A Deployable Sampling Strategy for Data Race Detection

Yan Cai[1], Jian Zhang[1], Lingwei Cao[1], and Jian Liu[2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
ycai.mail@gmail.com, zj@ios.ac.cn, lingweicao@gmail.com, liujian6@iie.ac.cn

ABSTRACT

Dynamic data race detection incurs heavy runtime overheads. Recently, many sampling techniques have been proposed to detect data races. However, some sampling techniques (e.g., *Pacer*) are based on traditional happens-before relation and incur a large basic overhead. Others utilize hardware to reduce their sampling overhead (e.g., *DataCollider*) and they, however, detect a race only when the race really occurs by delaying program executions. In this paper, we study the limitations of existing techniques and propose a new data race definition, named as *Clock Races*, for low overhead sampling purpose. The innovation of clock races is that the detection of them does not rely on concrete locks and also avoids heavy basic overhead from tracking happens-before relation. We further propose *CRSampler* (Clock Race Sampler) to detect clock races via hardware based sampling without directly delaying program executions, to further reduce runtime overhead. We evaluated *CRSampler* on Dacapo benchmarks. The results show that *CRSampler* incurred less than 5% overhead on average at 1% sampling rate. Whereas, *Pacer* and *DataCollider* incurred larger than 25% and 96% overhead, respectively. Besides, at the same sampling rate, *CRSampler* detected significantly more data races than that by *Pacer* and *DataCollider*.

CCS Concepts

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

Keywords

Data race, sampling, concurrency bugs, data breakpoints.

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 these accesses is a write [19][45]. Race occurrences may lead to occurrences of other concurrency bugs [39] and may result in real-world disasters [4][31][43].

On race detection, static techniques could scale up to a whole program but may report many false positives [26][37][41][51]. Dynamic techniques report fewer false positives. They are mainly based on either the *lockset discipline* [45] or the *happens-before relation* [19][29]. The lockset discipline requires that all accesses to a shared memory location should be protected by a common set of locks. However, even violating such a discipline, no data race may occur [8][12][19]. The Happens-before relation [29] is usually implemented via vector clocks [19]. Each vector clock contains n clock elements, where n is the number of threads. They are used to track statuses of threads, locks, and memory locations. Race detectors implementing vector clocks incur higher overhead than the lockset based ones. *FastTrack* [19] further improves the overhead of the happens-before based race detectors to be the same level as that of lockset based ones by avoiding most of $O(n)$ operations on memory accesses. However, *FastTrack* still incurs from 400% to 800% overhead [12][19][54].

To reduce runtime overhead, sampling techniques [8][35][59] were introduced to only monitor a small set of memory accesses. They could be deployed at the program user sites if they incur an enough low overhead (e.g., less than 5%) [8]. *LiteRace* [35] targets to sample memory accesses from cold (i.e., not frequently called) functions. However, it fully monitors synchronization operations, even those in non-sampled functions, and maintains data structures for threads, locks, and memory locations (needed by happens-before based race detectors). As a result, the overhead of *LiteRace* varies from several percentages to ~140% [35]. Besides, *LiteRace* needs to log various events for offline race detection, which may further prevent it from being deployed at user sites. *Pacer* [8] introduces periodical sampling strategy. It only tracks memory accesses and synchronization operations in full during its sampling periods. In non-sampling periods, it only checks race occurrences. However, *Pacer* is based on dynamic sampling (i.e., making sampling decision online) and has to maintain basic data structures like *LiteRace*, incurring certain basic overhead. For example, with 0% and 3% sampling rates, *Pacer* incurs 33% and 86% overhead, respectively [8]. Such overhead makes *Pacer* impractical to be deployed at user sites, as an acceptable overhead at user sites is usually 5% [6][27][33][60].

The latest sampling strategy *DataCollider* [17] completely discards both the monitoring on synchronization operations and the maintenance on data structures. It utilizes code and data breakpoints of hardware architectures to support its sampling for race detection. (A code breakpoint is set to a program instruction and is fired if the instruction is executed. A data breakpoint is set to a memory address and is fired if a memory access to the address is executed.) *DataCollider* firstly sets a code breakpoint on a random instruction. If this code breakpoint fires, it further sets a data breakpoint on the target address of this instruction and delays the execution of the instruction until the data breakpoint fires or a certain time limit is reached. If a data breakpoint fires, there must exist another access to the same address. Then, a data race is reported if at least one of the two accesses is a write.

DataCollider could incur a low runtime overhead by only focusing on memory accesses via hardware supports. However, by discard-

FSE'16, November 13–18, 2016, Seattle, WA, USA
ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950310

ing the data structures of memory locations, it can only detect data races actually occurring in a run but may miss those races that do not occur but could be detected by happens-before based detectors like *Pacer* and *LiteRace*. Besides, with a slight increase in sampling rate, its overhead increases quickly as it directly delays program execution. This increase is significantly larger than the increase by *Pacer*. As a result, *DataCollider* also becomes ineffective, and could only work at user sites at an extremely low sampling rate, which further makes it ineffective.

In this paper, we firstly analyze the limitations of existing sampling approaches on race detection. We then propose a light but novel definition of data races, called *Clock Races*, based on thread-local clocks without the need of vector clocks and concrete locks. Clock races are provable to be also happens-before races (i.e., those detectable by happens-before based detectors, *HB race* for short). The benefit of clock races is that the detection of them does not require heavy tracking on concrete locks. This results in $O(1)$ rather than $O(n)$ operations on synchronization events and hence further avoids memory maintenance overhead on the involved locks. We then propose *CRSampler*, a novel sampling approach for detection of clock races based on hardware support. Compared to *Pacer*, *CRSampler* does not rely on heavy tracking of happens-before relation, avoiding basic tracking overhead. Compared to *DataCollider*, *CRSampler* does not directly delay program execution to trap a second access, which not only reduces runtime overhead but also achieves a bigger capability on race detection.

Following *Pacer*, we have implemented *DataCollider* and *CRSampler* within Jikes RVM [3] and compared *CRSampler* with both *DataCollider* and *Pacer* on six benchmarks from Dacapo [9], including a large-scale `eclipse`. The experimental results show that, at 1% sampling rate, *CRSampler* only incurred less than 5% overhead on average; but *Pacer* incurred more than 25% overhead on average and *DataCollider* incurred more than 96% overhead on average. *CRSampler* also detected significantly more races (i.e., 404 races in total) than that by *Pacer* and *DataCollider* (31 and 20 races, respectively). Besides, with the increase in sampling rate from 0.1% to 1.0% (with step 0.1%), *CRSampler* not only incurred the least overhead increase but also detected an obviously increasing number of races. However, with the same increase on sampling rate, both *DataCollider* and *Pacer* incurred a larger overhead increase than that by *CRSampler*; they almost detected a constant number of races on most of the benchmarks.

In summary, the main contributions of this paper are:

- It presents a novel definition of data races, named as clock races. Clock races are proved to be also happens-before races. Detection of clock races only requires $O(1)$ operations in time on synchronization events, without the need of concrete synchronization objects.
- It presents *CRSampler* framework to sample clock races based on hardware supports. Unlike existing techniques, *CRSampler* does not directly delay program execution and hence incurs much lower overhead than existing ones.
- We have implemented *CRSampler* as a prototype tool (see http://lcs.ios.ac.cn/~yancai/cr/cr.html). The experiments confirm that, compared to state-of-the-art happens-before based *Pacer* and the hardware based *DataCollider*, *CRSampler* is significantly much more effective and efficient.

In the rest of this paper, Section 2 gives the background, followed by motivations in Section 3. Section 4 presents our clock races definition and *CRSampler*. The evaluation is in Section 5. Section 6 discusses related works and Section 7 concludes this paper.

2. BACKGROUND

A **multithreaded program** p consists of a set of threads $t \in \mathsf{T}$, a set of locks (or lock/synchronization objects) $l \in \mathsf{L}$, and a set of memory locations $m \in \mathsf{M}$. Each thread $t \in \mathsf{T}$ 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**, including:

(1) *Memory access* events: *read* or *write* to a memory location m, and

(2) *Synchronization* events: *acquire* or *release* a lock l. (Other synchronization events can be similarly defined [19].)

We define a **Non-Sync Block** (**NSB** for short), during an execution, as a sequence of consecutive memory accesses between two synchronization events such that no other synchronization event exists between the two synchronization events.

The **Happens-before relation** (denoted as $\rightarrowtail$, HBR for short) is defined by the following three rules [29]:

(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 $\alpha \rightarrowtail \beta$.

(3) If $\alpha \rightarrowtail \beta$ and $\beta \rightarrowtail \gamma$, then $\alpha \rightarrowtail \gamma$.

HBR is typically represented by **vector clocks** [19]. A vector clock C is an array of thread-local clocks. A clock is an integer, one for each thread t (denoted as $t.clock$). During program execution, one vector clock is allocated for each thread t, for each lock l, and for each memory location m, denoted as C_t, C_l, and C_m, respectively. The i-th element in any vector clock C (i.e., $C[i]$) represents the last known clock of the thread t with $t.tid = i$. Specially, for a thread t, $C_t[t.tid]$ is equal to $t.clock$. For a thread t, $t.clock$ is only incremented right after its value is distributed to others (e.g., locks) on synchronization events [19]. Therefore, during an execution, we always have [19] :

- $t.clock = C_t[t.tid]$, for any thread.
- $C_t[t.tid] > C_{any}[t.tid]$, where C_{any} is a vector clock of any thread (but not thread t) or any lock, and
- $C_t[t.tid] \geq C_m[t.tid]$, where C_m is a vector clock of any memory location m.

3. MOTIVATIONS

3.1 How to define and detect data races?

A data race occurrence involves two accesses from different threads including at least one write access [19]. However, there is no "gold standard" to define data races [17]. Existing techniques usually rely on either the locking discipline [45] or the happens-before relation [29]. The locking discipline defines a data race on a memory location, if all accesses to the memory location is not protected by a common set of locks. Approaches based on locking discipline may report false positives as discussed in Section 1.

3.1.1 Happens-before based approaches

The happens-before relation (HBR) defines [19] a race on two memory accesses e_1 and e_2, with a write on the same memory location, if neither $e_1 \rightarrowtail e_2$ nor $e_2 \rightarrowtail e_1$. Such kind of races is known as *HB Races* [48].

HBR requires full tracking of synchronizations from all threads. Algorithm 1 shows a simplified basic HB race detector. The functions *onAcquire*() and *onRelease*() (lines 1–11) track synchronization events *acquire*() and *release*(), respectively, to maintain vec-

Algorithm 1: Simplified Basic HB Race Detector

```
1.  onAcquire(Lock $l \in L$) {
2.     Let $t$ be the current thread;
3.     Fetch vector clocks of $l$ and $t$ as $C_l$ and $C_t$, respectively.
4.     $C_t := C_l \sqcup C_t$;  //i.e., for each $i$, $C_t[i] := max\{C_l[i], C_t[i]\}$     //$O(n)$
5.  }
6.
7.  onRelease (Lock $l \in L$) {
8.     Let $t$ be the current thread;
9.     Fetch vector clocks of $l$ and $t$ as $C_l$ and $C_t$, respectively.
10.    $C_l := C_l \sqcup C_t$;  $C_t[t.tid] := C_t[t.tid] + 1$;     //$O(n)$
11. }
12.
13. //simplified, do not distinguish read or write clock of m.
14. onRead/onWrite(Memory location $m \in M$) {
15.    Let $t$ be the current thread;
16.    Fetch vector clock of $t$ and $m$ as $C_t$ and $C_m$, respectively.
17.    $lastThd := m.lastAccessedThread()$;
18.    if( not $C_m[lastThd.tid] \rightarrowtail C_t[lastThd.tid]$)     //$O(n)$
19.          //Note that some $O(n)$ operations could be eliminated [19]
20.       Report a race on m.
21. }
```

Algorithm 2: *DataCollider*

```
1.  Input: r – a static sampling rate.
2.  Input: p – a multithreaded program.
3.  Input: timeLimit – the max time on a data breakpoint.
4.
5.  Let Σ be a set of sampled instructions in p w.r.t r.
6.  Let $\mathcal{M}$ be an empty set.
7.
8.  for each ins ∈ Σ                                    //static sampling
9.     insertCall (sample(ins)) ;
10.
11. sample (ins)                                        //event $e_1$
12. {
13.    ⟨addr, size, isWrite⟩ := PARSE (ins) ;
14.
15.    if ( isWrite)   setDataBreakpointRW (addr, size) ;
16.    else            setDataBreakpointW (addr, size) ;
17.
18.    delay (timeLimit);   //wait for an event $e_2$
19.    clearDataBreakpoint (addr) ;
20.
21.    if (addr ∈ $\mathcal{M}$) reportDataRace (ins) ;
22.    $\mathcal{M} := \mathcal{M} \setminus \{addr\}$;
23. }
24.
25. onDataBreakpointFired (addr)                        //event $e_2$
26. { $\mathcal{M} := \mathcal{M} \cup \{addr\}$;  }
```

tor clocks for threads and locks. During tracking, the vector clocks of the involved threads and locks are firstly fetched (i.e., C_t and C_l lines 3 and 9, respectively). Then an $O(n)$ operation (as noted) is performed to update the vector clock of the thread (i.e., $C_t := C_l \sqcup C_t$ at line 4) or the vector clock of the lock (i.e., $C_t := C_l \sqcup C_t$ at line 10). After that, the updated vector clock holds the latest thread-local clocks from the two vector clocks. The functions *onRead*() and *onWrite*() check whether a data race occurs when a read or a write occurs. Still, the vector clocks of the thread and the memory location are firstly fetched (line 16). Then, the algorithm checks whether the last access to *m* (by thread *lastThd*) happens-before the current access (line 18). If no happens-before relation exists from the last access to the current one, a data race is reported (if one of two accesses is a write, omitted in Algorithm 1). The check is also $O(n)$ as noted.

In Figure 1, we also show an example to illustrate the heavy tracking of HB based approaches. Figure 1(a) shows a Java program *p*.

```
thread t1        thread t2
acquire(l)       acquire(k)
  x = ...        ...
  ...            release(k)
release(l)       x += ...
```

(a)

```
thread t1        thread t2
acquire(l)       acquire(k)
delay();         ...
x = ...          release(k)
  ...            x += ...
release(l)
```

(c)

```
   thread t1            thread t2
acquire(l)
onAcquire(l);         acquire(k)
  x = ...             onAcquire(k);
onWrite(x);           ...
  ...                 onRelease(k);
onRelease(l);         release(k)
release(l)            tmp = x ...
                      onRead(x);
                      x = tmp;
                      onWrite(x);
```

(b)

Figure 1. (a): A program *p* with a race on *x*; (b): an instrumentation on *p* (in bold) by HB race detectors (e.g., *Pacer*); (c): a delay inserted by *DataCollider*.

The program *p* contains a data race on *x*: accesses to *x* from two threads t_1 and t_2 (i.e., "*x* =…" and "*x* +=…") could occur concurrently. Figure 1(b) shows how each synchronization is instrumented to track HBR in Algorithm 1: for each synchronization, a call *onAcquire*(...) or *onRelease*(...) is inserted and the lock object (i.e., lock *l* or lock *k*) is taken as an argument. And on each read or write, an *onRead*() or an *onWrite*() is also invoked.

As a result, the tracking of HBR itself (i.e., without race detection) incurs high overhead. For example, *Pacer* reports 15% overhead [8], which already includes various optimizations. In our practice, we also experienced about 15% overhead on tracking HBR. According to our experience, there are two factors contributing to the overhead: (1) the fetching of vector clocks of locks and threads (lines 3 and 9 in Algorithm 1), accounting for more than 5% overhead on average, and (2) operations on vector clocks of locks and threads (lines 4 and 10 in Algorithm 1), contributing more than 8% overhead on average.

Therefore, HBR via vector clocks is unlikely suitable for data race sampling as its tracking overhead (without race detection) is already larger than 5%, even with various optimizations [8]. Although other existing works target to track a subset of HBR [6][27] to reduce the tracking overhead to be low enough (i.e., less than 5%), their tracking is only designed to check two known memory accesses in a production run [6].

3.1.2 Hardware based approaches

Recently, *DataCollider* defines a race on two accesses (with a write on the same memory location), if they are executed at the same physical time. In this paper, we call such races *Collision Races*, which are also HB races [17].

Algorithm 2 shows the *DataCollider* algorithm. Given a sampling rate *r* (and a time limit *timeLimit* discussed in Section 4.2), *DataCollider* randomly chooses a set of instructions to sample (lines 5–9). For each sampled instruction *ins*, it delays the execution of *ins* and, at the same time, waits for a second access (line 18). The trapping of a second access is done by setting up a hardware data breakpoint on the target address of *ins* (line 15–16). If the data breakpoint fires (lines 25–26), a data race occurs (lines 21–22). The map $\mathcal{M}$ at line 6 is used to check whether any data breakpoint fires on the address from the instruction of the delayed thread (lines 22 and 26).

For example, Figure 1(c) shows how *DataCollider* detects the race on *x*. Suppose that the write to *x* by thread t_1 is sampled, then *DataCollider* sets a data breakpoint on the address of *x* and then delays the write to *x* for a certain time. During the delayed period, if thread t_2 reads or writes to *x*, the data breakpoint fires. Then the race on x is detected and *DataCollider* resumes the execution of thread t_1.

As the hardware breakpoint mechanism (supported as debug utilizations on modern CPUs [17]) incurs almost no overhead, *DataCollider* is able to limit almost all its overhead to be that caused by its delay. As the delay of *DataCollider* directly contributes to its overhead, the overhead of *DataCollider* could also be large. And with increasing sampling rate, its overhead increases quickly (see our experiments in Section 5.3). Of course, *DataCollider* could be configured to incur much lower overhead (e.g., 5%) by limiting its sampling rate to be much lower; but this also results in an extremely lower race detection rate and makes *DataCollider* much more ineffective.

Besides, the detection of collision races suffers the following limitations. First, if a collision race is detected, the race has actually occurred. Therefore, once deployed at user sites, the occurrences of harmful collision races may cause unexpected results to users. Second, not all races could be easily detected in such a way [6][12], especially on large-scale programs (e.g., the eclipse program in our experiment). As a result, *DataCollider* loses certain race detection ability compared to HBR based race detectors.

Therefore, our first two insights are, for a lightweight sampling technique:

> *(1) It should not define a race according to the full HBR involving vector clock operations;*
> *(2) It should not directly delay executing an instruction.*

In Section 4, we present our definition of data races with respect to the above two requirements.

3.2 Dynamic Sampling vs. Static Sampling

Dynamic sampling (as well as dynamic full race detection [19]) requires fully instrumenting program instructions, even many instructions may not be sampled during executions. For example, in Figure 1(b), every access to x incurs *onRead*(x) or *onWrite*(x) calls (of course, these calls could be inlined); but the sampling decision is made within these two functions. In other words, for dynamic sampling, to sample or not to sample an instruction is unknown until the instruction is about to be executed. And at that time, the instrumented function calls have been executed. As a result, additional overhead is incurred due to these per-instruction instrumentations prior to the sampling decision making. That is part of reasons contributing to large overhead of *Pacer* (i.e., 33% [8]) at 0% sampling rate.

However, static sampling only requires to instrument exact instructions that are to be sampled, as adopted by *DataCollider*. That is, the sampling decision could be made offline or during program loading time; and no additional overhead is introduced on those instructions not to be sampled.

Dynamic sampling incurs some basic overhead for all instructions (related to memory accesses as well as potential HBR maintenance) while static sampling only incurs overhead for those to-be-sampled instructions. From this point, static sampling is more suitable for lightweight sampling techniques. Therefore, our third insight is that:

> *(3) A lightweight sampling technique should adopt static sampling strategy to reduce its overhead per execution.*

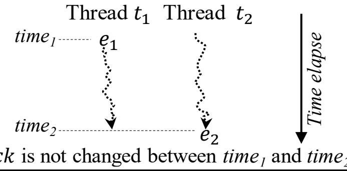


Figure 2. Illustration of clock races.

4. CRSAMPLER

4.1 Clock Races

As discussed in Section 3, the detection of HB races requires instrumentation on synchronization operations. Such operations involve fetching vector clocks of locks and threads as well as $O(n)$ operations on them. The resultant overhead is already more than 5% in previous experiment [8]. However, the definition of HB races is able to predict races that did not really occur in the monitored executions but may occur in other executions, which is more powerful than *DataCollider* that detects a race only when it really occurs.

Therefore, we only consider a subset of HBR to define data races. Our definition of data races only relies on thread-local clocks. Formally, we define our *Clock Races* as follows:

Definition 1. *Two memory accesses e_1 by thread t_1 and e_2 by thread t_2 form a* ***Clock Race*** *if:*

1) *The two accesses e_1 and e_2 operate on the same memory location and at least one of them is a write access, and*
2) *At time that e_2 occurs, $t_1.clock$ remains the same as that when t_1 performs e_1.*

The first condition is the basic requirement of a race definition. The second one is depicted in Figure 2. It requires that, between the occurrences of two accesses e_1 and e_2 in physical time (i.e., the period from *time₁* to *time₂* in Figure 2), the thread-local clock of t_1 remains unchanged. That is, during the period, thread t_1 either (Case 1) does not execute any event (including *sleep*()/*wait*() events that involve two increments on a thread's local clock, resulting three non-sync blocks [19]) or (Case 2) only executes memory accesses in the same non-sync block. Therefore, *DataCollider* only defines a subset of our clock races corresponding to Case 1 where a thread t_1 does not execute any event due to delaying by *DataCollider* (see Theorem 2 below).

We present two theorems to show that (1) clock races are also HB races as the second condition is a subset of HBR (i.e., the correctness of clock races), and (2) all races detected by *DataCollider* are also clock races (corresponding to the above Case 1).

Theorem 1. *If two events e_1 and e_2 form a clock race, they also form an HB race.*

Proof Sketch. (1) Firstly, it is impossible that $e_2 \rightarrowtail e_1$ as e_2 appears after e_1 occurs, which is implied by the second point of Definition 1.

(2) Suppose that $e_1 \rightarrowtail e_2$ is true. Let $C_m^{e_1}$ be the vector clock of m when e_1 occurs. When e_2 occurs, from $e_1 \rightarrowtail e_2$, we have:

$$C_{t_2}[t_1.tid] > C_m^{e_1}[t_1.tid] \qquad \text{EQ1}$$

(Otherwise, an HB race already occurs [8][19] and hence $e_1 \rightarrowtail e_2$ does not hold). However, as $t_1.clock$ is not changed since e_1 occurs, by definition of clock races (in Section 2), we then have:

$$C_m^{e_1}[t_1.tid] = C_{t_1}[t_1.tid] = t_1.clock \qquad \text{EQ2}$$

From EQ1 and EQ2, we know that thread t_1 does not own a largest clock of itself as $C_{t_2}[t_1.tid] > t_1.clock$, which contradicts the fact that a thread always has the largest clock of itself than any other thread. Therefore, the assumption $e_1 \rightarrowtail e_2$ is not true.

From (1) and (2), neither $e_1 \rightarrowtail e_2$ nor $e_2 \rightarrowtail e_1$ holds. As e_1 and e_2 operate on the same memory location and one of them is a write, by definition of the HB race, e_1 and e_2 form an HB race. Hence, a clock race is also an HB race. □

Theorem 2. *If two events e_1 and e_2 are detected by DataCollider as a collision race, they also form a clock race.*

Proof Sketch. Firstly, by definition of collision race, both events e_1 and e_2 operate on the same memory location and one of them is a write access. Secondly, suppose that the event e_1 is the event delayed by *DataCollider* (the other case can be proved similarly). When the event e_2 occurs, the thread (say t_1) performing e_1 does not execute any other event as it is delayed; therefore, the thread-local clock of thread t_1 is not changed. According to the definition of clock race, the two events e_1 and e_2 also form a clock race. □

Detection of clock races requires only thread-local clocks to identify NSB (non-sync blocks). Therefore, it is enough to perform a light instrumentation. Figure 3 shows how the example program is instrumented (i.e., "*onSync*()"). Compared to instrumentation for HB races as shown in Figure 1(b), on synchronization event, (1) no concrete lock is required (i.e., "*onSync*()" but not "*onAcquire*(***l***)" or "*onRelease*(***l***)"), and (2) hence no vector clock of locks or threads is operated, avoiding *O*(*n*) operations. Therefore, a call "*onSync*()" is enough for each synchronization event, which only increments thread-local clocks of the involved threads (see lines 41–45 in Algorithm 3 for details).

Figure 3 also shows how a clock race on *x* is detected. Let the initial clocks of two threads t_1 and t_2 be 10 and 8 (or any two other integers), respectively. For both threads, on their acquisitions of lock *l* and lock *k*, their clocks are incremented by 1 via calls *onSync*(). Then, their clocks become 11 and 9, respectively. Suppose that the write to *x* by thread t_1 is sampled (i.e., "*sample*(*x*)"). Next, suppose that before thread t_1 releases lock *l*, thread t_2 releases lock *k*, followed by its read and write to *x*. At this time, the clock of thread t_1 is still 11 which is the same as that when its write to *x* is sampled. Therefore, a clock race on *x* is detected.

Note that, if thread t_1 first releases lock *l* before thread t_2 reads from and writes to *x*, no clock race is then detected. It is because the clock of thread t_1 is changed to 12 which is different from that when the write to *x* by thread t_1 is sampled. Although in both cases, full HBR based race detectors (e.g., *FastTrack*) are able to detect the race on *x*, HBR based sampling detectors also suffer from the similar limitations. For example, *Pacer* is able to detect the race on *x* only if (1) the two accesses by two threads occur in the same sampling period or (2) one occurs in a sampling period and the second occurs in the right followed non-sampling period. For other cases, *Pacer* is unable to detect this race (e.g., both accesses occur in a non-sampling period or one access occurs in a non-sampling period and the second one occurs in a sampling period). For *DataCollider*, it is able to detect the race by delaying the write to *x* by thread t_1 until thread t_2 reads from and writes to *x* (as shown in Figure 1(c)). However, as discussed in Section 3, such delays directly increase its overhead. With increasing number of delays, its overhead increases significantly fast (see Section 5.3.1).

Thread t_1		$t_1.clock$	$t_2.clock$	Thread t_2	
		10	8		
acquire(*l*)	**onSync( );**	11	9	*acquire*(*k*)	**onSync( );**
x = ...	**sample(*x*);**	11	9	…	**onSync( );**
…		11	10	*release*(*k*)	
	onSync( );	11	10	*x* += …	
release(*l*)		12			

Figure 3. Instrumentation of *CRSampler* (highlighted) and detection of a clock race on *x* with thread-local clocks.

Discussion on Definition of Clock Races. In Definition 1, a clock race requires no change on $t_1.clock$. Symmetrically, we could also define a clock race by extending Definition 1 (i.e., the second bullet): if at time when e_2 occurs, $t_2.clock$ is not changed since the event e_1 occurs. This extension could increase the detection rate of clock races. However, it requires tracking of all clocks of any other threads by thread t_1 when e_1 occurs. This tracking is *O*(*n*) in time and increases sampling overhead. In our preliminary experiment, this extension slightly increased race detection rates; however, it doubled sampling overhead, making detection of clock races inefficient. Therefore, we only follow Definition 1 to detect races by sampling, avoiding incurring any *O*(*n*) operations.

4.2 Static and Hardware Based Sampling

CRSampler adopts static sampling strategy to sample each static instruction based on our third insight discussed in Section 3.2. When a sampled instruction is being executed, the access to the memory location (i.e., address) in this instruction is taken as a first event e_1 . *CRSampler* further sets a data breakpoint on this memory location to trap an event e_2. If such an event e_2 occurs and the clock of the thread performing e_1 remains unchanged, a clock race is detected. Note that, unlike *DataCollider*, during the trap of an event e_2, no thread is delayed.

However, *CRSampler* has to consider how much time is allowed to trap an event e_2. This is similar to *DataCollider* that has to determine how long it should delay the execution of a thread. It is because, a short time may not be enough for a race to occur. But a long time may make the usage of the data breakpoints ineffective, as the number of data breakpoints is limited. For example, popular X86 CPU supports only four data breakpoints and others may only support one [28].

One strategy is, like *DataCollider*, to set a constant time limit. Once such a time limit is reached, event e_1 is discarded. This strategy is simple and does not incur additional overhead. The second one is to monitor the clock changes of all threads, once the thread performing e_1 increments its clock, event e_1 is then discarded and the taken data breakpoint is cleared. The second strategy seems more effective than the first one. However, it requires additional overhead on synchronization events (e.g., a check on whether an event e_1 is sampled from the execution of the current thread). Therefore, *CRSampler* adopts the first strategy and sets a time limit (which is the same as that by *DataCollider*).

4.3 CRSampler Algorithm

Algorithm 3 shows the *CRSampler* algorithm. Given a sampling rate *r* and a program *p*, *CRSampler* first selects a set of instructions Σ randomly according to the given sampling rate *r*, and instruments these instructions (lines 8 and 9) to sample memory accesses at runtime. The input *timeLimit* is the max time for a data breakpoint to be valid for a given address. *CRSampler* maintains a thread-local clock for each thread *t* as $t.clock$ and a data structure $\mathcal{M}$. (Note that, to simplify our presentation, we use the notation

Algorithm 3: *CRSampler*

```
1.  Input: $r$ – a static sampling rate.
2.  Input: $p$ – a multithreaded program.
3.  Input: timeLimit – the max time on a data breakpoint.
4.
5.  Let $\Sigma$ be a set of sampled instructions in $p$ w.r.t $r$.
6.  Let $\mathcal{M}$ be a map from an address to a pair $\langle t, clock \rangle$.
7.
8.  for each ins $\in \Sigma$                                  //static sampling
9.    insertCall (sample(ins)) ;
10.
11. sample (ins)                                               //event $e_1$
12. {
13.   $\langle addr, size, isWrite \rangle$ := PARSE (ins) ;
14.   if ( isWrite)  setDataBreakpointRW (addr, size) ;
15.   else           setDataBreakpointW (addr, size) ;
16.
17.   Let $t$ be the current thread;
18.   $\mathcal{M} := \mathcal{M} \cup \{\langle addr, \langle t, t.clock \rangle \rangle\}$;
19.   setTimer (addr, timeLimit); //unlike DataCollider, no delay
20. }
21.
22. onDataBreakpointFired (addr)                               //event $e_2$
23. {
24.   Let $t$ be the current thread;
25.   $\langle$ lastThd, lastClock $\rangle := \mathcal{M}_{addr}$ ;
26.   //check clock races
27.   if ($t \neq lastThd \wedge lastClock = lastThd.clock$ )
28.   {
29.     reportDataRace (ins) ;
30.     clearDataBreakpoint (addr) ;
31.     $\mathcal{M}_{addr} := \emptyset$;
32.   }
33. }
34.
35. onTimer(addr)
36. {
37.   clearDataBreakpoint (addr) ;
38.   $\mathcal{M}_{addr} := \emptyset$;
39. }
40.
41. onSync ()                                      //synchronization events
42. {
43.   Let $t$ be the current thread;
44.   $t.clock := t.clock + 1$; //unlike HBR, no O(n) operations
45. }
```

$\mathcal{M}_x$ to denote the mapping from x to $\mathcal{M}(x)$.) The structure $\mathcal{M}$ maps from one address to a pair of a thread and a clock, corresponding to the thread t performing event e_1 in definition of clock race and the clock of t at that time. Note that, (1) the map $\mathcal{M}$ of Algorithm 3 is different from that in Algorithm 2; (2) as the number of data breakpoints is limited, operations over $\mathcal{M}$ are actually $O(1)$ operations in time (implemented via a global unique index for each data breakpoint).

At runtime, once an instrumented instruction *ins* is being executed, the instrumented call *sample*(*ins*) fires. *CRSampler* then sets a data breakpoint (lines 11–20) to the memory address (i.e., *addr*) that the instruction *ins* is about to access. (We use a function *PARSE*() to denote the extraction of the address *addr* and the size *size* in byte associated with the instruction *ins*, as well as *isWrite* indicating whether this instruction is a write one.) There are two types of data breakpoints: *read-write* data breakpoint and *write* data breakpoint. The former fires if either a read or a write to the target address occurs; the latter fires only if a write to the target address occurs. *CRSampler* chooses either type of breakpoint according to whether the sampled access is a write or a read (lines 13–15). That is, a read access only forms a data race with a write access while a write access forms a data race with either a read or a write access. After setting the breakpoint, the thread id and the clock of the current thread is mapped in $\mathcal{M}$ from the address *addr* (lines 17 and 18); and a time limit for this *addr* is set (line 19).

Once a data breakpoint on an address *addr* fires (corresponding to event e_2 of the clock race definition), a clock race on *addr* occurs if the current thread t is different from the last thread *lastThd* and the current clock of thread *lastThd* remains the same as that mapped in $\mathcal{M}$ (lines 27–29). A data breakpoint is cleared if either a second event e_2 occurs (as stated in the last paragraph) (lines 30–31) or a time limit is reached (lines 35–39, i.e., the function *onTimer*(*addr*)).

The function *onSync* () is called whenever a synchronization event occurs to increment the thread-local clock of the corresponding thread (lines 41–45).

Algorithm comparison. Compared to *Pacer* that is based on HBR, *CRSampler* does not maintain full HBR tracking but only a thread-local clock on synchronization events (i.e., $t.clock := t.clock + 1$ instead of $C_t := C_l \sqcup C_t$ in Algorithm 1). Therefore, *CRSampler* invokes *onSync*() instrumentation call but not *onAcquire*(*l*) or *onRelease*(*l*). Thus, *CRSampler* is able to avoid heavy tracking incurred by fetching lock objects and their vector clocks as well as the corresponding *O*(*n*) operations.

Compared to *DataCollider*, *CRSampler* does not rely on direct delays to actually trigger race occurrences. Instead, it checks races according to thread-local clocks if any data breakpoint fires. Such race detection avoids direct delay-caused overhead. Besides, with a longer time limit, *CRSampler* does not incur a larger overhead; however, for *DataCollider*, its overhead is increased by the same fold of the increase in its time limit. This is also verified by our experiments (see Section 5.3.1). Of course, given the same scheduling and the same set of sampled memory accesses, *CRSampler* guarantees to detect all races detected by *DataCollider*, as each collision race is a clock race (see Theorem 2). In practice, *CRSampler* is able to detect more races as it does not delay any execution, resulting in more memory accesses sampled (see our experiment in Section 5).

4.4 Limitations

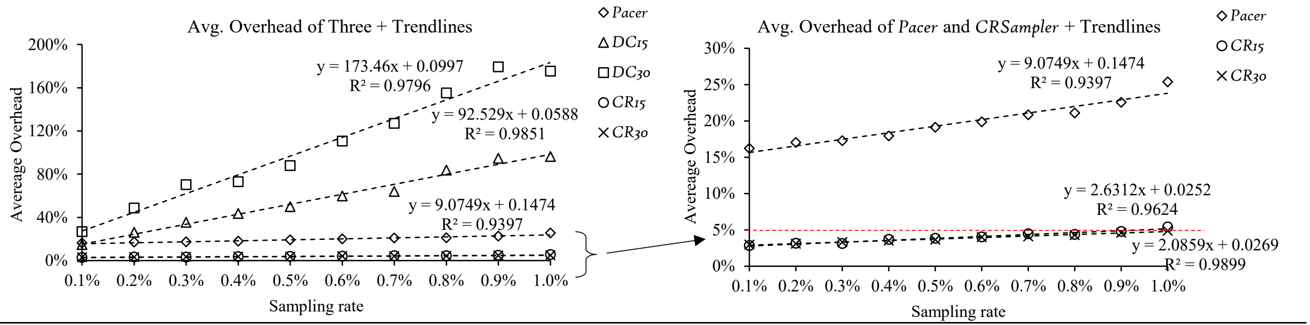
Like existing sampling based techniques, *CRSampler* also misses data races as expected. *CRSampler* utilizes thread-local clocks to detect HB races. It then suffers limitations suffered by HB race detectors. One of such limitations is the report of false positives: two accessed are reported as an HB race but they cannot occur at the same time. And data dependency is one factor. That is, two accesses form an HB race; but the second access may depend on the value of the memory location of the first access. However, *DataCollider* does not suffer from such a limitation as it only detects those data races occurring at the same time.

5. EXPERIMENTS

This section presents the evaluation on *CRSampler* (*CR* for short). We compared it with *Pacer* and *DataCollider* (*DC* for short), and indirectly compared it with *LiteRace* [35] via *Pacer* [8].

5.1 Implementation and Benchmarks

Implementation. We have implemented *DC* and *CR* in Jikes RVM [3][5], a widely used research JVM [8][12]. *Pacer* has also been implemented in Jikes RVM and is available online [2], we used its downloaded implementation. The static sampling of *DC* and *CR* is performed at Java class loading time.



(a) Average overhead of three techniques

(b) Average overhead of *Pacer* and *CRSampler*

Figure 5. Average runtime overhead.

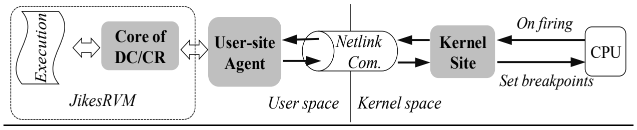


Figure 4. Architecture of *CRSampler* and *DataCollider*.

For *DC* and *CR*, the use of data breakpoints is only allowed at Linux kernel. We implemented them for Java programs as depicted in Figure 4: for each sampled access, the target address of the access is extracted within Jikes RVM (i.e., *Core of DC/CR*). It is passed to a *User-site Agent* (implemented in the interface extension of the Jikes RVM) and is then sent to Linux kernel-site via Netlink communication ("*Netlink Com.*") [1]. At Linux kernel-site, a data breakpoint is set on the given address. Once a data breakpoint fires, a message is sent from the kernel-site to the user-site agent, and is then sent to *DC*/*CR*. Both *DC* and *CR* set at most four breakpoints at a time as our experiment environment supports four data breakpoints.

Another challenge is that, unlike C/C++, Java offers reference types (and primitive types) with automatic garbage collection but not direct pointer types. Therefore, an address is not only accessed by application threads (i.e., those created by Java applications), but also accessed by Java VM threads (e.g., on objects moving and garbage collection). The latter kind of accesses does not incur data races with accesses from application threads. However, it is impossible to identify and skip such accesses from VM threads at hardware level. Therefore, once a data breakpoint fires, *DC*/*CR* extracts the native thread id (i.e., Pthread on Linux) and checks whether such a thread is a Java application thread or a Java VM thread. It discards any access from Java VM threads.

Table 1. The statistics of each benchmark and the number of total races detected by each technique.

Bench-marks	Binary Size (KB)	# of threads	# of sync.	*Pacer**	*DC15*	*DC30*	*CR15*	*CR30*
avrora09	2,086	7	3,312,801	3	3	3	5	3
xalan06	1,027	9	35,859,489	5	5	5	87	81
xalan09	4,827	9	12,599,144	0	2	2	84	91
sunflow09	1,017	17	1,590	0	0	2	46	45
pmd09	2,996	9	20,550	4	2	2	110	121
eclipse06	41,822	16	51,131,093	19	2	6	58	63
			Sum:	31	14	20	390	404

*for *Pacer* with 100% sampling rate, it was reported to have detected more races [8][18]. For example, on `eclipse06`, *Pacer* detected 39 races (out of 51 known races) [18] and 77 races [8]; and on `xalan06`, *Pacer* detected 36 races (out of 41 known races) [18] and 73 races [8].

Benchmarks. We selected a set of six multithreaded benchmarks from Dacapo [9] benchmark suite that could be run correctly by Jikes RVM. These benchmarks include `avrora09`, `xalan06`, `xalan09`, `sunflow09`, `pmd09`, and `eclipse06`, where the subscripts "`06`" and "`09`" indicate their version number (i.e., "`2006`" or the "`2009`" version, respectively). Table 1 shows the statistics of these benchmarks, including the binary size, the number of threads, and the number of dynamically collected synchronizations for each benchmark. The last five columns show the number of data races detected by each technique. The last row also shows the total number of data races detected by each technique.

5.2 Experimental Setup

Our experiment was performed on a workstation with an i7-4710MQ CPU (four cores), 16G memory, and 250G SSD. The workstation was installed with Ubuntu 14.04 x86 system.

To evaluate *CR*, we run six benchmarks under *Pacer*, *DC*, and *CR* 100 times for each sampling rate from 0.1% to 1.0% with step 0.1%. Note that, unlike *DC* and *CR*, *Pacer* adopts dynamic sampling strategy. We set three techniques to work at relatively low sampling rates. This is because the three sampling techniques target to detect races at user sites and their overhead should be low enough to be accepted by users. For *Pacer*, its overhead at 0% sampling rate is already much higher (i.e., 33% in [8]) than 5%. For *DC*, it incurred nearly 100% overhead at 1% sampling rates in our experiment to be presented below.

As *DC* and *CR* adopt time limit mechanism, we selected two time limit options: 15 ms (millisecond) and 30 ms. Note that the first one is the default option of *DC* [17]. We refer to the two options of *DC* and *CR* as *DC15*, *DC30*, and *CR15*, *CR30*, respectively.

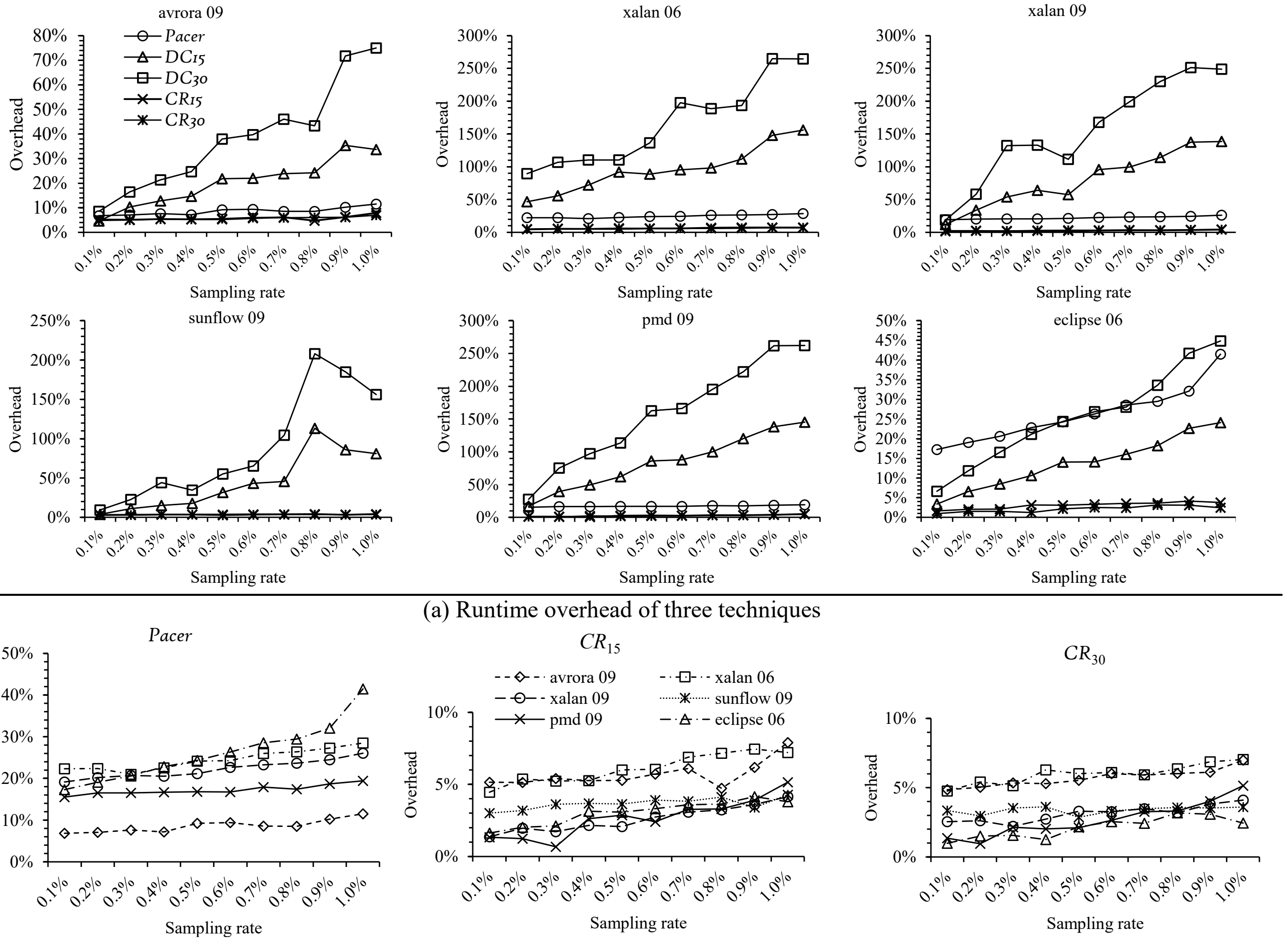
5.3 Experimental Results

We compare three techniques at their runtime overheads, total numbers of races detected, and per-race detection abilities.

5.3.1 Overhead

Figure 5[1] shows the average overhead (y-axis) of three techniques on all six benchmarks with increasing sampling rate from 0.1% to 1.0% (x-axis). Figure 6 (a) and (b) shows the overhead of three techniques on six benchmarks in detail.

[1] Note, in both Figure 5 and Figure 6, we do not count and show the overhead of *Pacer* on `sunflow09` as on which, *Pacer* incurred an overhead from 600% to nearly 2,000% with sampling rate from 0.1% to 1.0%. No evaluation of *Pacer* on `sunflow09` was reported in [8].



(a) Runtime overhead of three techniques

(b) Runtime overhead of *Pacer* and *CRSampler*

Figure 6. Runtime overhead of three techniques on each benchmark.

Average overhead. Figure 5(a) shows the average overhead of each technique. Figure 5(b) further shows the overhead comparison on *CR* and *Pacer*. The two figures also show the linear trendlines (with both equations $y = kx + b$ and goodness of fit R^2) to indicate their trends of overhead increase. Figure 5(b) also shows a dotted line indicating the 5% overhead.

The trendline equations in Figure 5 (a) and (b) show that, statistically, *DC* has the largest overhead increasing factors (i.e., 173.46 of *DC30* and 92.529 of *DC15*). Particularly, the increasing factors of *DC30* is about two times of that of *DC15*. This further validates the fact that, for *DC*, an increase in its delay time also incurs the same fold increase in its overhead. Although the increasing factor of *Pacer* (i.e., 9.0749) is less than one tenth of *DC15*, it is still significantly more than 3 times larger than that of *CR* (i.e., 2.0859 and 2.6312 for *CR15* and *CR30*, respectively). From the two factors of *CR*, we observe that, even with a longer time limit (but at the same sampling rate), the overhead of *CR* does not increase but slightly decreases (see the second point of Section 5.4).

Another insight from Figure 5 (a) and (b) is that, statistically, hardware based sampling has a smaller basic overhead (i.e., 0.0588, 0.0997, 0.0252, and 0.0269 of *DC15*, *DC30*, *CR15*, and *CR30*) than that (i.e., 0.1474 of *Pacer*) by tracking HBR. And the basic overheads of *CR15* and *CR30* are also the two least ones.

Overhead on each benchmark. Figure 6(a) shows the detailed overhead of three techniques on each benchmark, where the legend of the subfigure "avrora09" applies to all other subfigures.

From Figure 6(a), we observe that at 0.1% sampling rate, all three techniques incurred relatively small overhead. And most of their overheads are below 20%, where one exception is that *DC15* and *DC30* incurred more than 46.6% and 89.3% overhead on xalan06. However, with increasing sampling rate from 0.1% to 1.0%, both *DC15* and *DC30* incurred significantly larger overhead. This is expected as the delayed time of *DC* is directly added into its overhead (as discussed in Section 3). Whereas, *Pacer*, *CR15*, and *CR30* incurred a slight overhead increase with increasing sampling rate except on eclipse06 by *Pacer*. This slower increase shows the advantage of *Pacer* and *CR* by defining data race based on synchronization tracking (i.e., HBR tracking by *Pacer* and thread-local clock tracking by *CR*) instead of direct delaying.

Although *Pacer* has a linear increase on its overhead with increasing sampling rate, its basic overhead is much larger than that of *CR*. Figure 6(b) (where the legend of "*CR15*" applies to all other subfigures) further shows the comparison of *Pacer*, *CR15*, and *CR30* on their overhead with increasing sampling rate. From Figure 6(b), we observe that *Pacer* usually incurred from 15.5% to 22.3% overhead with 0.1% sampling rate except on avrora09 where the overhead was about 6.8%. However, the 15.5% to

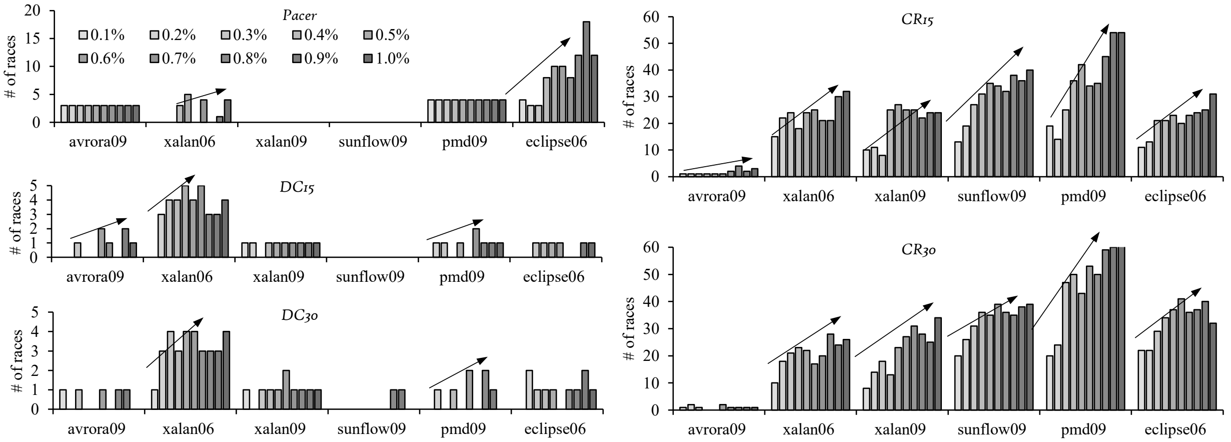


Figure 7. Comparisons on the number of distinct races detected in every 100 runs under different sampling rates.

22.3% overhead of *Pacer* is much larger than 5%. With sampling rate increased from 0.1% to 1.0%, the overhead of *Pacer* increased to more than 25% or even up to 41.4% (i.e., on eclipse$_{06}$ at 1.0% sampling rate). Compared to *Pacer*, with 0.1% sampling rate, both *CR15* and *CR30* incurred less than 5% overhead (i.e., 1.6%, 1.3%, 3.0%, 1.4%, and 4.4% of *CR15*, 0.98%, 1.4%, 3.3%, 2.5%, 4.7% and 4.9% by *CR30*) or slightly above 5% (i.e., 5.12% on avrora$_{09}$ by *CR15*). With sampling rate increased from 0.1% to 0.9%, the largest overhead by *CR15* and *CR30* is 4.20% except on xalan$_{06}$ and avrora$_{09}$. On these two benchmarks, the largest overhead is 7.4%. At 1.0% sampling rate, the largest overhead is 7.9% (i.e., on avrora$_{09}$ by *CR15*).

In summary, from Figure 5 and Figure 6, we can see that *CR* incurred a much lower overhead than that by *Pacer* and *DC*. At 1.0% sampling rate, *CR* incurred less than or slightly over 5% overhead on average, which makes *CR* practical to be deployed at user sites.

5.3.2 *Total Races Detected*

Total races detected in 1,000 runs. The last five columns of Table 1 show the total number of data races detected by *Pacer*, *DC15*, *DC30*, *CR15*, and *CR30* in all their 1,000 runs (100 runs × 10 sampling rates). As shown in Table 1, *CR* detected significantly more races than that detected by both *Pacer* and *DC*. And, *DC* detected more races on some benchmarks but fewer races on others than that by *Pacer*. Note that, on eclipse$_{06}$ and xalan$_{06}$, *Pacer* with 100% sampling rate was reported to have detected more races [8] (see the note of Table 1). However, we focus on their race detection abilities and runtime overhead at low sampling rate in this paper.

Among all races detected by three techniques, both *Pacer* and *DC* missed a larger number of races detected by *CR* as shown in Table 1. However, *CR* only missed four races detected by *Pacer* and three races detected by *DC*. All these races were from eclipse$_{06}$. This is not a surprise as eclipse$_{06}$ is a large-scale program (e.g., Table 1 shows that its binary size is nearly ten times larger than that of others). Hence, it requires more runs to detect more races from eclipse$_{06}$.

Distinct races per-100 runs with different sampling rates. In our experiment, we run each technique 100 times under 10 sampling rates. Figure 7 shows the number of distinct races detected in every 100 runs under each of 10 sampling rates. The *x*-axis firstly shows the six benchmarks and then shows 10 sampling rates from 0.1% to 1.0% for each benchmark. The *y*-axis shows the number of distinct races detected. The legend of "*Pacer*" in Figure 7 also applies to other subfigures.

From Figure 7, we observe that *CR* detected significantly more races than *Pacer* and *DC* among most of 100 runs. This indicates that *CR* has a stronger ability to detect races in each run on average than *Pacer* and *DC*.

On eclipse$_{06}$ and xalan$_{06}$, *Pacer* detected an increasing number of races with increasing sampling rate (indicated by our manually added trend-like arrows for reference); and on avrora$_{09}$, xalan$_{06}$, and pmd$_{09}$, one or both of *DC15* and *DC30* also detected an increasing number of races. However, on other benchmarks, both *Pacer* and *DC* detected almost the same number of races with increasing sampling rates. Whereas, *CR* detected an increasing number of races among all benchmarks except on avrora$_{09}$. On avrora$_{09}$, there are totally 5 races detected by *CR*, *Pacer*, and *DC*. On this benchmark, only *CR15* detected an increasing number of races.

5.4 More Discussion on *DC* and *CR*

Both *DC* and *CR* adopt time limit mechanism, although their usages are different. It is interesting that whether a longer time limit is better. In our experiment, we set two time limits: 15 ms and 30 ms. In this subsection, we analyze how different time limits affect (1) their race detection abilities and (2) their runtime overheads.

Race detection under different time limit. In theory, with a longer time limit, there are two effects on both *DC* and *CR*: (1) the total sampled accesses were reduced due to limited number of data breakpoints, which may result in fewer races to be detected; (2) on the other hand, with a longer time limit, *DC* and *CR* could detect those races needing a longer time to expose.

As shown in Table 1, when the time limit was 30 ms, *DC* detected more races on sunflow$_{09}$ and eclipse$_{06}$ by 2 and 4, respectively. On other benchmarks, *DC* detected the same number of races. However, for *CR*, there is no consistent result. On avrora$_{09}$, xalan$_{06}$, sunflow$_{09}$, *CR* with 30 ms detected less races by 2, 6, and 1, respectively; on xalan$_{09}$, pmd$_{09}$, and eclipse$_{06}$, *CR* with 30 ms detected more races by 7, 11, and 5, respectively. Therefore, it is difficult to say that a longer time limit may result in a better race detection, even for *DC*. Actually, previous work has pointed out that it is impossible to predict the time limit of *DC* [17]; and, timing operations (e.g., instrumentation calls) may *increase* the probability of a race occurrence but may also *decrease* it [8].

Overhead under different time limit. Although there is no conclusion on whether a longer time limit may produce better race detection, its effect on overhead is much clearer. From Figure 5, it is obvious that, *DC* with a longer time limit incurred a larger overhead. In our experiment, the first time limit is 15 ms and the second one is 30 ms (which is twofold of the first one). From the equations of trendlines in Figure 5, we could observe that, for *DC*, the overhead increasing factor (i.e., 173.46) at time limit of 30 ms is nearly two times of the former (i.e., 92.529).

However, from Figure 5 for *CR*, there is no obvious difference between the two time limits as the two trendlines almost overlap with increasing sampling rate. In detail, CR_{30} incurred a slightly less overhead. This is reasonable as with a longer time limit, (1) the increased time is not added into the overhead of *CR*; (2) however, the total number of sampled accesses could be reduced, resulting in less maintenance overhead. That is, an increasing time limit has no bad effect on the overhead of *CR*. This further provides flexibility for developers to set a time limit according to their programs without any worry on overhead increase.

5.5 Threats to Validity

Our benchmarks are Java programs. The JVM contains other threads accessing application memory locations. Although we have carefully compared whether two accesses were both from application threads, some scenarios might be ignored in our implementation. A more careful implementation may produce more precise results. Besides, hardware breakpoints can only be accessed within (Linux) kernel space. So we adopted the Netlink communication approach between user space and kernel space. Other communication approaches may produce different performance results that may also affect the effectiveness of *CRSampler* and *DataCollider* in the evaluation.

6. RELATED WORK

Concurrency bugs widely exist in multithreaded programs, including data races [12][19], atomicity violations [32][49], and deadlocks [15]. Both static techniques [26][37][41][51] and dynamic techniques [19][40][45][48][54] aim to detect data races. Static ones [51][41] are able to analyze a whole program but are imprecise due to lack of runtime information. Dynamic ones detect data races from execution traces. They either rely on the strict locking discipline (i.e., lockset) [45][47][57] or the relatively precise happens-before relation [19][40] (including its improvement [7][44] [50][52]). However, dynamic detection usually incurs heavy overhead [13][14][19]. Existing sampling techniques aim to detect races at user sites by incurring much lower overhead, which is the focus of this paper.

Systematic scheduling techniques such as model checking [53] [36], are in theory able to exhaustively execute every schedule to achieve certain coverage [30]. However, due to the state explosion problem, enumerating each schedule is not practical for real-world programs, even with reduction techniques [20]. *Chess* [36] 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 [10][21] tool to mine thread interleaving so as to expose unknown concurrency bugs. *PCT* [11][38] 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 (e.g., eclipse in our experiment).

RVPredict [24] 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* [18] 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 [42][46]. Both *RVPredict* and *Racageddon* have to solve scheduling constraints for each predicted race, which may fail. A recent work *DrFinder* [12] tries to 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% [12]).

RaceMob [27] 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. *CCI* [25] 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% [25]) although it targets on lightweight sampling. *Carisma* [59] improves *Pacer* by further sampling memory locations allocated at the same program location for Java program. *Carisma* could be integrated into *CRSampler* to improve its effectiveness.

ReCBuLC [58] also adopts thread-local clocks (time stamps) to reproduce concurrency bugs. Unlike *CRSampler*, *ReCBuLC* requires concrete objects and may still incur large overhead if applied to race sampling.

Recently, race detection has been extended to even-driven applications [34][23][22], concurrent library invocations [16], and modified program codes [56]. *CRSampler* could also be adapted to detect these races. We leave it as future work.

7. CONCLUSION

Existing sampling techniques for race detection still incur high overhead, even with 0% sampling rates, and/or detect races only when they occur by delaying program executions. We have proposed a new data race definition (i.e., clock races) for race sampling purpose. Detection of clock races avoids $O(n)$ operations and concrete synchronization objects, which hence incurs a much lower overhead. We also proposed *CRSampler* to sample clock races via hardware support. The experiment on six benchmarks confirms that *CRSampler* is both efficient and effective on race detection via sampling. At 1% sampling rate, it only incurs nearly 5% overhead, indicating that *CRSampler* is suitable to be deployed at user site for race detections.

8. ACKNOWLEDGEMENT

We thank anonymous reviewers for their invaluable comments and suggestions on improving this work. This work is supported in part by National 973 program of China (2014CB340702), and National Natural Science Foundation of China (NSFC) (grant No. 61502465, 91418206, and 61572481).

9. REFERENCE

[1] Netlink communication between Linux user space and kernel space. http://man7.org/linux/man-pages/man7/netlink.7.html

[2] Jikes Research Archive. http://www.jikesrvm.org/Resources/ResearchArchive

[3] Jikes RVM 3.1.3. http://jikesrvm.org

[4] 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.

[5] 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.

[6] S. Biswas, M. Zhang, and M.D. Bond. Lightweight data race detection for production runs. *Ohio State CSE Technical Report #OSU-CISRC-1/15-TR01*, January 2015. 23 pages, available at: http://web.cse.ohio-state.edu/~mikebond/litecollider-tr.pdf

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

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

[9] 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.

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

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

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

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

[14] Y. Cai and W.K. Chan. Lock trace reduction for multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems* (*TPDS*), 24(12): 2407−2417, 2013.

[15] Y. Cai and W.K. Chan. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* (*TSE*), 40(3), 266–281, 2014.

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

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

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

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

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

[21] 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.

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

[23] 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.

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

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

[26] 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.

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

[28] P. Krishnan. Hardware Breakpoint (or watchpoint) usage in Linux Kernel. IBM Linux Technology Center, Canada, July 2009.

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

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

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

[32] 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.

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

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

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

[36] 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.

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

[38] 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.

[39] 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.

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

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

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

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

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

[45] 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.

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

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

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

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

[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] 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.

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

[58] X. Yuan, C. Wu, Z. Wang, J. Li, P.C. Yew, J. Huang, X. Feng, Y. Lan, Y. Chen, and Y. Guan. ReCBuLC: reproducing concurrency bugs using local clocks. In *Proc. ICSE*, 824–834, 2015.

[59] 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.

[60] 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.