yu Guo †
Department of Computer Science
Western Michigan University
Kalamazoo, MI, USA
yu.guo@wmich.edu

Yan Cai †, ‡
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
ycai.mail@gmail.com

Zijiang Yang ‡
Department of Computer Science
Western Michigan University
Kalamazoo, MI, USA
zijiang.yang@wmich.edu

ABSTRACT

Data race is a major source of concurrency bugs. Dynamic data race detection tools (e.g., *FastTrack*) monitor the executions of a program to report data races occurring in runtime. However, such tools incur significant overhead that slows down and perturbs executions. To address the issue, the state-of-the-art dynamic data race detection tools (e.g., *LiteRace)* apply sampling techniques to selectively monitor memory accesses. Although they reduce overhead, they also miss many data races as confirmed by existing studies. Thus, practitioners face a dilemma on whether to use *FastTrack*, which detects more data races but is much slower, or *LiteRace*, which is faster but detects less data races. In this paper, we propose a new sampling approach to address the major limitations of current sampling techniques, which ignore the facts that a data race involves two threads and a program under testing is repeatedly executed. We develop a tool called *AtexRace* to sample memory accesses across both threads and executions. By selectively monitoring the pairs of memory accesses that have not been frequently observed in current and previous executions, *AtexRace* detects as many data races as *FastTrack* at a cost as low as *LiteRace*. We have compared *AtexRace* against *FastTrack* and *LiteRace* on both Parsec benchmark suite and a large-scale real-world MySQL Server with 223 test cases. The experiments confirm that *AtexRace* can be a replacement of *FastTrack* and *LiteRace*.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging • Theory of computation → Program verification.

KEYWORDS

Data race, sampling, concurrency bugs

† Co-first author.
‡ Corresponding authors.

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
http://dx.doi.org/10.1145/3106237.3106242

ACM Reference format:

Yu Guo, Yan Cai, and Zijiang Yang. 2017. AtexRace: Across Thread and Execution Sampling for In-House Race Detection. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4-8 2017 (ESEC/FSE'17),* 11 pages. *http://dx.doi.org/10.1145/3106237.3106242*

1. INTRODUCTION

A data race (or race for short) occurs when two or more threads access the same memory location at the same time, and at least one of them is a write [16]. Race is a major source of concurrency bugs [38] and may result in real-world disasters [23][29][40].

Static race detection techniques are scalable but may report many false positives [25][37][42][51]. Various filters have been developed to address this issue. However, false positives remain and false negatives emerge with these filters in the static race detection tools [37]. Dynamic techniques report much fewer false positives. They are mainly based on either the *lockset discipline* [44] or the *happens-before relation* [16][27]. The former requires that all accesses to a shared memory location should be protected by a common set of locks. The latter [27] is usually implemented via vector clocks [16] to track the status of threads, locks and memory locations. Happens-before based race detectors (HB detectors for short) report less false positives but incur higher overhead than the lockset based ones. *FastTrack* [16], by avoiding a large number of $O(n)$ operations on memory accesses, reduces the overhead to the level as that of the lockset based race detectors. Even so, by continuously monitoring all memory accesses of a multithreaded program, *FastTrack* still incurs from 400% to 800% overhead [10][16][54].

Sampling [7][34][58] is a promising technique to reduce the overhead of dynamic detectors by selectively monitoring memory accesses. There are two types of sampling. With the assumptions that concurrency bugs cannot be eliminated during testing and daily uses of released software provide a large test bed, the first type attempts to detect races at user sites, including *Pacer* [7], *CRSampler* [12], and a possible adaption of *DataCollider* [14]. This type of sampling must be extremely light-weight (i.e., <5% overhead [3][26][31][59]). And they usually detect a small number of data races depending on the sampling rate and the overhead limit.

The second type aims at reducing in-house testing overhead. Before releasing a software, the developers usually test the program against a large number of test cases, and for each test case, the program may be executed multiple times. Lower overhead

```
Thread t1                    Thread t2
1.  for (...){               5.  for (...){
2.    if(...) f1();          6.    if(...) f3();
3.    else  f2();            7.    else  f4();
4.  }                        8.  }
```

Figure 1. A code sketch with two threads and four function calls.

enables more testing and thus less races in the tested software. *LiteRace* [34] is a representative tool in this category. It is based on the hypothesis that undetected races often exist in *cold functions* that have not been frequently called. Therefore, *LiteRace* reduces overhead by avoiding the sampling of memory accesses in *hot functions* that have been frequently executed.

Figure 1 shows a code sketch with two threads t_1 and t_2. Functions f_1 and f_2 are repeatedly executed in t_1, and f_3 and f_4 are repeatedly executed in t_2. Races occur when f_1 and f_4 execute simultaneously, and when f_2 and f_3 execute simultaneously. Assume that t_1 is executed more frequently than t_2 and the *then*-branches are executed more frequently than the *else*-branches. Initially all functions are cold, but quickly f_1 becomes hot while other three functions are still cold. At this moment *LiteRace* stops monitoring f_1 and becomes faster than *FastTrack* because the latter still continuously monitors f_1. After a while f_2 and f_3 get a chance to be executed. Since both functions are cold, *LiteRace* still monitor their executions and thus can report the race between f_2 and f_3 at a cost lower than that of *FastTrack*. Next f_4 is executed at the same time with f_1. In this case *LiteRace* fails to detect the race between f_1 and f_4 because it already stopped tracking f_1. On the other hand, *FastTrack* can catch the race because it still monitors f_1. This example illustrates the dilemma in choosing between full scale tools and sampling based tools. A programmer has to either sacrifices efficiency for accuracy, or sacrifices accuracy for efficiency.

We argue that programmers do not have to choose between efficiency and accuracy. This is achievable because there are two major limitations in current sampling techniques. From the definition, a race occurrence requires *two memory accesses of different threads*. Therefore, sampling memory accesses in isolation is ineffective. The aforementioned example shows that a function f may become hot before any other functions that race with f. In this case, sampling those functions that race with f is useless. We call this inefficiency *thread-local sampling* because it does not consider other threads when it decides whether to sample the current thread. The second major limitation is that sampling algorithms remain the same for all the executions of a program. This is ineffective because in in-house testing a program is usually executed repeatedly against a large set of test cases. For a multithreaded program, a develop may even run it multiple times under a single test case. The net effect of current sampling strategy is that those functions that are cold in individual execution but hot in accumulative executions are repeatedly sampled. We call this inefficiency *execution-local sampling* as it does not consider previous executions when decides whether to sample the current execution.

In this paper, we propose *AtexRace*, a new dynamic race detection tool based on across-thread and across-execution sampling. It is designed to sample memory access *pairs* from different threads and is also aware of executions. However, several challenges must be resolved to make it practical. Firstly, tracking memory accesses across threads incurs much larger overhead than tracking thread-local data only (e.g., higher cache miss rate). Secondly, even if a pair of memory accesses is observed to be race-free before, it does not mean that the pair will not race later. This is because while instructions are static, the memory addresses they access are dynamic. Lastly, *AtexRace* avoids sampling previously observed memory pairs, which requires additional recording. With increasing number of executions, the recorded data set may grow rapidly, which further slows down the sampling processes (e.g., the need of more time to search memory access pairs).

We have implemented *AtexRace*, *FastTrack*, and *LiteRace* on top of Pin [32] and evaluated them on five programs on Parsec benchmark suite [2] and a real-world program MySQL. In the experiments, we run each Parsec program for 100 times and run MySQL under 223 different test cases. The experimental results surprisingly show that *AtexRace* detects more races in Parsec benchmarks than *FastTrack* does! As for MySQL, *AtexRace* detects almost the same number of dynamic races as that by *FastTrack*. *LiteRace*, as predicted, detects significantly fewer races than both *FastTrack* and *AtexRace*. If we do not consider the same races that are detected again, *AtexRace* detects more unique races than *FastTrack* and *LiteRace*. In terms of efficiency, *LiteRace* and *AtexRace* reduce almost the same percentage of overhead on top of *FastTrack*. This makes *AtexRace* a replacement of *FastTrack* and *LiteRace*. The main contributions of this paper are:

- We present a novel sampling technique called *AtexRace* toward race detection. Unlike existing sampling techniques that are thread-local and execution-local, *AtexRace* is across-thread and across-execution.
- To make *AtexRace* practical, we have designed optimization heuristics that include (1) utilizing thread-local storage to avoid competing accesses to shared sampling data set, (2) exploiting burst sampling strategy to enhance race coverage, and (3) adopting *n-frequent* (function) pairs to improve map lookup efficiency.
- We have implemented *AtexRace* and conducted a set of experiments on benchmarks including a real-world large-scale program MySQL. Our experiments confirm that *AtexRace* detects as many races as *FastTrack* at a cost as low as *LiteRace*. The tool is at http://lcs.ios.ac.cn/~yancai/atexrace .

2. BACKGROUND

2.1 Multithreaded Programs

A *multithreaded program* consists of a set of threads, a set of locks (or lock/synchronization objects), and a set of memory locations (or locations for short). Each thread t has a unique thread identifier tid, denoted as $t.tid$. During an execution of a multithreaded program p, each thread t performs a sequence of events $\langle e_1, e_2, ..., e_k \rangle$. An event can be one of the following types: (1) ***acq(m)*** or ***rel(m)***: synchronization events: to acquire or release a lock m. (Other synchronization events can be similarly defined [16].) (2) ***read(x)*** or ***write(x)***: memory access events: to read from or write to a memory location x, and (3) ***call(f)*** or ***return(f)***: control events: to execute events in function f or return to execute the events from the previous function f.

```
Shared variables: int x = 0, y = 0; Lock m, n;
Input: ⟨a, b⟩;
            Thread t1                              Thread t2
1.  for (i =1 to 2 × a){                 17. for (j =1 to 2 × b) {
2.    if(i < a)      f1( i );            18.   if( j < b)      f3( j );
3.    else           f2( i );            19.   else           f4( j );
4.  }                                    20. }
5.                                       21.
6.  Function f1(i){                      22. Function f3(j){
7.    acq(m)                             23.   acq(m)
8.    x += i;                            24.   x += j;
9.    y += i;                            25.   y += j;
10.   rel (m)                            26.   rel (m)
11. }                                    27. }
12. Function f2(i){                      28. Function f4(j){
13.   acq(n)                             29.   acq(n)
14.   y += i;                            30.   y += j;
15.   rel (n)                            31.   rel (n)
16. }                                    32. }
```

Figure 2. A program with races on variable *y* between line 9 and line 30, and between line 14 and line 25.

2.2 Data Races

Data races can be defined according to either the lockset discipline [44] or the happens-before relation [27]. In this paper, we adopt the later one as it is relatively precise [16]. However, our sampling strategy is independent from concrete definitions. The happens-before relation (denoted as $\rightarrowtail$, HBR for short) is defined by the three rules [27]: (1) If two events α and β are performed by the same thread, and α appears before β, then $\alpha \rightarrowtail \beta$, (2) If α is a lock release event and β is a lock acquire event on the same lock, and α appears before β, then $\alpha \rightarrowtail \beta$, and (3) If $\alpha \rightarrowtail \beta$ and $\beta \rightarrowtail \gamma$, then $\alpha \rightarrowtail \gamma$. Given two memory access e_1 and e_2 that access the same memory location and one of them is a write events, a race occurs if neither $e_1 \rightarrowtail e_2$ nor $e_2 \rightarrowtail e_1$.

3. MOTIVATIONS

3.1 Motivating Example

Figure 2 shows a multithreaded program *p* that extends the code sketch given in Figure 1. The program consists of two threads t_1 and t_2 operating on two shared variables *x* and *y*. There are two locks *m* and *n* protecting accesses to shared variables *x* and/or *y*. Given two parameters $\langle a, b\rangle$, thread t_1 consecutively calls function f_1 for *a* times and then calls function f_2 for *a* times within a loop (lines 1–4); and thread t_2 performs similar calls to functions f_3 and f_4 each for *b* times (lines 17–20). The four functions increase the values of *x* and *y* based on the passed parameters.

Due to the parallel execution of the two threads in Figure 2, any pair of functions between threads t_1 and t_2 can potentially be executed simultaneously. The four pairs of functions that can be executed at the same time are $\langle f_1, f_3\rangle$, $\langle f_1, f_4\rangle$, $\langle f_2, f_3\rangle$, and $\langle f_2, f_4\rangle$. For the pairs $\langle f_1, f_4\rangle$ and $\langle f_2, f_3\rangle$, as the variable *y* is protected by different locks (i.e., lock *m* in function f_1 and f_3 but lock *n* in function f_2 and f_4), races may occur. For example, if lines 9 and 30, or lines 14 and 25, are executed at the same time, the program may produce incorrect results due to the race on variable *y*.

3.2 Heavy Overhead of Dynamic Data Race Detection

Dynamic race detectors usually incur large overhead [16][12] due to heavy instrumentation and race checking per memory access. This is unavoidable because they have to track whether the pair of a current access and a previous access violates any HBR. We use the memory access "*x += i*" in Figure 2 (line 8) to illustrate the overhead. For each access to the location *x*, one function call like *onRead*(*x*) or *onWrite*(*x*) is inserted [16], see Figure 3. Within these calls, there are two types of operations that cost time [16][17][34].

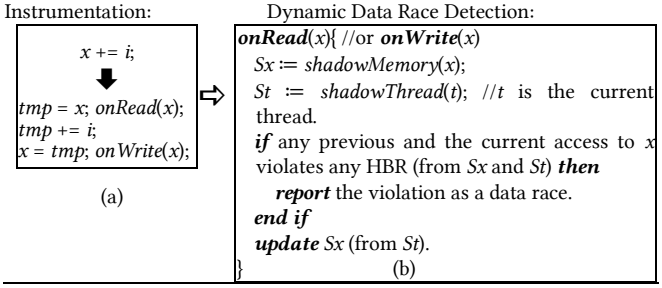


Figure 3. An illustration on the instrumentation and race detection for each memory access.

The first type is from fetching shadow data (or meta data [16][34]) for each thread and each memory location. For each memory location, dynamic ones track all accesses to it and store the information at *shadow memory* (e.g., *shadowMemory*(*x*) in Figure 3). Similarly, *shadow threads* (e.g., *shadowThread*(*t*) in Figure 3) are used for each thread. Therefore, a memory access in the original program is accompanied by several additional memory accesses to get the shadow data for a memory location and a thread (e.g., *Sx* and *St* for memory location *x* and thread *t*, respectively). For the shadow threads, many instrumentation frameworks provide fast access interface (e.g., Thread Local Storage in Pin [32] and Thread Execution Blocks in Windows [49]). However, to the best of our knowledge, no fast access to shadow memory is supported. The latter is much difficult in practice. For Java program, the shadow memory could be allocated together with the memory allocation in the original program [17]. However, for C/C++ programs, this becomes difficult.

The second type is from race checking. After fetching shadow data, the values from two shadow data (i.e., from the memory location and from the current thread) are checked to detect any HBR violation. This process also involves additional memory accesses, especially the write operations to maintain the access information (i.e., to update *Sx* in Figure 3). Note that, *FastTrack* optimizes the process on race detection but it still requires maintenance (read and write) on shadow data.

3.3 Limitations of Existing Sampling Approaches

Although dynamic approaches incur heavy overhead, they are usually precise for data race detection. Therefore, sampling approaches have been proposed to reduce the runtime overhead by tracking a subset of events and to detect races among them.

Existing sampling approaches include deployed sampling [7][12] and in-house sampling [3][14]. The former approaches are deployed at the users' sites after a software is released. Such approaches are based on the crowd-source testing: if there are many users, races escaped during in-house testing may be detected by sampling a tiny portion of an execution by each user. Hence, deployed sampling requires extremely low run time

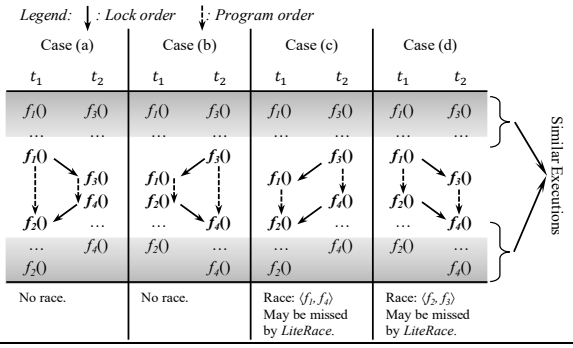


Figure 4. Three executions scenarios of the program in Figure 2 and the similarity of different executions.

overhead (e.g., 5% [7]). The latter attempts to reduce runtime overhead during in-house testing phase. The representative tool is *LiteRace* [34]. As our approach falls into this category, we discuss *LiteRace* in detail in the rest of this subsection.

LiteRace is based on the cold-region hypothesis: races are likely to occur when a thread is executing a cold region (i.e., the program portion not frequently executed). *LiteRace* tries to avoid tracking those frequently executed functions (i.e., hot functions). Initially, it sets up a thread-local sampling rate of 100% for each function. This sampling rate is then gradually reduced whenever a function is called by the corresponding thread until the rate reaches a low bound (e.g., 0.1%). For example, in Figure 2, *LiteRace* initially checks all events from function f_1. After the function is executed once, the thread-local sampling rate of function f_1 by thread t_1 is reduced. If thread t_2 calls function f_3, the sampling rate of function f_3 by thread t_2 is also reduced in the same way.

LiteRace reduces runtime overhead at the expense of its race detection capability. For example, in an evaluation, it only detected about 70% of frequent data races and about 50% of rare data races of continuously monitoring tools such as *FastTrack* [34]. This is also verified by other works [7]. We explain this limitation of *LiteRace* via our running example in Figure 2.

Figure 4 gives four execution cases that illustrate how the functions in the two threads interleave. In each case, a column shows the execution of a thread in term of function calls. The difference between the four cases is at how the last call to function f_1 and the first call to function f_2 by thread t_1 interleaves with the last call to function f_3 and the first call to function f_4 by thread t_2.

Recall that locks m and n protect the accesses to y in functions f_1 and f_3, and in functions f_2 and f_4, respectively. Because two different locks are used, a race on variable y occurs when either functions f_1 and f_4 execute in parallel or functions f_2 and f_3 execute in parallel. No data race occurs in either case (a) or case (b) because neither pair of functions may execute in parallel. That is, we can infer that accesses in function f_1 happen before accesses in function f_4 by following lock acquisition order (i.e., the solid arrows) and the program order within each thread (i.e., the dashed arrows). The same reasoning also applies on the functions f_3 and f_2. However, for case (c), there is no strict order between the accesses in functions f_1 and f_4; hence, a HB detector may detect the race on y from the two functions. Due to same reason, for case (d), the race on y from functions f_3 and f_4 may also be detected.

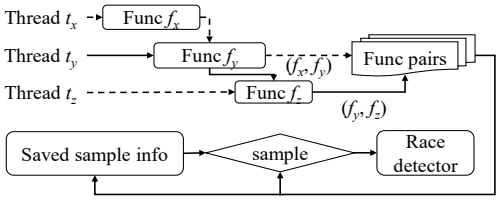


Figure 5. The overview of *AtexRace* framework.

When *LiteRace* is applied to the four cases in Figure 4, a function is not tracked after it has been called by the same thread for certain number of times. Therefore, function f_1 executed by thread t_1 and f_3 executed by thread t_2 are no longer tracked if they become hot functions. In case (c), even when function f_1 and function f_4 execute in parallel, *LiteRace* may miss the race. This is because *LiteRace* only tracks the cold function f_4 without tracking function f_1. Similarly, In case (d), *LiteRace* may also miss the race.

We believe the main reason that *LiteRace* frequently fails to detect races, as observed previously [7], is that its sampling across threads is not coordinated. Since a data race requires two conflict memory accesses from two threads, sampling one memory access from one thread but not the other is useless. This is illustrated by cases (c) and (d) above. Consider an extreme case where all races involve a function. If this particular function is considered hot after being visited several times, all future samplings are in vein.

Besides the issue of thread-local sampling, *LiteRace* also suffers from execution-local sampling. When testing a multithreaded program by running it repeatedly against a large number of test cases, the same thread interleaving, with minor variations, tend to be exercised since thread schedulers generally switch among threads at the same program locations. In addition, although the whole program execution may witness variants from one run to another, partial execution may exhibit similar behaviour. For example, even all the four cases in Figure 4 are executed in different runs, the initial interleaving of two threads are similar. That is, functions f_1 and f_3 interleave until functions f_2 or f_4 is called. We highlight these function calls in grey background for illustration purpose. As *LiteRace* is unware of execution similarities, it adopts the same sampling strategy across different executions. The net effect of strategy is that those functions that are cold in individual execution but hot in accumulative executions are repeatedly sampled. This defeats the principle of sampling that the real cold cases should be tracked.

The two main limitations of current sampling techniques motivate our work in this paper.

4. OUR APPROACH

4.1 Goal and Challenges

In this section, we present our approach to fix the two limitations of current sampling techniques. In order to address thread-local sampling, our insight is that whether to sample a memory access event should also depend on the execution of other threads and those already observed executions. That is, even if a memory address has been accessed by a thread many times, we may still need to sample it if a second thread access the memory

address for the first time. As for execution-local sampling, our idea is to keep and store sampling information from previous runs. Except the first execution that starts with a cold run, the subsequent executions load sampling information of accumulated prior executions. Although such approach incurs overhead, we blieve less sampling with optimization heuristics can lead to net benefit.

The new sampling approach, *AtexRace,* also works at function levels like *LiteRace*. But unlike *LiteRace, AtexRace* mainly samples accesses inside a pair of functions whose simultaneous executions are not observed before, including previous executions. Unfortunately, a basic implementation of the idea is not very scalable. Firstly, tracking executions across threads usually incur larger overhead than thread-local tracking. Secondly, even two functions are observed to have executed in parallel before, data races may still occur within them. Thirdly, as *AtexRace* performs sampling across different executions instead of within a single execution, it must effectively record function interleaving information to be used in the subsequent executions.

4.2 Basic *AtexRace* Algorithm

The overview of *AtexRace* is shown in Figure 5. During execution. when function f_y in thread t_y is being executed, *AtexRace* collects all the functions (e.g., f_x and f_z) that are being executed by other threads. By doing so *AtexRace* forms pairs of functions that are being executed simultaneously (e.g., $\langle f_x, f_y \rangle$). It then makes a sampling decision according to whether a pair of functions have been executed in parallel before. If so, neither function is sample; otherwise, both are sampled. If a function is sampled, all its events are passed to a race detector. At the end of an execution, all function pairs are saved and will be used in the next execution. Note, in order not to report false positives, all synchronizations are fully sampled. This is the same as *LiteRace*.

Algorithm 1 gives the basic *AtexRace* algorithm that takes a program *p* and a set of function pairs *FPair* that have been observed in the previous executions. The first three lines initialize two necessary runtime data structures: a map *F* that maintains the functions being executed by each thread, and a map *S* that indicates whether memory accesses from a thread should be sampled. Both *F* and *S* are empty initially.

The function *onCallFunc* (lines 5–19) is the core of our sampling. Whenever a function *f* is to be executed (i.e., at the entrance of function *f*) by a thread *t*, for every other thread *t'* in program *p*, *AtexRace* checks whether the pair $\langle f, F(t') \rangle$ already exists in *FPairs*. If not, *S* is updated to map both threads *t* and *t'* to *true*; otherwise, *S* maps *t* to false. A true value of *S(t)* mandates sampling of the current memory access in thread *t* and a false value does the opposite. Next, *AtexRace* executes events in function *f* (line 14) and samples its memory accesses (i.e., function *onMemoryAccesses*) if *S(t)* is true. At the end of the call to function *f*, *AtexRace* merges *FPairs* and the observed function pairs $\langle f, F(t') \rangle$, which indicates that the function *f* and another function $F(t')$ in thread *t'* have been executed simultaneously.

In practice, two functions from different threads are usually called at different time. Therefore, it is the case that, a function *f* is initially not sampled but later it should be sampled as a different thread *t'* calls a function $f' = F(t')$ and the pair $\langle f, F(t') \rangle$ is never observed before. This is considered by *AtexRace*. We can see

Algorithm 1: Basic *AtexRace*

```
   Input: p – a multithreaded program.
   Input: FPairs – a set containing functions.
1.  let F be an empty map from a thread to a function
2.  let S be a map from a thread to a Boolean value.
3.  for each thread t ∈ p, F(t) := ∅, S(t) := true end for
4.
5.  Function onCallFunc (Thread t, Func f)
6.      let F(t) := f and St := false //St is a temporary variable that keeps S(t)
7.      for each thread t' ∈ p, t ≠ t' do
8.          pair := ⟨f, F(t')⟩
9.          if pair ∉ FPairs then
10.             St := true
11.             S(t') := true
12.         end if
13.     end for
14.     S(t) := St
15.     execute f
16.     for each thread t' ∈ p, t ≠ t' do
17.         FPairs := FPairs ∪ {⟨f, F(t')⟩}
18.     end for
19. end Function
20. Function onMemoryAccess(Thread t, Event e)
21.     if S(t) = true then
22.         call data race detector
23.     end if
24. end Function
25. save FPairs
```

from lines 10 and 11 that at the call entrance to function *f*, thread *t'* also performs an iteration over other threads at line 7. At the iteration on thread *t*, it cannot find the pair in *FPairs*. Then it maps both threads *t'* and *t* to be true value in structure *S*. So, the function *f* executed by thread *t* has to be sampled.

4.3 Limitations of Basic *AtexRace*

The basic sampling algorithm of *AtexRace* suffers from the two limitations: (1) given two function f_1 and f_2, even if their parallel execution has been observed and tracked (thus become hot), races between them may still not detected; and (2) significant overhead resulted from across thread and execution sampling.

The first limitation is the issue of *Race Coverage*. A function usually contains multiple basic blocks (BBLs). An execution of a function does not mean all its BBLs are executed. For example, Figure 6 shows two functions f_5 and f_6 that contain two races on variables *x* (lines 6 and 21) and *y* (lines 18 and 9). There are four BBLs b_{11}, b_{12}, b_{21}, and b_{22} (we omit other BBLs in the *if* statement for simplicity). Since the two threads in the example execute $f_5(10)$ and $f_6(100)$, respectively, only b_{11} and b_{22} are executed. Hence, the race on variable *x* (lines 6 and 24) is detected while the race on variable *y* (lines 19 and 10) is not. If the pair $\langle f_5, f_6 \rangle$ is considered hot after this execution, the race on *y* can never be detected by the basic *AtexRace*. One approach to address this issue is to degrade the sampling level from functions to BBLs and then apply either *LiteRace* or the Part 1 of our *AtexRace*. However, this bring heavy runtime overhead and may even incur more overhead than a full detector such as *FastTrack*. This is because, compared to a function, a BBL usually contains much fewer instructions. As a result, the sampling overhead (in time) per BBL may already larger than the race detection overhead without sampling. Because sampling algorithm is not extremely lightweight, it is not worthy to perform sampling at BBL level.

```
Thread t1                                  Thread t2
1.  f5(10);                                14. f6(100);
2.                                         15.
3.  Function f5(i){                        16. Function f6(j){
4.    if ( i < 100){                       17.   if ( j < 100){
5.      acq(m)        ┐                    18.     acq(m)        ┐
6.      x ++;         │ BBL b11            19.     y ++;         │ BBL b21
7.      rel (m)       ┘                    20.     rel (m)       ┘
8.    } else{                              21.   } else{
9.      acq(n)        ┐                    22.     acq(n);       ┐
10.     x += y;       │ BBL b12            23.     y += x;       │ BBL b22
11.     rel(n);       ┘                    24.     rel(n);       ┘
12.   }                                    25.   }
13. }                                      26. }
```

Figure 6. A program consisting of two threads with two data races on variables *x* (lines 6 and 23) and *y* (lines 19 and 10).

On the other hand, for C/C++ programs, even an instruction contains one or more memory accesses, it is possible that each execution of the instruction may accesses different memory location. For example, considering the following two lines of code:

```
1.  Object obj = &getObj (...);
2.  obj ->val ++;
```

We can observe that, within the same and repeated executions of the two lines, if the pointer *obj* points to different objects, it accesses different memory locations at line 2. Therefore, for sampled memory accesses, it is still necessary to track them.

The second limitation is the *Sampling Overhead* of *AtexRace* itself. A sampling tool should sample as fewer memory accesses as possible to reduce the overhead. At the same time, it should also try to incur less overhead from its sampling strategy. *LiteRace* adopts thread-local sampling and requires two thread-local counters per-function. This can be efficiently implemented [34].

For *AtexRace*, there are expansive map queries (i.e., *FPairs*) on each function call (lines 9–10). These operations bring heavy slowdown for two reasons. Firstly, with the increasing number of function calls by multiple threads, the size of *FPairs* also increases, resulting in a large data set. For example, in our experiment, after 223 executions on MySQL, there are nearly 70,000 function pairs. A query over such a large map is time consuming. Secondly, the map *FPairs* is accessed by multiple threads. This requires synchronizations among different threads when they operate on the map. Such synchronization incurs further slowdown. Besides, when different threads access the map *FPairs*, the cache miss rate will be higher because once a thread updates the map, all other threads that query the map must wait until their local caches are updated. This again leads to additional time consumption. All these reasons bring challenges to reduce the overhead of our sampling algorithm *AtexRace* itself.

4.4 Optimizations

Algorithm 2 is an enhancement to the basic *AtexRace* algorithm that addresses the two kinds of limitations.

To address the issue of race coverage, *AtexRace* further samples those sampled function pairs in order to increase their coverage on data race detection. This corresponds to lines 18–24 in Algorithm 2. For this part, *AtexRace* accepts a sampling rate (i.e., the input *r* to Algorithm 2) and samples the function pair according the rate. Note that, *AtexRace* does not perform a simple sampling that generates a random number and compares the

Algorithm 2: Complete *AtexRace*

Input: p – a multithreaded program.
Input: r – a sampling rate.
Input: n – a number determine n-frequent value
Input: *FPairs* – a map (from functions pairs to counters) of the last n - 1 executions.

```
    //Initialization
1.  let F be an empty map from a thread to a function
2.  let S be a map from a thread to a Boolean value.
3.  let FP be a map from a thread to a copy of FPairs. //thread-local maps
4.  for each thread t ∈ p do
5.      F(t) := ∅
6.      S(t) := true
7.      FP(t) := FPairs //deep clone
8.  end for
9.  //Runtime Sampling
10. Function onEnterFunc(Thread t, Func f)
11.     let F(t) := f and St := false  //St is a temp variable that keeps S(t)
12.     for each thread t' ∈ p, t ≠ t' do
13.         pair := ⟨f, F(t')⟩
14.         if pair ∉ FP(t) then
15.             St := true
16.             S(t') := true
17.         else
18.             FP(t) := FP(t) ∪ {⟨pair, Counter(FP, pair) + 1⟩}
19.             if counter(pair, FP(t)) satisfies r then
20.                 St := true
21.                 S(t') := true
22.             else
23.                 St := false
24.             end if
25.         end if
26.     end for
27.     S(t) := St
28.     execute f
29.     for each thread t' ∈ p, t ≠ t' do
30.         FP(t) := FP(t) ∪ {⟨pair, 1⟩}
31.     end for
32. end Function
33. Function onMemoryAccess(Thread t, Event e)
34.     if S(t) = true then
35.         call data race detector
36.     end if
37. end Function
38. //The End of an Execution
39. Let FPairs' be an empty map.
40. for each thread t ∈ p do
41.     FPairs' := FPairs' ∪ F(t)
42. end for
43. save FPairs'
```

random number with the given sampling rate. Instead, *AtexRace* adopts burst sampling strategy [34]. It samples the first n consecutive calls out of all m calls to a function such that the rate ($n \div m$) × 100% equals to the given sampling rate r. For example, if the sampling rate is 10%, it samples the first 10 calls and discards the next 90 calls to the same function, resulting the sampling rate of 10%. Of course, to implement this functionality, a counter mapped from each function pair is required. Hence, the original set of function pairs is changed into a map (see the fourth input and the lines 18, 19 and 29 in Algorithm 2).

To overcome the second kind of limitations, we firstly propose to use thread-local maps. In Algorithm 2, we use the symbol *FP* to denote the thread-local maps of function pairs. That is, we allocate one map structure for each thread; and when *AtexRace*

starts an execution, it duplicates the given map data (line 7). During an execution, *AtexRace* only checks whether the pair exists in the map *FP* of the current thread (lines 14 and 19). If a pair already exists in a thread-local map, its counter is incremented by 1 at line 18. At the end of an execution, *AtexRace* merges all thread-local maps and saves the merged map (lines 39–43).

Secondly, we do not record all function pairs observed in previously executions. Instead, we only keep the recently frequently observed function pairs. Given an execution e and a number n ($n \geq 1$), we define a function pair $\langle f_x, f_y \rangle$ to be *n-frequent* with respect to execution e if $\langle f_x, f_y \rangle$ is observed in current and all the n-1 previous executions. Specially, when the value of n is 1, the *1-frequent* function pairs are those observed in the current execution. By keeping only, the *n-frequent* function pairs, the recorded function pairs are those frequently executed. This is reasonable not to sample these frequent function pairs to reduce sampling overhead. Hence, for each execution, the number of function pairs taken as input is small and does not increase with increasing number of executions. The third and the fourth inputs to Algorithm 2 reflects this design, where n determines the function pairs in *FPairs*.

By adopting thread-local maps and recording only *n-frequent* function pairs, the only side effect is that *AtexRace* may sample function pairs that have been sampled in the same execution due to the content difference of different threads within the same execution. This may incur unnecessary overhead. However, it produces no bad result on the data race coverage as sampling the same functions more than one time also increases the probability to detect those missed data races (see the first kind of limitations in Section 4.3).

4.5 *AtexRace* on Example Program

In this section, we use the running example in Figure 2 to illustrate how *AtexRace* sampling its executions in Figure 4. Initially, both functions f_1 and f_3 are sampled as the input *FPairs* are empty. Such sampling continues until in each thread the recorded functions pairs contain $\langle f_1, f_3 \rangle$. Probably[1], after a certain number of calls to both functions, *AtexRace* stops continuous sampling of f_1 and f_3 because $\langle f_1, f_3 \rangle$ is hot. Of course, in our algorithm, functions in a hot pair still have chances to be sampled due to our burst sampling strategy.

Next, suppose thread t_1 calls f_2 for the first time while t_2 is executing f_3. Because pair $\langle f_2, f_3 \rangle$ is cold, *AtexRace* restarts to sample function f_2. Of course, f_3 is sampled as well. Similarly, *AtexRace* restarts to sample function f_1 if functions f_1 and f_4 are executed at the same time. On the other hand, if it is f_2 and f_4 that are executed at the same time, neither f_1 nor f_3 is sampled.

Hence, for cases (c) and (d), *AtexRace* has a larger probability to detect the two races that are probably missed by *LiteRace*. However, for cases (a) and (b), although there is no race, *AtexRace* still samples the first calls to function f_3 and f_4. In the subsequent execution, after functions f_3 and f_4 are called for several times, *AtexRace* stops the continuous sampling of the two functions.

After one execution of the example program, *AtexRace* records the observed function pairs (probably the four pairs: $\langle f_1, f_3 \rangle$,$\langle f_1, f_4 \rangle$,$\langle f_2, f_3 \rangle$, and $\langle f_2, f_4 \rangle$). If the program is executed again, *AtexRace* may not continuously sample the function pairs already collected. Hence, the total overhead to detect data race can be reduced, not only within the same execution but also across different executions of the same program.

[1] In this section, we frequently used the word "probably" because the execution of multiple threads is undetermined. E.g., we say that, if functions f_1 and f_3 are called multiple times (as shown in Figure 2), most of their executions are simultaneous. But in theory, it is possible that all executions of function f_1 are executed before any execution of function f_3. Or given two threads that can be executed in parallel, there are executions where they can be sequentially executed.

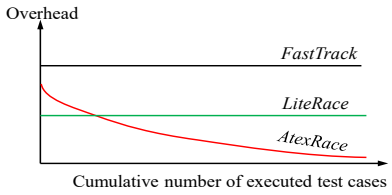


Figure 7. Ideal overhead changes with increasing executions.

4.6 Discussion on *AtexRace*

We aim to reduce race detection overhead without sacrificing race detection capability when there are many test cases. *AtexRace* does not target a single execution as one of our innovations is to record the recently observed function pairs and skips their sampling in subsequent executions. Hence, on a small number of executions, it may initially incur larger overhead than that by *FastTrack* and *LiteRace* (see **Figure 8** (a) and Figure 10 in our experiment). *AtexRace* is more suitable for programs (e.g., industrial programs) that are tested against a large number of test cases. Of course, as a dynamic sampling approach, it also reports false negatives.

Figure 7 shows the ideal scenario of *AtexRace*. Initially, *AtexRace* may incur higher overhead than *LiteRace* or even *FastTrack*. However, with increasing number of executions, *AtexRace* gradually incurs lower overhead.

5. EXPERIMENTS

This section presents the evaluation on *AtexRace*. We compared it with *LiteRace* and *FastTrack*. Because *FastTrack* is one of the fastest and most widely used tools in this category. It fully detects data races and can be considered as a sampling tool with a rate of 100%. And *LiteRace* is the state-of-the-art in-house sampling tool. Both are representative and well-known.

5.1 Implementation and Benchmarks

Implementation. We have implemented *AtexRace*, *FastTrack* and *LiteRace* on top of Pintool [11][32], a widely used binary instrumentation framework. Our implementation targets multi-threaded programs with Pthread library on Linux 32 system. Note that, Pintool runs like a virtual machine [32] and incurs large overhead. A better implementation can be done as the original *LiteRace* implementation [34] (i.e., to integrate sampling tools into the program under testing at compilation time).

Function encoding. On Linux platform, Pintool modes each program as Image that contains Sections and each section con-

Table 1. The statistics of Parsec benchmark and its overall results.

Benchmarks	Size (SLOC)	Time (seconds)				Overhead (%)			# of Unique Races		
		Pin	*FT*	*LR*	*AR*	*FT*	*LR*	*AR*	*FT*	*LR*	*AR*
Blackscholes	1,380	1048.53	1612.63	1652.39	1604.82	53.80%	57.59%	53.05%	0	0	0
Bodytrack	16,479	633.396	884.281	718.466	686.812	39.61%	13.43%	8.43%	14	14	32
Canneal	2,847	3314.22	7237.03	3640.18	4947.22	118.36%	9.84%	49.27%	2	2	2
Freqmine	2,192	2812.04	6451.47	4398.5	3317.82	129.42%	56.42%	17.99%	160	263	280
Streamcluster	1,795	111.626	135.767	131.325	131.915	21.63%	17.65%	18.18%	8	8	8
					Avg.:	72.56%	30.98%	29.38%	**Sum**: 190	291	326

sists of multiple Routines (or functions). We use a 32-bit integer to encode a routine. The first 6 bits are used as the Image identifier and the remaining 26 bits are used as routines identifier per image. This encoding allows at most $2^6 = 64$ images and $2^{26} \approx 64 \times 10^6$ routines per-image, which is enough in practice.

Benchmarks. We choose the Parsec benchmark suite 3.1 [2] to evaluate the race detectors. The suite consists of 13 benchmarks. After eliminating the benchmarks that are not multi-threaded or cannot be compiled under the Pin environment, we obtain five benchmarks: *Blackscholes*, *Bodytrack*, *Canneal*, *Freqmine*, and *Streamcluster*. In our experiments, we run each benchmark from Parsec for 100 times to collect their results.

Table 1 gives the source code size (SLOC) of the five benchmarks. It can be observed that the lines of code range from 1.3K to 16K. To further evaluate the performance of *AtexRace*, we select the MySQL database server (v6.0.4), a widely used real-world program. The version we use, mysql-6.0.4-alpha, has 1,114,980 lines of code. Among the 399 test cases that comes with its distribution, 223 of them can be successfully executed in the Pin environment. We run all the 223 test cases in our experiment.

5.2 Experimental Setup

Our experiments were performed on a workstation (ThinkPad W540) with an i7-4710MQ CPU (four cores), 16G memory, and 250G SSD. The workstation was installed with Ubuntu 14.04 x86 system. For *AtexRace*, we set its sampling rate and the value *n* (determining *n-frequent* function pairs) to be 10/100 and 2, respectively. For *LiteRace*, we adopt the fixed thread-local sampling configuration as defined in previous work [34].

5.3 Result Analysis on Parsec Benchmark Suite

5.3.1 Overall Results. For all techniques, Table 1 gives the time of the executions spent by Pin and the three tools of the benchmarks, the overhead of the race detectors compared to the time consumed by Pin framework, and the number of unique races (i.e., the number of variables in the source code) detected by each tool.

As expected, both *LiteRace* and *AtexRace* are much faster than *FastTrack* by reducing about 40% overhead based on Pin. It can also be observed that *LiteRace* and *AtexRace* incurred almost the same average overhead. On race detection capability, both *LiteRace* and *AtexRace* outperform *FastTrack*. At first glance, the results are surprising. However, it is known that sampling perturbs thread scheduling so a race detector with sampling runs different executions with the one without sampling, even under the same test case. Such phenomenon is previously observed [15]. Table 1 shows that *LiteRace* detects 53% more unique races than *FastTrack*, all of the additional races are from the single benchmark *Freqmine*. *AtexRace* detects 12% more unique races than *LiteRace*.

The above results indicate that *AtexRace* detect the most number of races at a cost almost the same as *LiteRace*. However, among the five benchmarks *LiteRace* is better in only two of them. On the other hand, the three benchmarks seem to have very few races (or even no races) so none of the race detectors can discover more races. When there are more races, the benefit of *AtexRace* seems obvious. Since these relatively small benchmarks do not give a doubtless evaluation of *AtexRace*, we further evaluate our approach on a large real-world database server MySQL. But before we present our empirical study on MySQL,

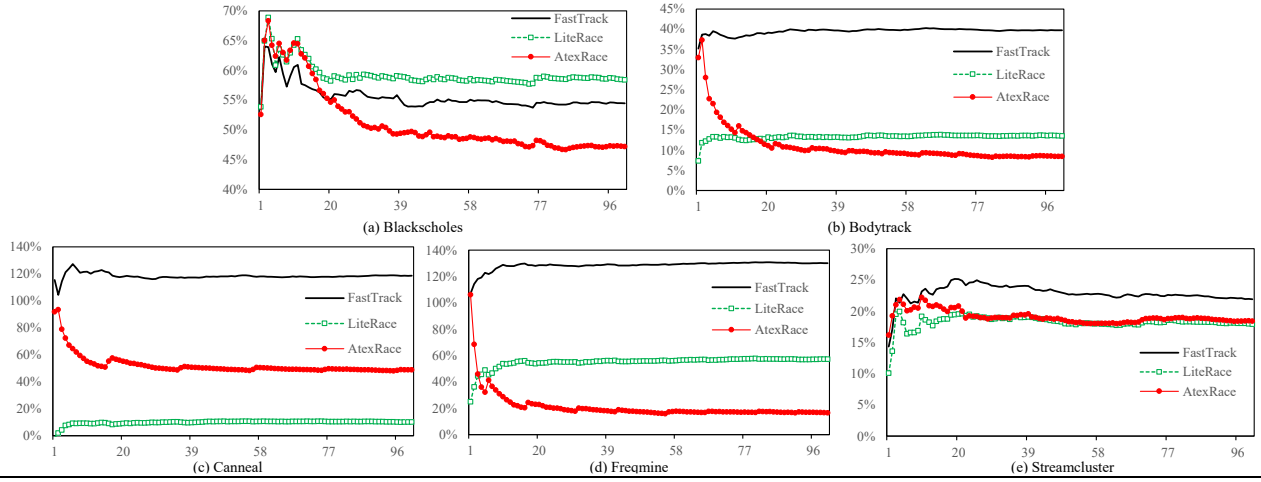
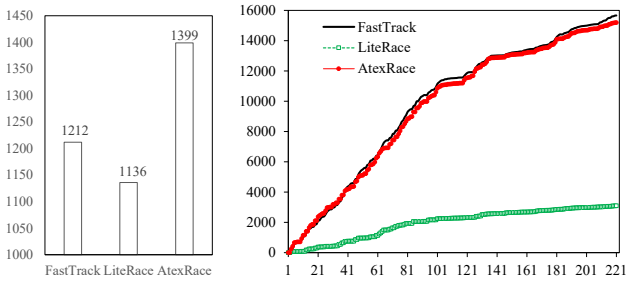


Figure 8. The trend of overhead with increasing number of executions.



(a) # unique races. (b) # dynamic races

Figure 9. The number of races detected by three.

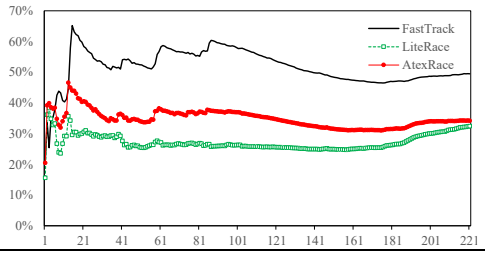


Figure 10. The cumulative overhead of three techniques with increasing executions.

we use Parsec to illustrate the advantage of cross-execution sampling of *AtexRace*.

5.3.2 Overhead Trend Across Executions. One of key features of *AtexRace* is its cross-execution sampling, which may result in lower overhead with increasing number of executions. In **Figure 8**, we show how the overhead changes with increasing number of executions for three techniques on the five benchmarks. In each subfigure, the x-axis shows the number of executions; and the *y*-axis shows the overhead incurred by three techniques on each execution. The cumulative overhead on the *i*-th execution is calculated by the following formula:

$$Overhead(i) = \frac{\Sigma(T_{tool}(i)) - \Sigma(T_{prog}(i))}{\Sigma(T_{prog}(i))} \times 100\% \quad \text{(Eq. 1)}$$

where $T_{tool}(i)$ represents the execution time under a *tool* on the *i*-th execution, and $T_{prog}(i)$ represents the native program execution time under Pin.

From **Figure 8**, we see that, overall, *FastTrack* and *LiteRace* incur almost the same overhead across executions (i.e., nearly a horizontal line). Except the first benchmark, *LiteRace*'s overhead is much lower than that by *FastTrack*, which is expected due to the sampling of *LiteRace*. However, on *Blackscholes*, *LiteRace* incurs larger overhead than that by *FastTrack*. We have performed several additional experiments and confirmed the results.

For *AtexRace*, overall, its overhead decreases with increasing number of executions, although the trend is less obvious in *Steamcluster*. This is consistent with our theoretical analysis in Section 4.6. It can also be observed that, with increasing number of executions, *AtexRace*'s performance becomes the best on three benchmarks (i.e., the subfigure (a), (b), and (d)). However, the overhead reduction reaches a plateau after a certain number of executions. This is not surprising because according to Section 5.3.3 the number of recorded function pairs barely increases.

5.3.3 Number of Function Pairs. As a Parsec benchmark is repeatedly executed under the same input, there is no obvious increase in the number of function pairs with more executions. After 100 executions, the number of functions pairs of the five benchmarks are 9, 409, 46, 250, and 23. If we store 2-*frequent* pairs only, the number of function pairs after 100 executions are 8, 227, 32, 232 and 16.

5.4 Result Analysis on MySQL

MySQL has one million lines of code. We run it against 223 test cases in the default order of the test script "*mysql-test-run*".

5.4.1 Number of Detected Races. Figure 9(a) gives the number of unique races that are detected by *FastTrack*, *LiteRace* and *AtexRace* after 223 executions of MySQL. Not surprisingly, compared with *LiteRace*, *AtexRace* detects 23% more unique races. What we have not expected is that *AtexRace* detects even 15% more unique races than *FastTrack*. Of course, as explained in Section 5.3.1, this is possible because sampling perturbs thread scheduling. However, we would like to have a clearer picture of race detecting capability. Thus, we collect data on all the races, not just unique races, that are detected by the three tools. Although in theory unique races are more interesting, in practice the number of total races is helpful to debugging because they can illustrate different scenarios how a race occurs. Detecting the same traces multiple times is also a good indicator of a race detector's capability.

The results of total races are illustrated in Figure 9(b). The number of total races is significantly more than the number of unique races. It can be observed that *FastTrack* detects the most races, but *AtexRace* is a very close second. *LiteRace*, on the other hand, detects significantly fewer number of races than the other two.

5.4.2 Overhead. Figure 10 depicts how the overhead (*y*-axis) changes across 223 executions (*x*-axis). Unlike benchmarks from Parsec where all repeated executions are conducted against the same test cases, each of the 223 MySQL executions is conducted against a different test case. Therefore, on MySQL, *FastTrack* (as well and *LiteRace* and *AtexRace*) may incur different overhead on different executions. The formula to calculate the cumulative overhead of the first *i* executions is the same as that on Parsec (i.e., Eq. 1).

The results shown in Figure 10 are as expected, where *FastTrack* incurs the largest overhead over native execution on Pin and *LiteRace* incurs the smallest. Although *AtexRace*'s overhead is larger than *LiteRace*'s, the gap is gradually shrinking. At the end of all 223 executions, *AtexRace* incurs almost the same overhead as that by *LiteRace*. Given more test cases, *AtexRace* may have a chance to incur less overhead than *LiteRace*.

Considering both Figure 9 and Figure 10, our experiments confirm that *AtexRace* achieves a sweet spot between *LiteRace* and *FastTrack*, by detecting almost the same number of races as *FastTrack* at a cost almost the same as *LiteRace*.

5.4.3 Number of Function Pairs. AtexRace does not record all observed function pairs but only keeps recently observed ones to avoid potentially unlimited increase on the number of function pairs. Figure 11 shows a comparison on the cumulative number of function pairs (*y*-axis) with the increasing number of executions (up to 223). The two lines represent the data by recording all observed ones ("*All Pairs*") and recording recently observed

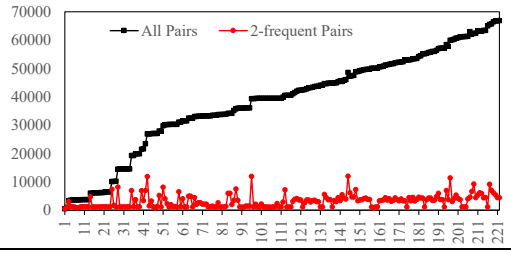


Figure 11. The cumulative number of function pairs.

ones ("*2-frequent Pairs*"), respectively.

We observe that, with increasing number of executions, the number of all function pairs also increases. After 223 executions, the number of observed function pairs is nearly 70,000. If we keep all these function pairs, a large overhead on querying is inevitable, which may eventually offset the benefit of sampling. This is the reason we only rely on the recently observed function pairs. From Figure 11, we see that this strategy is effective, as over all the 223 executions, the numbers of the 2-frequent function pairs are almost always below 10,000 (with only six exceptions). And on 198 out of 223 (~89%) executions, there are less than 5,000 function pairs. On average, there are 3,320 function pairs on each execution. Our experiments are all performed with 2-frequent function pairs and the data confirm its effectiveness.

6. RELATED WORK

Data races [10][16] are extremely difficult to be found and reproduced. Both static techniques [25] [37][42][51] and dynamic techniques [16][41][44][47][54] aim to detect data races. Static ones [51][42] can analyse the source code of a whole program; however, due to lack of runtime information, static approaches can easily report many false positives. Dynamic ones analyse concrete executions to detect data races according to some rules (e.g., the lockset discipline [44][46][56] and the happens-before relation [6][16][41][43][50][52]). Although dynamic techniques are relatively precise, they incur heavy overhead.

We have heavily discussed sampling approaches on data race detection. *CRSampler* [12] also targets on sampling but its main purpose is at user site. It is based on hardware breakpoints and clock races to detect data races. *DataCollider* [14] purely relies on hardware breakpoints to detect those occurred data race by suspending threads. *AtexRace* aims at in-house sampling.

To explore all possible executions is one direction to find concurrency bugs (e.g., Model checking [53][35]). However, it is usually impossible to explore all the interleaving although they may achieve certain coverage [28]. Practically, enumerating each schedule is not practical for large-scale real-world programs, even with reduction techniques [18].

Therefore, to explore a small portion of interleaving space that are error prone is also one direction. *Chess* [35] sets a heuristic bound on the number of pre-emptions to explore the schedules. Also, although systematic approaches avoid executing previously explored schedules, they usually incur large overheads and fail to scale up to handle long running programs. For example, *Maple* [55] is a coverage-driven [8][19] tool to mine thread interleaving so as to expose unknown concurrency bugs. *PCT* [9][36] randomly schedules a program to expose concurrency bugs, which also requires large number of executions. However, it is difficult to apply these techniques to large-scale programs such as MySQL.

Other works aim to firstly predict a set of potential data races and then to verify them. *RVPredict* [22] achieves a strictly higher coverage than HBR based detectors. It firstly predicts a set of potential races and then relies on a number of production executions to check against each predicted race. *Racageddon* [15] aims to solve races that could be predicted in one execution but require different inputs. It still needs a larger number of executions to check against each predicted race [39][45]. Both *RVPredict* and *Racageddon* have to solve scheduling constraints for each predicted race, which may fail. *RaceMob* [26] statically detects data race warnings and distributes them to a large number of users to validate real races. In such a run, the schedules are guided by the set of data race warnings to trigger real data races. This kind of approach is able to confirm real races but cannot eliminate false positives. Besides, it may miss real races if such races are not predicted in the (static) prediction phase.

DrFinder [10] tries to predict the happens-before relation to further expose races hidden by the happens-before relation. It dynamically predicts and tries to reverse happens-before relations from observed executions. However, its active scheduling is also heavy (e.g., about 400% [10] for Java programs).

CCI [24] proposes cross-thread sampling strategies to find causes of concurrency bugs based on randomized sampling. Unlike race sampling techniques (e.g., *CRSampler*, *DataCollider*, *Pacer*, and *LiteRace*), *CCI* focuses on failure diagnosis. However, *CCI* may cause heavy overhead (e.g., up to 900% [24]) although it targets on lightweight sampling. *Carisma* [58] improves *Pacer* by further sampling memory locations allocated at the same program location for Java. *Valor* [4] infers data races by detecting region confilt, which has good performance compared with *FastTrack*.

Bedides multithreaded programs, data race may also exist in other kinds of pgorams, such even-driven programs such as android applications [33][21][20], concurrent library invocations [13], and modified program codes [57]. *AtexRace* could also be adapted to detect these races. We leave it as future work.

7. CONCLUSION

We have proposed a new cross-thread and cross-execution sampling approach to achieve both high race detection rate and high efficiency. By adopting several novel designs, our prototype *AtexRace* shows its potential to replace *FastTrack* and *LiteRace*. This is confirmed by the experiments with benchmarks obtained from both Parsec benchmark suite and a real-world large-scale MySQL database.

ACKNOWLEDGEMENT

We thank anonymous reviewers for their invaluable comments and suggestions on improving this work. This work is supported in part by National Natural Science Foundation of China (NSFC) (61502465, 61472318 and 61632015), National 973 program of China (2014CB340702), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (2017151), and the National Science Foundation (NSF) (DGE-1522883).

REFERENCE

[1] B. Alpern, C.R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J.J. Barton, S.F. Hummel, J.C. Sheperd, and M. Mergen. Implementing jalapeño in Java. In *Proc. OOPSLA*, 314–324, 1999.

[2] C. Bienia. Ph.D. Thesis: Benchmarking modern multiprocessors. Princeton University, January 2011.

[3] S. Biswas, M. Cao, M. Zhang, M.D. Bond, and B.P. Wook. Lightweight data race detection for production runs. In *Proc. CC*, 11 – 21, 2017.

[4] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: Efficient, Software-Only Region Conflict Exceptions. In *Proc. OOPSLA*, 241–259, 2015.

[5] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The Dacapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*, 169–190, 2006.

[6] E. Bodden and K. Havelund. Racer: effective race detection using AspectJ. In *Proc. ISSTA*, 155–166, 2008.

[7] M.D. Bond, K. E. Coons and K. S. Mckinley. PACER: Proportional detection of data races. In *Proc. PLDI*, 255–268, 2010.

[8] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proc. PPoPP*, 206–212, 2005.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proc. ASPLOS*, 167–178, 2010.

[10] Y. Cai and L. Cao. Effective and precise dynamic detection of hidden races for Java programs. In *Proc. ESEC/FSE*, 450–461, 2015.

[11] Y. Cai and W.K. Chan. LOFT: Redundant synchronization event removal for data race Detection. in *Proc. ISSRE*, 160–169, 2011.

[12] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In Proc. FSE, 810–821, 2016.

[13] D. Dimitro, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In Proc. PLDI, 305–315, 2014.

[14] J. Erickson, M. Musuvathi, S. Burckhardt and K. Olynyk. Effective data-race detection for the kernel. In Proc. *OSDI*, 1–6, 2010.

[15] M. Eslamimehr and J. Palsberg. Race directed scheduling of concurrent programs. In *Proc. PPoPP*, 301–314, 2014.

[16] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proc. PLDI*, 121–133, 2009.

[17] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proc. PASTE*, 1–8, 2010.

[18] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL*, 110–121, 2005.

[19] S. Hong, J. Ahn, S. Park, M. Kim, and M.J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proc. ISSTA*, 210–220, 2012.

[20] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side java script web applications. In *Proc. ICST*, 61–70, 2014.

[21] C. Hsiao, Y. Yu, S. Narayanasamy, Z. Kong, C.L. Pereira, G.A. Pokam, P.M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In Proc. PLDI, 326–336, 2014.

[22] J. Huang, P.O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proc. PLDI*, 337–348, 2014.

[23] J. Jackson. Nasdaq's Facebook glitch came from 'race conditions', May 21 2012. http://www.computerworld.com/article/2504676/financial-it/nasdaq-s-facebook-glitch-came-from--race-conditions-.html, last visited on March 2016.

[24] G. Jin, A. Thakur, B. Liblit and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proc. OOPSLA*, 241–225, 2010.

[25] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proc. CAV*, 226–239, 2007.

[26] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced data race detection. In *Proc. SOSP*, 406–422, 2013.

[27] L. Lamport. Time, clocks, and the ordering of events. Communications of the ACM 21(7):558–565, 1978.

[28] Z. Letko, T. Vojnar, and B. K˘rena. Coverage metrics for saturation-based and search-based testing of concurrent software. In *Proc. RV*, 177–192, 2011.

[29] N.G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7), 18–41, 1993.

[30] S. Lu, S. Park, E. Seo, and Y.Y. Zhou, Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, 329–339, 2008.

[31] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Proc. ASPLOS*, 39–50. 2013.

[32] 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 *Proc. PLDI*, 191–200, 2005.

[33] P. Maiya, a. Kanade, and R. Majumdar. Race detection for Android applications. In Proc. PLDI, 316–325, 2014.

[34] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *Proc. PLDI*, 134–143, 2009.

[35] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proc. OSDI*, 267–280 2008.

[36] S. Nagarakatte, S. Burckhardt, M. M.K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proc. PLDI*, 2012, 543–554, 2012.

[37] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. PLDI*, 308–319, 2006.

[38] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proc. PLDI*, 22–31, 2007.

[39] C.S. Park, K. Sen, P. Hargrove, and C. Iancu. Efficient data race detection for distributed memory parallel programs. In *Proc. SC*, 2011.

[40] K. Poulsen. Software bug contributed to blackout. http://www.securityfocus.com/news/8016, Feb. 2004.

[41] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. PPoPP*, 179–190, 2003.

[42] P. Pratikakis, J.S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proc. PLDI*, 320–331, 2006.

[43] A.K. Rajagopalan and J. Huang. RDIT: race detection from incomplete traces. In *Proc. ESEC/FSE*, 914 - 917, 2015.

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

[45] K. Sen. Race Directed Random Testing of Concurrent Programs. In *Proc. PLDI*, 11–21, 2008.

[46] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proc. WBIA*, 62–71, 2009.

[47] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proc. POPL*, 387–400, 2012.

[48] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proc. FSE*, 37–46, 2010.

[49] Microsoft. Thread execution blocks. http://msdn.microsoft.com/enus/library/ms686708.aspx

[50] K. Vineet and C. Wang. Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In *Proc. CAV*, 434–449, 2010.

[51] J.W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Proc. FSE*, 205–214, 2007.

[52] C. Wang, K. Hoang. Precisely Deciding Control State Reachability in Concurrent Traces with Limited Observability. In *Proc. VMCAI*, 376–394, 2014.

[53] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proc. ICSE*, 221–230, 2011.

[54] X.W. Xie and J.L. Xue. Acculock: Accurate and Efficient detection of data races. In *Proc. CGO*, 201–212, 2011.

[55] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Proc. OOPSLA*, 485–502, 2012.

[56] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proc. SOSP*, 221–234, 2005.

[57] T. Yu, W. Srisa-an, and G. Rothermel. SimRT: An automated framework to support regression testing for data races. In Proc. ICSE, 48–59, 2014.

[58] K. Zhai, B.N. Xu, W.K. Chan, and T.H. Tse. CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In *Proc. ISSTA*, 221–231, 2012.

[59] W. Zhang, M. d. Kruijf, A. Li, S. Lu and K. Sankaralingam. ConAir: featherweight concurrency bug recovery via single-threaded idempotent execution. In *Proc. ASPLOS*, 113–126. 2013.