SimpleLock: Fast and Accurate Hybrid Data Race Detector

Misun Yu[*] [†], Sang-Kyung Yoo[†], Doo-Whan Bae[†]
[*]Electronics and Telecommunications Research Institute
msyu@etri.re.kr
[†]KAIST
{msyu, skyoo94, bae}@kaist.ac.kr

***Abstract*—Data races are one of the major causes of concurrency bugs in multithreaded programs, but they are hard to find due to nondeterministic thread scheduling. Data race detectors are essential tools that help long-suffering programmers to locate data races in multithreaded programs. One type of detectors precisely detects data races but is sensitive to thread scheduling, whereas another type is less sensitive to thread scheduling but reports a considerable number of false positives.**

In this paper, we propose a new dynamic data race detector called SimpleLock that accurately detects data races in a scheduling insensitive manner with low execution overhead. We reduce execution overhead by using two assumptions. The first is that most data races are caused by the accessing of shared variables without locks. The second is that two accesses that cause a data race have not a long distance between them in an execution trace.

The results of experiments conducted on the RoadRunner framework confirm that these assumptions are valid and that our SimpleLock detector can efficiently and accurately detect real and potential data races in one execution trace. The results also indicate that the execution overhead of SimpleLock is not much higher than that of FastTrack, the fastest happens-before race detector.

I. Introduction

As multicore computers become more popular and widespread, multithreading is increasing in importance because it can maximize the computing power of multicore processors. However, finding and resolving concurrency bugs caused by concurrent access to shared memory is very difficult due to nondeterministic thread interleaving. Data race is one of the main causes of concurrency bugs in multithreaded programs. A data race occurs in a multithreaded program when at least two different threads access the same memory location without an ordering constraint enforced between the accesses, such that at least one of the accesses is a write [1]. Although data races do not always cause failures in programs, they can directly or indirectly lead to incorrect results or crashes in the future. Therefore, automatic race detection tools that can accurately and quickly detect data races are essential.

Recent race detection tools are based on dynamic detection techniques that use the information generated during program execution. Dynamic data race detection techniques categorized into three types: Lockset, happens-before (HB) and hybrid. Lockset-based detectors, which originated from Eraser [2], are insensitive to thread scheduling but generate false positives. In contrast, happens-before detectors, which are based on vector clocks [3]–[7], generate no false positives but are sensitive to thread scheduling. High overhead time was once a weakness of vector-clock detection techniques as well; however, FastTrack [3] reduced the time to nearly the same level as that of the Lockset technique. Starting from these facts, recent hybrid detectors [8], [9] try to find more data races than happens-before detectors in one execution trace with low execution overhead using the two detection techniques. Although these hybrid techniques detect more data races than their predecessors, they generate false positives or their overhead are too high to be used frequently.

In this paper, we propose a fast hybrid data race detection algorithm that detects real and potential data races in one execution trace. Previous Lockset-based detectors performed intersection operations between two sets of locks to find accesses not protected by consistent locks - more specifically, those protected by different locks as well as those not protected by any lock. We remove the intersection operation based on the assumption that most data races are caused by accesses to shared variables that are not protected by locks. In addition, we significantly reduce execution overhead by utilizing the fact that two accesses making a data race have not a long distance between them in an execution trace.

The remainder of this paper is organized as follows. Section II gives an overview of our detection technique. Section III reviews preliminary concepts associated with the two race detection algorithms that form the basis of our proposed algorithms. Section IV describes our detection technique in detail, and Section V presents our experimental results. Section VI discusses related work. Section VII summarizes and concludes this paper.

II. Detection Technique Overview

A. Motivation and Solution

The Lockset technique effectively detects data races caused by inconsistent lock protection due to its insensitivity to thread scheduling. However, it generates many false positives. Conversely, FastTrack is capable of precisely detecting all data races in one execution trace, but due to its sensitivity to thread scheduling, it can miss possible data races that can be detected by scheduling insensitive detectors. The design objectives of SimpleLock are as follows:

- To detect real and potential data races that can be missed by happens-before detectors in one execution trace, and to provide coverage comparable to that of state-of-the-art accurate hybrid race detectors.

- To provide execution overhead comparable to that of FastTrack to facilitate frequent usage during the software development process.

To achieve these objectives, we analyzed the cause of false positives by Lockset and missing potential data races by happens-before detectors. Main reason for false positives by Lockset is its neglect of the deterministic execution order between two threads by events such as the starting and resuming of thread execution. As a result, it cannot exactly identify concurrently executable accesses by different threads.

Figure 1 shows the execution trace of two threads. The accesses from 1.3 to 1.6 by t1 are executed nondeterministically with the accesses from 2.1 to 2.3 by t2. The access at 1.1 by t1 before t2.start() cannot be interleaved with all the accesses of t2, so write access at 1.1 must be excluded from the Lockset operation.

Of note is the fact is that VC-based detectors cannot detect any data races in this execution trace despite the fact that there are potential data races between 1.3 and 2.2. If write access in line 1.3 is executed after 1.6, happens-before detectors report a data race. Due to their nature, VC-based detectors may miss the clock information for accesses that have executed before synchronization operations by lock acquire and release, such as the write access at line 1.3 in this example.

Thread $t1$	**Thread** $t2$
1.1 write x	
1.2 t2.start()	
1.3 **write x**	
1.4 **lock** l_1	
1.5 **write x**	
1.6 **unlock** l_1	
	2.1 **lock** l_1
	2.2 **read x**
	2.3 **unlock** l_1

Figure 1. An example of thread interleaving

The main reason for this is that happens-before detectors do not discriminate among various different types of synchronization events such as the following:

- Synchronization events that enforce deterministic order among threads, such as thread start and awaking threads (start(), and notify()/notifyAll() calls in Java).
- Synchronization events that do not enforce deterministic order among threads, such as lock acquire and lock release.

On the basis of this analysis, we come up with a Lockset-based technique that discriminates the above two different types of events to reduce false positives. We adopted a similar approach to Acculock-Multi [9], a hybrid race detector that accurately detects potential data races. However, SimpleLock incurs low execution overhead while maintaining virtually the same detection coverage.

B. Contributions

Our contributions are as follows:

- We reduce execution overhead by using two assumptions: 1) At least one of the accesses causing a data race is not protected by a lock, and 2) the distance between the accesses is not long.
- We prove empirically that our two assumptions are valid.
- We propose a new detection algorithm that accurately detects more data races than FastTrack, while incurring a lower execution overhead than a conventional state-of-the-art hybrid detector.
- We implement our proposed detector, SimpleLock, which uses our proposed algorithm, and two state-of-the-art race detectors - FastTrack and AccuLock-Multi on Road-Runner [10] framework, and compare their performance in terms of coverage, accuracy and execution overhead using 11 benchmark programs from the Parallel Java Benchmark suite (PJBench) [11].

III. Background

A. Vector Clock, Djit+ and FastTrack

In VC-based detectors, a vector clock, [12] VC: Tid→Nat, records the clocks of all threads in a program. Two vectors, V_1 and V_2, are partially-ordered ($\sqsubseteq$) if V_1 is smaller than or equal to V_2 in a point-wise manner. Their join operation ($\sqcup$) is used to get a point-wise maximum:

$$\begin{array}{lcl} V_1 \sqsubseteq V_2 & iff & \forall t.V_1(t) \leq V_2(t) \\ V_1 \sqcup V_2 & = & \lambda t.max(V_1(t), \quad V_2(t)) \end{array}$$

In Djit+ [4], each thread maintains vector clock C_t such that, for any thread u, the clock entry $C_t(u)$ records the clock for the last operation of thread u that happens before the current operation of thread t. Additionally, the algorithm maintains a vector clock L_m for each lock m. The clock of a thread is updated when a synchronization action is performed using L_m. For example, if a thread t subsequently acquires m, the algorithm updates C_t to $C_t \sqcup L_m$, since subsequent operations by thread t would now happen after that release operation. In addition, $C_t[t]$ is incremented, and $L_m[t]$ is updated to $C_t[t]$ when the release operation is performed.

Djit+ keeps two vector clocks, W_x and R_x, for read/write access to a shared variable x. When a read/write access to x occurs during program execution, a check for data races on that variable is done. $R_x(t)$ and $W_x(t)$ record the clocks of the last read and the last write for shared variable x by thread t. If the current access is a read from x and $W_x(t)$ is partially-ordered with the vector clock of the current active thread ($W_x(t) \sqsubseteq C_u$), the read access is not a data race. If the current access is a write from x and $R_x(t)$ and $W_x(t)$ are partially-ordered with the vector clock of the current active thread ($W_x(t) \sqsubseteq C_u$ and $R_x \sqsubseteq C_u$), the write access is not a data race.

The problem with Djit+ and other VC-based race detectors is high execution overhead. All vector clock operations and memory space increase in proportion to the number of threads, that is, VC-based data race detectors require O(n) times. FastTrack reduced most O(n) VC operations to O(1) by introducing epoch-VC comparison ($\preceq$).

Algorithm 1 shows the FastTrack notation and algorithm. We introduces a funtion $E(t)$ that returns current epoch of thread t. FastTrack exploits following observations: (1) all writes to x are totally ordered by the happens-before relation (assuming no races are detected so far), and so it records the thread identifier (tid) and its clock only about the very last write to x (2) Read operations on thread-local and lock-protected data are also totally ordered (assuming no races have been detected) and so FastTrack records only the tid and the clock of the last read to the data. FastTrack uses the new representation called epoch, which includes only a clock and tid. Epoch is used to record totally ordered last write and read.

Algorithm 1 FastTrack

```
    Acquire:
 1: $C_t \leftarrow C_t \sqcup L_t$

    Release:
 2: $L_t = C_t$
 3: $C_t[t] = C_t[t] + 1$

    Read:
 4: if $R_x \neq E(t)$ then
 5:     report a warning unless $W_x \preceq C_t$              ▷ write-read race
 6:     if $R_x = 1 \quad \wedge \quad R_x \preceq C_t$ then
 7:         $R_x \leftarrow E(t)$
 8:     else
 9:         $R_x \leftarrow C_t$
10:     end if
11: end if

    Write:
12: if $W_x \neq E(t)$ then
13:     report a warning unless $W_x \preceq C_t$              ▷ write-write race
14:     if $|R_x| = 0$ then
15:         report a warning unless $R_x \preceq C_t$          ▷ read-write race
16:     else
17:         report a warning unless $R_x \sqsubseteq C_t$     ▷ read-write race
18:     end if
19:     $R_x \leftarrow \emptyset$
20:     $W_x \leftarrow E(t)$
21: end if
```

B. LockSet and Eraser

The basic idea underlying Lockset is that locks should consistently protect each access by threads to every shared variable. Algorithm 2 shows the basic algorithm for Lockset. For each shared variable x, Lockset maintains a candidate set L_x containing all the locks that have consistently protected every access to x so far and, for each thread t, L_t holds the set of all the locks acquired by t at that time.

Algorithm 2 Lockset

```
1: For each $x$, initialize $L_x$ to the set of all locks.
2: On each access to $x$ by thread $t$,
3: $L_x \leftarrow L_x \cap L_t$
4: if $L_x = \emptyset$ then
5:     report a warning.
6: end if
```

Although Eraser tries to reduce false positives by using a state machine to process thread-local and read-shared data, it generates a lot of false positives as well.

IV. SIMPLELOCK

SimpleLock is a hybrid data race detector that combines happens-before and Lockset to accurately and quickly detect data races in one execution trace. In this paper, we refer to all real and potential data races simply as data races.

The first basic idea used by SimpleLock to reduce execution overhead is to find two unordered nondeterministic accesses by two different threads in which at least one is a write, and *at least one is not protected by a lock*. The second basic idea, which is used by SimpleLock for accurate detection with high coverage, is to use *fixed-size queues* to record a list of the minimum number of locks that have protected read/write access for each thread during the same clock period.

Like conventional happens-before detectors, each thread has a vector clock. The clock is updated when all the synchronization events that make the execution order of threads deterministic have occurred (Algorithms 3 and 4). In SimpleLock, lock acquire/release events do not change the clocks because they do not make any deterministic execution order between threads.

SimpleLock has several algorithm-variables. The first one is $LockCnt_t$, which keeps track of the number of locks held by the current thread t. $lockCount_t$ is initialized to zero and is incremented when t acquires a lock and decremented when t releases a lock (Algorithm 5 and 6). The other variables are $WC_x[MAX]$ and $RC_x[MAX]$, which keep a list of the minimum number of locks protecting the writes and reads, respectively, of a shared variable x during the same clock period in the form of <clock, lockCount> pairs. MAX is the maximum number of threads. $WC_x[t]$ and RC_x[t] are fixed length (Q_LEN) queues.

The queue provides two variables *"first"* and *"last"* to represent its first and last element. In addition, the queue provides a function called *"updateLast (p)"* to update the last element to p and *"addLast (p)"* to add new element p to the tail of the queue. The update policy for WC_x and RC_x and the detection algorithms are follows:

Read/Write Lock Count Update for Variables

$RC_x[t]$ keeps a list of the minimum number of locks that have protected read access to shared variable x by t during the same clock period. When a read access to x by a thread occurs, SimpleLock checks whether the clock of the last element of $RC_x[t]$ is the same as the current clock. Because the element is always added to the end of the queue and the clock is not decremented, SimpleLock only checks the last element. If the clock of the last element is the same as the current clock, SimpleLock updates its $lockCount$ to the minimum number between them. Otherwise, a new pair <$E(t)$, $lockCount_t$> is added to the tail of $RC_x[t]$.

$WC_x[t]$ keeps a list of the minimum number of locks that have protected write access to shared variable x by t during the same clock period. The update policy of $WC_x[t]$ is similar to that of $RC_x[t]$. When a write access to x by thread t occurs, SimpleLock checks whether the clock of the last element of $WC_x[t]$ is the same as the current clock. If the clock of the last element is the same as the current clock, SimpleLock updates its $lockCount$ to the minimum number between them. Otherwise, a new pair <$E(t)$, $lockCount_t$> is added to the tail of $WC_x[t]$.

In addition, after updating $WC_x[t]$, SimpleLock checks whether the clock of the last element of $RC_x[t]$ is the same as the current clock and whether its $lockCount$ is larger than the current one. If the check evaluates to true, SimpleLock removes the element to reduce the checking overhead.

Detecting Write-Read Races

When a read access by thread t occurs, SimpleLock searches for an element whose clock is not partially ordered with the clock of t in $WC_x[t']$ for each thread t'. If there is such an element and its $lockCount$ or $lockCount_t$ is zero, SimpleLock reports a warning, as indicated at line 15 of Algorithm 7.

Detecting Write-Write Races

When a write access by thread t occurs, SimpleLock searches for an element whose clock is not partially ordered with the clock of t in $WC_x[t']$ for each thread t'. If there is such an element and its $lockCount$ or $lockCount_t$ is zero, SimpleLock reports a warning, as indicated at line 19 of Algorithm 8.

Detecting Read-Write Races

When a write access by thread t occurs, SimpleLock searches for an element whose clock is not partially ordered with the clock of t in $RC_x[t']$ for each thread t'. If there is such an element and its $lockCount$ or $lockCount_t$ is zero, SimpleLock reports a warning, as indicated at line 26 of Algorithm 8.

Algorithm 3 Fork

```
1: $C_u \leftarrow C_u \cup C_t$                ▷ thread $t$ forks thread $u$
2: $C_u[t] \leftarrow C_u[t] + 1$
```

Algorithm 4 Notify/NotifyAll

```
1: $C_u \leftarrow C_u \cup C_t$      ▷ thread $t$ wakes up waiting thread(s) $u$
2: $C_u[t] \leftarrow C_u[t] + 1$
3: $C_t[t] \leftarrow C_t[t] + 1$
```

Algorithm 5 Acquire

```
1: $LockCnt_t = LockCnt_t + 1$
```

Algorithm 6 Release

```
1: $LockCnt_t = LockCnt_t - 1$
```

Missing Data Races

SimpleLock finds data races by checking for the existence of locks protecting an access. As a result, if all accesses have at least one lock protecting them, no warnings are reported. Figure 2 shows one such example. The number of locks protecting access is always greater than zero. Thus, the data race at line 2.2 is not reported. However, our experiments show that such cases are rare.

Algorithm 7 Read

```
1: $rdLast \leftarrow RC_x[t].last$
2: if $rdLast.epoch \neq E(t)$ then
3:     $RC_x[t].addLast(<E(t), lockCnt_t>)$
4:     if $|RC_x[t]| > Q\_LEN$ then
5:         remove $RC_x[t].first$
6:     end if
7: else
8:     if $rdLast.epoch = E(t) \wedge rdLast.lockCount > lockCnt_t$ then
9:         $RC_x[t].updateLast(<E(t), lockCnt_t>)$
10:    end if
11: end if
12: for all thread $t'$ in $WC_x$ do
13:     for all $<epoch, lockCnt> \in WC_x[t']$ do
14:         if $epoch \npreceq C_t[t] \wedge (lockCnt = 0 \vee lockCnt_t)$ then
15:             report a warning.                ▷ write-read race
16:         end if
17:     end for
18: end for
```

Algorithm 8 Write

```
1: $wrLast \leftarrow WC_x[t].last$
2: if $wrLast.epoch \neq E(t)$ then
3:     $WC_x[t].addLast(<E(t), lockCnt_t>)$
4:     if $|WC_x[t]| > Q\_LEN$ then
5:         remove $WC_x[t].first$
6:     end if
7: else
8:     if $(wrLast.epoch = E(t)) \wedge (wrLast.lockCount > lockCnt_t)$ then
9:         $WC_x[t].updateLast(<E(t), lockCnt_t>)$
10:    end if
11: end if
12: $rdLast \leftarrow RC_x[t].last$
13: if $rdLast.epoch = E(t)) \wedge (rdLast.lockCount > lockCnt_t)$ then
14:     remove $WC_x[t].last$
15: end if
16: for all thread $t'$ in $WC_x$ do
17:     for all $<epoch, lockCnt> \in WC_x[t']$ do
18:         if $epoch \npreceq C_t[t] \wedge (lockCnt = 0 \vee lockCnt_t)$ then
19:             report a warning.                ▷ write-write race
20:         end if
21:     end for
22: end for
23: for all thread $t'$ in $RC_x$ do
24:     for all $<epoch, lockCnt> \in RC_x[t']$ do
25:         if $epoch \npreceq C_t[t] \wedge (lockCnt = 0 \vee lockCnt_t)$ then
26:             report a warning.                ▷ read-write race
27:         end if
28:     end for
29: end for
```

Thread $t1$	**Thread** $t2$
1.1 lock l_1	
1.2 write x	
1.3 unlock l_1	
	2.1 lock l_2
	2.2 write x /* No Warning */
	2.3 unlock l_2

Figure 2. Missing Data Race

Table I
BENCHMARK CONFIGURATION

Name	Specification
avrora	Simulation and analysis tools for AVR micro-controllers
luindex	Text indexing tool
lusearch	Text search tool
sunflow	Implementation of a classic Cornell box rendering; a simple scene comprising two teapots and two glass spheres within an illuminated box
batik	SVG toolkit produced by the Apache foundation
cache4j	Fast thread-safe implementation for caching Java objects
elevator	Simple elevator simulation program without GUI
hedc	Web-crawler from ETH
sor	Implementation of successive over-relaxation
tsp	Implementation of Traveling Sales Person (TSP) algorithm
weblech	Web site download tool in Java

Table II
BENCHMARK RESULTS

Program	# of warnings				Execution overhead			
	FT	**SL**	SL'	**AL**	**FT**	**SL**	SL'	AL
avrora	3	**4**	4	**4**	**4.25**	**7.5**	110.03	251.87
luindex	1	**1**	1	**1**	**6.63**	**9.29**	10.3	11.99
lusearch	0	**2**	2	**2**	**4.49**	**9.19**	4.49	11.69
sunflow	5	**31**	31	**31**	**33.92**	**60.59**	74.73	106.63
batik	0	**0**	0	**0**	**2.82**	**3.55**	4.07	3.89
cache4j	2	**8**	8	**8**	**1.17**	**1.27**	1.43	1.27
elevator	0	**0**	0	**0**	**1.02**	**1.02**	1.02	1.02
hedc	4	**4**	4	**4**	**1.15**	**1.02**	1.01	1.13
sor	0	**0**	0	**0**	**11.62**	**10.12**	44.99	73.69
tsp	0	**0**	0	**0**	**21**	**15.42**	21.05	21.26
weblech	3	**4**	4	**4**	**1.28**	**1.19**	1.32	1.46
Average	2.7	**5.4**	5.4	**5.4**	**8.12**	**10.92**	24.95	44.17

V. EXPERIMENTAL EVALUATION

We evaluated the performance of SimpleLock in terms of execution overhead and the number of reported data races, and compared the results with those of two other state-of-the-art dynamic data race detectors using 11 Java benchmark programs. The two other detectors compared were FastTrack (FT) and Acculock-Multi (AL). FastTrack is the fastest data race detector for Java programs, while Acculock-Multi is an accurate hybrid detector. All detectors, including ours, were implemented on the RoadRunner framework, for fair comparison.

A. Platform

The evaluation was conducted on an Intel Core i-7-3770K (quad core) CPU 3.50 GHz machine with 16GB RAM and operating system 64-bit Ubuntu 12.4.

B. Benchmark Configuration

We selected 11 multithreaded programs from PJBench: avrora, luindex, lusearch, sunflow, batik, cache4j, elevator, hedc, sor, tsp and weblech. Table I gives a brief description of these programs.

Table III
THE NUMBER OF READ/WRITE ACCESSES

Program	Write ($\times 10^6$)	Read ($\times 10^6$)	Total ($\times 10^6$)
avrora	394	906	1300
sunflow	787	7000	7787
luindex	65	207	272
lusearch	228	929	1157
Average	*270*	*1593*	1863

C. Performance Comparison

Table II shows the number of reported data races and the execution overhead for SimpleLock and the two detectors. Execution overhead is the ratio of the instrumented running time to uninstrumented running time of the benchmark programs. We tested two SimpleLock variations, differentiated by Q_LEN: 1 (SL) and limitless (SL'). The default value of Q_LEN for SimpleLock is one. We performed 10 executions with the same input and calculated the average for each result. The number of reported races across consecutive runs was the same, and variability was less than 10%.

Race Warnings

The number of data races reported by the all hybrid detectors (SL, SL' and and AL) was always greater than or equal to that of FastTrack. Of note here is the fact that SimpleLock detected exactly the same data races as Acculock-Multi, which was a subset of those detected by FastTrack. Acculock-Multi is an accurate data race detector that removes false positives with multiple locking of shared variables. Therefore, SimpleLock provides broader coverage than FastTrack and its accuracy is comparable to that of AccuLock-Multi.

Because SimpleLock and Acculock-Multi detected the same data races, our first assumption, *"most of data races are caused by accesses to shared variables not protected by lock"*, is valid. Further, SL and SL' detected the same data races although they have different queue lengths (Q_LEN). This means that our second assumption, *"the distance between accesses causing a data race is not long"*, is also valid.

Performance Slowdown

In addition to the broad coverage provided by SimpleLock, its execution overhead was also remarkably lower than that of AccuLock-Multi and just 1.34 times higher than that of FastTrack on average.

In the case of sunflow, SimpleLock generated the highest overhead. The cause of this overhead is the number of accesses to $W_x[t]$ and $R_x[t]$. Table III shows the number of read/write accesses for four benchmark programs that had relatively high overhead. Among them, sunflow had the highest number of read/write accesses to shared variables.

We reduced the execution overhead using a fixed-sized queue, used for keeping a list of the minimum number of locks that have protected read/write access to each shared variable by a thread during the same clock period, and replaced lockset-intersection, used for checking for the existence of a lock.

VI. Related Work

Much research has been done on dynamic data race detection. Dynamic analysis techniques can be categorized into Lockset-based detectors, Happens-before detectors, and hybrid detectors. Lockset-based detectors are based on the Lockset algorithm of Eraser [2], which generates false positive but is not very sensitive to thread scheduling. Happens-before detectors detect data races by verifying happens-before relations represented using vector clocks (VCs).

Prior to the proposal of FastTrack [3], the cost of VC operations was very high with in the Djit+ detector [4], [13]; each VC required O(n) storage space and O(n) time. For this reason, happens-before detectors were not commonly used. Consequently, many detectors with lower performance overhead and lower false positive rates were proposed [14], [15]. The shortcoming of this strategy, however, is the possibility of introducing quite a number of false positives or false negatives.

LiteRace [16] is a precise and lightweight happens-before data race detector that utilizes an adaptive sampling technique that samples and analyzes only selected portions of a program's execution to reduce runtime overhead and minimize the number of missing data races.

FastTrack is a pure happens-before detector that has lowered the analysis overhead of almost all VC operations and the storage requirements from O(n)-time to O(1)-time. This improvement results from the replacement of heavyweight vector clocks with an adaptive lightweight representation called epoch that requires only constant-space and constant-time operations. With FastTrack, the execution overhead of happens-before detectors has become comparable to that of Eraser. Subsequent to the introduction of FastTrack, several techniques to lower runtime overhead and extend detection coverage has been proposed.

Pacer [17] uses a sampling technique like LiteRace, but is based on FastTrack. It detects any data races at a rate equal to the sampling rate by finding data races whose first access occurs during a global sampling period. It reduces runtime overhead by avoiding nearly all O(n) operations during non-sampling periods. However, like other sampling-based detectors, Pacer may miss some data races.

ACCULOCK [8] tries to make up for the scheduling sensitivity of happens-before detectors by combining Lockset and the epoch-based happens-before algorithm. It analyzes a program's execution by reasoning about the subset of happens-before relations observed with lock acquisition and releases excluded. Although ACCULOCK provides low overhead that is comparable with FastTrack, it is prone to reporting false positives because it cannot exactly trace accesses that are protected by multiple locks. In addition, it misses some races. ACUULOCK-Multi [9] solves these problems, but is three times slower than FastTrack on average for the 11 benchmark programs.

VII. Conclusion

One of the difficulties of finding data races comes from their irregular occurrence, even with the same input, and a lack of fast and accurate detection tools that have broad coverage. SimpleLock is a novel dynamic detector that provides broad coverage and a low execution overhead. SimpleLock's high performance stems from its use of the epoch-based representation of vector clock from FastTrack, the simplified Lockset that only checks for the existence of accesses protected by no locks and the fact that the distance between two accesses causing a data race is not long. We implemented and evaluated the effectiveness of SimpleLock only for Java programs. However, the proposed algorithm may be useful for C/C++ multithreaded program analysis.

References

[1] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992.

[2] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[3] C. Flanagan and S. N. Freund, "Fasttrack: efficient and precise dynamic race detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 121–133, Jun. 2009.

[4] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded c++ programs," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '03, 2003, pp. 179–190.

[5] M. Christiaens and K. De Bosschere, "Trade, a topological approach to on-the-fly race detection in java programs," in *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, ser. JVM'01, 2001, pp. 15–15.

[6] E. Schonberg, "On-the-fly detection of access anomalies," in *In Proceedings of the SIGPLAN 1989 Conference on Programming Language Design and Implementation*, 1998, pp. 285–297.

[7] S. L. Min and J.-D. Choi, "An efficient cache-based access anomaly detection scheme," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS IV, 1991, pp. 235–244.

[8] X. Xie and J. Xue, "Acculock: Accurate and efficient detection of data races," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11, 2011, pp. 201–212.

[9] J. Z. Xinwei Xie, Jingling Xue, "Acculock: accurate and efficient detection of data races." *Softw: Pract. Exper.*, 2012.

[10] C. Flanagan and S. N. Freund, "The roadrunner dynamic analysis framework for concurrent programs," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '10, 2010, pp. 1–8.

[11] "Parallel java benchmarks," https://code.google.com/p/pjbench.

[12] F. Mattern, "Virtual time and global states of distributed systems," in *Parallel and Distributed Algorithms*, 1989, pp. 215–226.

[13] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded c++ programs." *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[14] C. von Praun and T. R. Gross, "Object race detection," in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '01, 2001, pp. 70–82.

[15] H. Nishiyama, "Detecting data races using dynamic escape analysis based on read barrier," in *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium - Volume 3*, ser. VM'04, 2004, pp. 10–10.

[16] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: effective sampling for lightweight data-race detection," *SIGPLAN Not.*, vol. 44, no. 6, pp. 134–143, Jun. 2009.

[17] M. D. Bond, K. E. Coons, and K. S. McKinley, "Pacer: proportional detection of data races," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10, 2010, pp. 255–268.