Computing and Informatics, Vol. 37, 2018, 186–212, doi: 10.4149/cai_2018_1_186

HYBRID DATA RACE DETECTION FOR MULTICORE SOFTWARE

Alper Sen, Onder Kalaci

Department of Computer Engineering
Bogazici University, Turkey
e-mail: {`alper.sen, onder.kalaci`}`@boun.edu.tr`

Abstract. Multithreaded programs are prone to concurrency errors such as deadlocks, race conditions and atomicity violations. These errors are notoriously difficult to detect due to the non-deterministic nature of concurrent software running on multicore hardware. Data races result from the concurrent access of shared data by multiple threads and can result in unexpected program behaviors. Main dynamic data race detection techniques in the literature are happens-before and lockset algorithms which suffer from high execution time and memory overhead, miss many data races or produce a high number of false alarms. Our goal is to improve the performance of dynamic data race detection, while at the same time improving its accuracy by generating fewer false alarms. We develop a hybrid data race detection algorithm that is a combination of the happens-before and lockset algorithms in a tool. Rather than focusing on individual memory accesses by each thread, we focus on sequence of memory accesses by each thread, called a segment. This allows us to improve the performance of data race detection. We implement several optimizations on our hybrid data race detector and compare our technique with traditional happens-before and lockset detectors. The experiments are performed with C/C++ multithreaded benchmarks using Pthreads library from PARSEC suite and large applications such as Apache web server. Our experiments showed that our hybrid detector is 15 % faster than the happens-before detector and produces 50 % less potential data races than the lockset detector. Ultimately, a hybrid data race detector can improve the performance and accuracy of data race detection, enhancing its usability in practice.

Keywords: Software testing and debugging, multithreaded programs, data race, concurrency, happens-before, lockset

Mathematics Subject Classification 2010: 68-N19

1 INTRODUCTION

Multicore processors provide high computation power, however, in order to utilize the increased power, concurrent software must be designed and written. Concurrency is achieved by multithreading in many systems. The interleaving of multiple threads can result in concurrency bugs which are hard to reproduce.

Data race is a well-known concurrency problem which is defined as two threads accessing a single memory address where at least one access is write and there is no appropriate synchronization among the accesses [19]. There have been many works in the past for detection of data races in multithreaded programs. The prior work is divided into two broad categories, which are static data race detection [28, 22, 3] and dynamic data race detection [13, 21, 2, 20, 15, 31].

Static data race detectors generate all possible thread interleavings and data races are searched among these interleavings. This approach is often infeasible due to state space explosion problem [12]. Moreover, they end up with many false positives due to the interleavings that are not possible to happen with any input data. Typically, the number of false positives is more than the number of real races [18]. *Dynamic data race detectors* observe memory accesses while an application is running. This approach is scalable to real life programs. The deficiency of dynamic analysis is that they consider an execution with a single input data, which limits the coverage of the analysis. Due to limited coverage, dynamic detectors miss some of the potential data races. In other words, dynamic detectors produce false negatives. In order to overcome this problem, dynamic analysis must be repeatedly executed with different inputs. Due to its scalability and potentially lower false positive rates, dynamic data race detection is the most commonly used method of data race detection.

There are two state-of-the-art algorithms used in dynamic data race detection, *lockset* and *happens-before* data race detection algorithms. Lockset algorithm [25] checks whether two threads access a shared variable while holding a common lock or not. For each shared memory address, the algorithm maintains a candidate lockset. Lockset detectors produce many false positives. In other words, some of the data races detected by the detectors are not real races. The main source of the false positives is that the detectors ignore all the synchronization operations except for locks. For instance, the detectors produce false positives for every shared memory access where the synchronization among accesses is generated by condition variables. Happens-before data race detection algorithm is based on Lamport's happens-before relation [13]. Happens-before relation defines a partial order among the events generated during the execution of a program in a distributed system. This relation has been extended for applications using shared memory as well. Happens-before data race detection algorithms utilize vector clocks for maintaining the happens-before relation [21, 15]. These detectors do not produce false positives, however they may miss real races (false negative). Happens-before based detectors suffer from high execution time and memory overhead. The size of each vector clock is proportional to the number of threads count in the program. On the contrary, lockset based detectors are scalable and can be implemented with a low overhead.

There has been prior work [20, 26, 11, 21] for combining the benefits of lockset and happens-before detectors in a hybrid data race detector that has good performance and few false positives. We develop a hybrid approach in this paper as well. Almost all dynamic data race detection algorithms [2, 20, 21, 12, 15, 5] detect potential data races by tracking accesses on each memory address during the execution. This can result in memory overhead, which is crucial when working with large programs. Our algorithm instead detects potential races by tracking accesses on each segment [26], where a segment is formed by consecutive memory accesses of a single thread. No synchronization operation is allowed inside a segment, thus, all the memory accesses use the same vector clocks and the same locks. Moreover, as we later show in Table 2, for most of the applications, the number of segments is much less than the number of memory addresses accessed. This observation allows us to reduce memory overhead.

We propose four optimizations on our segment based hybrid algorithm. The first optimization is based on the observation that vector clock values of a segment does not change after it is assigned. Thus, we added a limited vector clock cache for the vector clocks of segments. The second optimization is based on exploiting the same memory accesses inside a segment. If a memory address is accessed more than once inside a segment, the second and subsequent accesses have no effect on detecting the potential data races. The third optimization is based on the active number of segments. We define a maximum number of active segments, if that is exceeded, we start discarding older segments. Although this may lead to missing some of the potential data races, performance increase is considerable in many cases. Our last optimization is based on sampling a user given percentage of memory accesses during analysis.

We implemented our hybrid detector and our optimizations using PIN Dynamic Binary Instrumentation (DBI) tool [14]. This tool allows us to work with binaries of applications rather than their source code, which may be crucial in commercial settings. In order to compare our hybrid detector we also implemented using PIN a lockset based detector (Eraser [25]) and a happens-before based detector (*DJIT+* [21]). We performed experiments on 8 different applications from PARSEC benchmark suite [1], Apache httpd web server [9] and a parallel compression tool pbzip2 [6]. All benchmarks are written in C/C++ using Pthreads library. Our experiments showed that our hybrid detector is 15 % faster than happens-before detector and produces 50 % less potential data races than lockset detector.

The main contribution of this work can be summarized as follows:

- We develop a segment-based hybrid dynamic data race detector and present a formal treatment of the concepts and our algorithms.
- We propose four different optimizations on the hybrid algorithm. Two of these optimizations increase the performance of data race detection without sacrificing the precision. The remaining two optimizations provide a trade-off between

the number of potential data races detected and the performance of data race detection.

- We implement our techniques in a tool and compare with traditional dynamic data race algorithms on large multithreaded benchmarks.

The rest of the paper is organized as follows. Section 2 gives background on our multithreaded program model and happens-before relation. We describe dynamic data race detection algorithms in Section 3. In Section 4, we present our segment based hybrid data race detection algorithm and describe several optimizations on this algorithm in Section 5. We discuss experimental results in Section 6, which is followed by related work and conclusions.

2 BACKGROUND

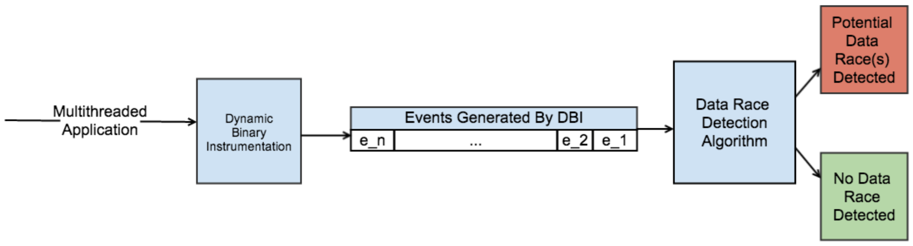
Multithreaded Application
Dynamic Binary Instrumentation
Events Generated By DBI
e_n
...
e_2
e_1
Data Race Detection Algorithm
Potential Data Race(s) Detected
No Data Race Detected

Figure 1. Overview of our dynamic data race detection

In Figure 1, an overview of our dynamic data race detection is displayed. A dynamic binary instrumentation tool instruments a multithreaded application and when this instrumented program is executed it generates events, which are input to the data race detection algorithm. Then, the algorithm decides whether the application has potential data race(s) or not.

2.1 Multithreaded Program Model

In this work, we consider multithreaded C/C++ applications that use the Pthreads library [23]. A multithreaded program consists of threads, memory addresses, and synchronization objects such as locks, condition variables, barriers, and semaphores.

During the execution of an instrumented program, a sequence of atomic operations (events), denoted by $e_x, \dots, e_y$, are generated by each thread. We utilize the following types of events in our data race detection algorithms, similar to earlier works [20, 5, 26].

- $READ(x, t_i)$: Memory address x is read by thread t_i.
- $WRITE(x, t_i)$: Memory address x is written by thread t_i.

- *ACCESS*(x, t_i): Either *READ*(x, t_i) or *WRITE*(x, t_i).
- *WR_LOCK*(l, t_i): Lock l is acquired write-held by thread t_i.
- *RD_LOCK*(l, t_i): Lock l is acquired read-held by thread t_i.
- *LOCK*(l, t_i): Either *WR_LOCK*(l, t_i) or *RD_LOCK*(l, t_i).
- *UNLOCK*(l, t_i): Lock l is released by thread t_i. This is also composed of read and write unlock operations.
- *SIGNAL*(cv, t_i): Unblock at least one of the threads that are blocked on the condition variable cv.
- *SIGNAL_ALL*(cv, t_i): Unblock all threads currently blocked on the condition variable cv.
- *WAIT*(cv, t_i): Thread t_i blocks on a condition variable cv.

We use the notion of synchronization points to generate the causality relationship among events. The following pair of corresponding events constitute the start (left) and the end (right) of synchronization points, denoted by *SYNCH_POINTS*: (*UNLOCK*(l, t_i), *LOCK*(l, t_j)), (*SIGNAL*(cv, t_i), *WAIT*(cv, t_j)), (*SIGNAL_ALL*(cv, t_i), *WAIT*(cv, t_j)). Note that similar synchronization points can be defined for semaphores, barriers as well as thread creation and exit events.

2.2 Happens-Before Relation and Vector Clocks

There exist several techniques for tracking the concurrency information or the dependencies between events. Lamport's happened-before relation [13], which is a partial order relation, is used for capturing ordering between events generated during the execution of a concurrent program. More formally, the happens-before relation ($\rightarrow$) among two events e_x and e_y is denoted as $(e_x \rightarrow e_y)$ and is the smallest transitive relation that satisfies the following properties (where $x \neq y$ and $i \neq j$) [21, 5]:

- Program Order: $(e_x \in t_i \wedge e_y \in t_i) \wedge$ (e_x is executed before e_y in t_i),
- Synchronization Order: $(e_x \in t_i \wedge e_y \in t_j) \wedge (i \neq j) \wedge$($e_x$ and e_y is a pair of events from *SYNC_POINTS*),
- Transitivity: $(e_x \rightarrow e_z) \wedge (e_z \rightarrow e_y)$.

Two events, e_x and e_y are *concurrent* ($\|$) if neither of them happens-before the other, that is, $e_x \parallel e_y \Leftrightarrow (\neg(e_x \rightarrow e_y) \wedge \neg(e_y \rightarrow e_x))$.

Vector clocks [4, 17] are used to capture the happens-before relation among the events generated during program execution. A vector clock assigns timestamps to events such that the partial order relation between events can be determined by using the timestamps. A *vector clock*, VC, consists of a vector of n integers where n is the total number of threads in the execution. VC_i identifies the vector clock of thread t_i and $VC_i[j]$ holds the logical time of thread t_j known by thread t_i. Initially, $VC_i[j] = 0$, for $i \neq j$, and $VC_i[i] = 1$. A thread increments its own component of the vector clock after each event. For certain events that will be described in the next section, it updates its vector clock by taking a component wise maximum with the vector clock included in the message.

Below we describe these common vector clock operations:

- *INIT*(VC_i): Initialize a Vector Clock, $VC_i[j] = 1$, for $i == j$ and $VC_i[j] = 0$, for $i \neq j$,
- *INCREMENT*(VC_i): Increment a Vector Clock, $VC_i[i] = VC_i[i] + 1$,
- *RECV*(VC_i, VC_j): Receive Vector Clock, $VC_i[k] = \max(VC_i[k], VC_j[k])$, $\forall k \in \{1, \ldots, n\}$,
- Compare Two Vector Clocks: $VC_i < VC_j$ is true, if $\forall k \in \{1, \ldots, n\} : VC_i[k] \leq VC_j[k] \wedge \exists k \in \{1, \ldots, n\} : VC_i[k] < VC_j[k]$.

We say that $e_x \rightarrow e_y$ iff $e_x.VC < e_y.VC$. Hence, vector clocks precisely capture the happens-before relation.

A sample execution of the vector clock algorithm is given in Figure 2, where the tuples in brackets represent the vector clocks. In the example, event (action) s happened-before t since $[1, 0, 0] < [2, 1, 3]$, where $v_i < v_j$ if all elements of v_i are less than or equal to the corresponding elements of v_j and at least one element of v_i is strictly less than the corresponding element of v_j. Whereas u is concurrent with t since their vector clocks are not comparable.

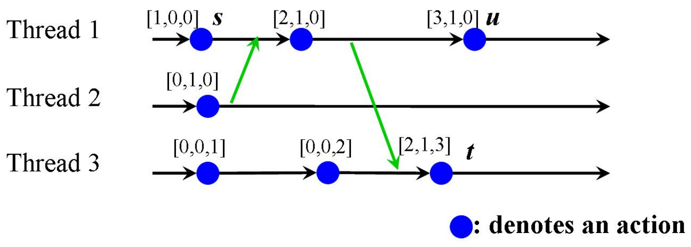

Figure 2. Happens-before relation and vector clocks

3 DATA RACE DETECTION ALGORITHMS

We briefly describe two state-of-the-art algorithms in data race detection, namely happens-before algorithm and lockset algorithm.

3.1 Happens-Before Data Race Detection

Happens-before data race detection algorithms check whether concurrent accesses by multiple threads to the same memory address are possible, if so they return a race warning. These detectors do not produce false positives but they can produce false negatives.

We describe *DJIT*$^+$ [21] algorithm as the happens-before algorithm. The algorithm maintains a vector clock for each thread t_i, denoted by VC_i, which is initialized at startup. A vector clock is kept for each synchronization object s, denoted by $s.VC$, which is initialized to all zeros at startup. For each left event of a *SYNC_POINTS*, such as *UNLOCK*(s, t_i) the following operations take place, $s.VC = VC_i$ and $VC_i =$ *INCREMENT*(VC_i). Similarly, for each right event of a *SYNC_POINTS*, such as *LOCK*(s, t_i), the following operation takes place, $VC_i =$ *RECV*$(VC_i, s.VC)$. For example, in Figure 2, in Thread 1, one can think of having an *UNLOCK* event after the second event with vector clock $[2, 1, 0]$ and the corresponding *LOCK* event occurs on Thread 3 changing the vector clock to $[2, 1, 3]$. Two vector clocks are kept for each memory address x, denoted by $x_r.VC$ and $x_w.VC$, which are used to keep track of the last read and the last write to x by each thread, respectively.

In Algorithm 1, we display the race detection portion of the happens-before based data race detection algorithm. For each read of x by t_i in line 1, the algorithm checks whether $x_w.VC$ happens-before VC_i in line 3. If not, a race is detected in line 12. Similarly, for each write of x by t_i in line 6, the algorithm checks whether each of $x_w.VC$ and $x_r.VC$ happens-before VC_i. If one of them does not, then the algorithm concludes a potential data race in line 12. Note that by exploiting the transitivity of the happens-before relation, the algorithm can decide on potential race conditions by only comparing the vector clock of the last read and write with the vector clock of the current access.

Algorithm 1 Happens-Before based data race detection algorithm

Input: Thread t_i generates a memory access event on address x, *ACCESS*(x, t_i)
Output: Potential data race detected or not

```
1:  if $ACCESS(x, t_i) == READ(x, t_i)$ then
2:      $x_r.VC[i] = VC_i[i]$;
3:      if $x_w.VC < VC_i$ then
4:          return false;                                   ▷ No Race Found
5:      end if
6:  else if $ACCESS(x, t_i) == WRITE(x, t_i)$ then
7:      $x_w.VC[i] = VC_i[i]$;
8:      if $(x_w.VC < VC_i) \wedge (x_r.VC < VC_i)$ then
9:          return false;                                   ▷ No Race Found
10:     end if
11: end if
12: return true;                                            ▷ Race Found
```

3.2 Lockset Data Race Detection

Lockset data race detection algorithms check whether accesses by multiple threads to the same memory address can occur while threads are not holding a common lock, if so they return a race warning. These detectors do not produce false negatives but they can produce false positives for a given execution.

We describe Eraser [25] algorithm as the basic lockset algorithm and show it in Algorithm 2. For each shared memory address x, the algorithm maintains a candidate lockset, $CLS(x)$. The name *candidate* is given since the algorithm cannot determine which lock is intended for which memory address. Thus, via candidate locksets, the algorithm attempts to infer whether a shared memory address is protected by a unique lock throughout the execution. When a memory address is accessed for the first time during the execution, its candidate lockset is assigned to include all possible locks. Then, on each access in line 1, its candidate lockset is updated to its intersection with the lockset of the thread that is executing the access, $LS(t_i)$. This lock refinement step aims to find the unique locks that protect the variable during the execution. If the intersection ends up with an empty set in line 2, the algorithm concludes that there is a potential data race in line 3. Eraser lockset algorithm includes optimizations and we implemented the Eraser lockset algorithm that includes optimizations in this paper.

Algorithm 2 Lockset based data race detection algorithm

Input: Thread t_i generates a memory access event on address x
Output: Potential data race detected or not

```
1: $CLS(x) = CLS(x) \cap LS(t_i)$
2: if $CLS(x) == \{\}$ then
3:     return true;                                   ▷ Race Found
4: end if
5: return false;                                      ▷ No Race Found
```

3.3 Hybrid Data Race Detection

When the above two approaches are examined in terms of preciseness, lockset detectors can produce false positives, whereas happens-before based detectors can produce false negatives. In terms of performance, lockset detectors are scalable, whereas happens-before based detectors are not because happens-before detectors require a high memory and processing overhead.

A hybrid data race detector combines the lockset and happens-before approaches. A naive hybrid algorithm maintains two vector clocks and a lockset for each memory address. Such an approach may help reduce the number of false positives compared to a lockset detector. However, performance of the detector would be worse than a happens-before based detector. First, the memory requirements would be more than a happens-before detector since it also uses a lockset for each memory. Second, the computation overhead would increase since both vector clock comparisons and lockset calculations are needed.

4 SEGMENT BASED HYBRID DATA RACE DETECTION ALGORITHM

In order to combine the best of traditional race detectors, we developed a segment based hybrid algorithm that utilizes the concept of a *segment* to improve performance. Our segment based hybrid approach is based on Threadsanitizer [26] algorithm. We formalize this algorithm and extend it with several performance optimizations as discussed in the next chapter.

A segment seg_i of a thread t_i is a sequence of memory accesses of t_i, denoted by $\{e_i\}$, where the lockset and vector clock of the segment is the same as that of thread t_i. No function calls or synchronization operations are allowed inside a segment. The outcome of this is that all the events inside a segment are executed while the thread holds the same vector clock and the same locks. Thus, the vector clocks and the locksets could be kept on the granularity of segments instead of memory accesses. Since we observe that the total number segments is much less than the total number of memory addresses in an execution for most of the applications, this approach reduces the memory requirements and increases the performance considerably as shown in the experiments.

Our segment based hybrid algorithm maintains several data structures as shown in Table 1. Algorithm 3 shows how these data structures are updated during program execution. Note that a happens-before relation is established from a *SIGNAL* to a *WAIT* but not from an *UNLOCK* to a *LOCK* operation as in the case for happens-before algorithm. This is because the happens-before relation on lock operations ensures that the same lock is used by threads during memory access, however the lockset algorithm portion of the hybrid algorithm will consider this case.

For each memory access in line 17 of Algorithm 3, Algorithm 4 is executed. In Algorithm 4, first, the current segment of thread t_i that is executing the memory access is obtained as seg_i in line 1. Then, in line 2 for the write access, writer segment set of the memory address is updated so that it only includes segments that do not happen-before the current executing segment in line 3. Additionally, the current segment is added to the writer segment set of the memory address. Similarly, reader segment set of the memory address is updated so that it only includes segments that do not happen-before the current executing segment in line 4. It can be observed from the definition of concurrency that segments in write segment sets are pairwise concurrent, similarly for read segment sets. This is a useful performance optimization because accesses from non-concurrent segments, which are happens-before ordered, cannot cause a potential data race on a shared memory address.

Thread t_i	
Vector Clock	VC_i
Writer Lockset	$WRLS(t_i)$
Reader Lockset	$RDLS(t_i)$
Segment s_i	
Vector Clock	$seg_i.VC$
Writer Lockset	$WRLS(seg_i)$
Reader Lockset	$RDLS(seg_i)$
Condition Variable cv	
Vector Clock	$cv.VC$
Memory Address x	
Writer Segment Set	$WRST_x$.
Reader Segment Set	$RDST_x$.

Table 1. Segment-based hybrid algorithm data structures

It can be seen from Algorithm 4 that read and write accesses update segment sets differently. On write accesses, both writer and reader segment sets are updated (line 3 and 4), whereas, on read accesses, only reader segment set is updated (line 6). The reason is as follows, on write accesses, it is safe to remove any of the read accesses from $RDST_x$. Remember that, $RDST_x$ consists of concurrent segments where x is read. Since there is no read-read type of data race, removing any segment from $RDST_x$ does not lead to missing any races. On the contrary, on read accesses it is not safe to remove any segment from $WRST_x$. The reason is that, it may lead to missing a write-write data race because the removed segment might have a potential race with one of the prospective segments in the same set. The outcome of this is that all segments within any segment set are concurrent with each other. However, not all segments in $RDST_x$ are concurrent with all segments in $WRST_x$, which is handled while checking race among $WRST_x$ and $RDST_x$.

Algorithm 5 describes the data race checking between segment sets *checkRace(WRST,RDST)*, which is called at line 8 in Algorithm 4. The algorithm checks for common locks among concurrent segments such that one segment is a writer segment and the other is either a reader or a writer segment since there is no read-read data race. If the segments are a writer and a reader segment then the algorithm also makes sure that there is no happens-before relation from the writer to the reader segment as shown in line 8. We know that a happens-before relation cannot exist from the reader to the writer as these have been removed during the reader segment set update in Algorithm 4. If the algorithm could not find a common lock among concurrent segments (lines 4 and 9), it returns true indicating a data race. Note that if any two segments are ordered by the happens-before relation, they are not checked for data race, which is similar to the happens-before algorithm. Then, if two segments are concurrent then they are checked for holding a common lock, which is similar to the lockset algorithm.

Algorithm 3 Segment based hybrid data race detection algorithm instrumentation for each operation

Input: Thread t_i generates an operation *op*
Output: Instrumented program

```
1: if $op$ is thread creation then
2:     $INIT(VC_i)$; $WRLS(t_i) = RDLS(t_i) = \{\}$
3: else if $op$ is segment creation then
4:     $seg_i.VC = VC_i$; $WRLS(seg_i) = WRLS(t_i)$; $RDLS(seg_i) = RDLS(t_i)$
5: else if $op$ is $WAIT(cv, t_i)$ then
6:     $RECV(VC_i, cv.VC)$
7: else if $op$ is $SIGNAL(cv, t_i)$ then
8:     $RECV(cv.VC, VC_i)$; $INCREMENT(VC_i)$
9: else if $op$ is $WR\_LOCK(l, t_i)$ then
10:    $WRLS(t_i) = WRLS(t_i) \cup \{l\}$
11: else if $op$ is $WR\_UNLOCK(l, t_i)$ then
12:    $WRLS(t_i) = WRLS(t_i) \setminus \{l\}$
13: else if $op$ is $RD\_LOCK(l, t_i)$ then
14:    $RDLS(t_i) = RDLS(t_i) \cup \{l\}$
15: else if $op$ is $RD\_UNLOCK(l, t_i)$ then
16:    $RDLS(t_i) = RDLS(t_i) \setminus \{l\}$
17: else if $op$ is $WRITE(x, t_i)$ or $READ(x, t_i)$ then
18:    $ACCESS(x, t_i)$
19: end if
```

4.1 Segment Based Hybrid Algorithm Example

We now show an execution of our segment-based hybrid approach. A sample execution of the events belonging to two different threads and the state of data structures for each line of the execution is described in Figure 3. For simplicity, we do not display $RDST_x$ and $RDLS$ of threads.

Algorithm 4 Segment-based hybrid data race detection algorithm – $ACCESS(x, t_i)$

Input: Thread t_i generates a memory access event on address x
Output: Potential data race detected or not

```
1: $seg_i = CurrentSegment(t_i)$
2: if $ACCESS(x, t_i) == WRITE(x, t_i)$ then
3:     $WRST_x = \{seg_x \mid seg_x \in WRST_x \wedge \neg(seg_x.VC \rightarrow seg_i.VC)\} \cup seg_i$
4:     $RDST_x = \{seg_x \mid seg_x \in RDST_x \wedge \neg(seg_x.VC \rightarrow seg_i.VC)\}$
5: else if $ACCESS(x, t_i) == READ(x, t_i)$ then
6:     $RDST_x = \{seg_x \mid seg_x \in RDST_x \wedge \neg(seg_x.VC \rightarrow seg_i.VC)\} \cup seg_i$
7: end if
8: $checkRace(WRST_x, RDST_x)$
```

Algorithm 5 Segment-based hybrid data race detection algorithm – *checkRace*(*WRST*, *RDST*)

Input: Writer Segment Set *WRST*, Reader Segment Set *RDST*
Output: Potential data race detected or not

```
for $wr_1 \in WRST$ do
    for $wr_2 \in WRST$ do
        if $WRLS(wr_1) \cap WRLS(wr_2) = \{\}$ then
            return true;                                  ▷ Write-Write Race Found
        end if
    end for
    for $rd \in RDST$ do
        if $\neg(wr_1.VC \rightarrow rd.VC) \wedge (WRLS(wr_1) \cap RDLS(rd) = \{\})$ then
            return true;                                  ▷ Read-Write Race Found
        end if
    end for
end for
return false;                                             ▷ No Race Found
```

After t_1 acquires lock l_1, a new segment s_1 is initialized. Memory address x is written in s_1, thus, s_1 is added to $WRST_x$ in ACCESS algorithm. s_1 is finalized when l_1 is released by t_1. Then, after t_2 acquires l_1, segment s_2 is initialized. When x is written in s_2, it is added to $WRST_x$ in ACCESS algorithm since s_1 and s_2 are concurrent. Since both s_1 and s_2 have a common lock, which is l_1, checkRace algorithm does not produce a data race alarm. The execution ends up with no race, which is a *True Negative*.

5 OPTIMIZATIONS ON SEGMENT-BASED HYBRID ALGORITHM

In this section, we are going to discuss four optimizations that we proposed and implemented on segment-based hybrid algorithm.

5.1 Optimization 1: Storing Vector Clock Comparison History Cache

We observed that maintaining a vector clock comparison history cache for the previously calculated vector clock comparisons can increase the performance of data race detection. This is motivated by the fact that multiple memory accesses can belong to the same segment and since all these memory accesses have the same vector clock as the segment that they belong to, there may be an excessive number of comparison operations between the same vector clocks. For each vector clock, we keep a list that holds the previous comparisons of the vector clock with other vector clocks. Since the same vector clock could be accessed concurrently, a lock is required for accessing the list. These two requirements increase the total memory requirement of data race detection but improves the performance as shown in the

Line	Thread 1	Thread 2
1	$WR\_LOCK(l_1, t_1)$	
2	$WRITE(x, t_1) \in s_1$	
3	$WR\_UNLOCK(l_1, t_1)$	
4		$WR\_LOCK(l_1, t_2)$
5		$WRITE(x, t_2) \in s_2$
6		$WR\_UNLOCK(l_1, t_2)$

Line	VC_1	$WRLS(t_1)$	VC_2	$WRLS(t_2)$	$WRST_x$
0	$\langle 1,0\rangle$	$\{\}$	$\langle 0,1\rangle$	$\{\}$	$\{\}$
1	$\langle 1,0\rangle$	$\{l_1\}$	$\langle 0,1\rangle$	$\{\}$	$\{\}$
2	$\langle 1,0\rangle$	$\{l_1\}$	$\langle 0,1\rangle$	$\{\}$	$\{s_1\}$
3	$\langle 1,0\rangle$	$\{\}$	$\langle 0,1\rangle$	$\{\}$	$\{s_1\}$
4	$\langle 1,0\rangle$	$\{\}$	$\langle 0,1\rangle$	$\{l_1\}$	$\{s_1\}$
5	$\langle 1,0\rangle$	$\{\}$	$\langle 0,1\rangle$	$\{l_1\}$	$\{s_1, s_2\}$
6	$\langle 1,0\rangle$	$\{\}$	$\langle 0,1\rangle$	$\{\}$	$\{s_1, s_2\}$

Figure 3. A program execution with segment-based hybrid algorithm and update of data structures during the execution

experiments. We also only cache vector clocks of segments since after initialization their values do not change, whereas vector clocks of threads and synchronization objects can change during execution, increasing computation overhead.

Our optimization is formalized in Algorithm 6. For each vector clock VC_i, a list $vc_i\_prev\_comparisons$ and a lock vc_i_lock is required. On each comparison, first the local cache $vc_i\_prev\_comparisons$ is searched. If the result is already there, there is no need to make comparison. Otherwise, the comparison is done and the result is added to $vc_i\_prev\_comparisons$.

The algorithmic complexity of this optimization is O(n), where n is the vector clock history size.

Algorithm 6 Optimization 1

Input: Vector Clocks vc_1 and vc_2

Output: Comparison of vc_1 and vc_2

```
1: $LOCK(vc_1\_lock)$;
2: $result = vc_1\_prev\_comparisons(vc_2)$;
3: if $result == None$ then
4:     $result = compare(vc_1, vc_2)$
5:     $vc_1\_prev\_comparisons(vc_2) = result$
6: end if
7: $UNLOCK(vc_1\_lock)$;
8: return $result$;
```

5.2 Optimization 2: Multiple Accesses of a Single Variable in a Segment

This optimization exploits the fact that repeated memory accesses of the same type and same variable belonging to the same segment do not make a difference in terms of race detection. This is because the earlier of the accesses will have already been added to the writer or reader segment set of the variable and since the later access has the same vector clock and locksets it will not have any impact. This optimization can quickly be added to the *ACCESS* algorithm as shown in Algorithm 7. There is no extra memory requirement for implementing this optimization. The only requirement is the increased CPU utilization. The algorithmic complexity of this optimization is $O(n)$, where n is the average number of segments in the segment sets.

Algorithm 7 Optimization 2 – updated Algorithm *ACCESS*

Input: Thread t_i generates a memory access event on address x

Output: Potential data race detected or not

```
1: $seg_i = CurrentSegment(t_i)$
2: if $ACCESS(x, t_i) == WRITE(x, t_i)$ then
3:     if $seg_i \notin WRST_x$ then
4:         $WRST_x = \{seg_x \mid seg_x \in WRST_x \wedge \neg(seg_x.VC \rightarrow seg_i.VC)\} \cup seg_i$
5:         $RDST_x = \{seg_x \mid seg_x \in RDST_x \wedge \neg(seg_x.VC \rightarrow seg_i.VC)\}$
6:     end if
7: else if $ACCESS(x, t_i) == READ(x, t_i)$ then
8:     if $seg_i \notin RDST_x$ then
9:         $RDST_x = \{seg_x \mid seg_x \in RDST_x \wedge \neg(seg_x.VC \rightarrow seg_i.VC)\} \cup seg_i$
10:    end if
11: end if
12: $checkRace(WRST_x, RDST_x)$
```

5.3 Optimization 3: Limiting The Total Number of Segments

In segment-based hybrid approach, many of the segments do not cause potential data races. For instance, most segments do not access shared variables. Or even if some shared variables are accessed, many of the segments are happens-before ordered or protected by a common lock. Also, we make the observation that in general segments that are closer to each other in terms of execution time are more likely to cause race conditions. Therefore, discarding some of the segments may increase the performance while preserving the number of potential data races found.

In our implementation, we define a limit that identifies the maximum number of segments that can be utilized by our segment-based hybrid approach. Whenever the total number of segments exceeds the maximum number, we discard a previously created segment and remove that segment from any of the segment sets that it is present. We utilize a FIFO queue for choosing which segment to be discarded. Although limiting the maximum number of segments increases the performance of data race detection, it may lead to missing some of the potential data races since the discarded segment can be a part of a potential data race in the execution.

It is important that the limit should be determined exclusively for each application. In our experiments, we choose the maximum number relative to the total segment count in the original execution.

In the worst case, the algorithmic complexity of this optimization is $O(n)$, where n is the total number of segment sets in the execution.

5.4 Optimization 4: Proportional Detection of Data Races

It is widely accepted that dynamic data race detection requires excessive processing power and memory. This hinders the utilization of dynamic data race detection tools in real deployed environments. PACER [2] proposes proportional detection of data races. It makes a proportionality guarantee by detecting data races at a rate equal to the sampling rate. Our sampling approach takes advantage of this observation but it is simpler than PACER with no proportionality guarantee. On each memory access operation in Algorithm 3, according to the given sampling rate, we decide whether to call $ACCESS$ procedure or not. Specifically, a random integer is generated between 0 and 100. We use an equidistributed uniform pseudo-random number generator [16].

If the generated integer is smaller than the sampling rate, we execute the instrumentation for the memory access. Our experiments show that the percentage of instrumented memory accesses still converges to the sampling rate.

6 EXPERIMENTS

In this section, we describe our implementation and experiments on dynamic data race detection. We implemented three dynamic data race detection algorithms, lockset, happens-before, and segment-based hybrid algorithm. We also show the results of our optimizations described in the previous section. We performed experiments with eight multi-threaded applications from PARSEC benchmark suite [1]. We also used a parallel compression tool pbzip2 [6] and Apache httpd web server [9] as test cases. All the experiments were performed on a PC running Linux with a 4 cores CPU of 2.27 GHz and 32 GB of memory. We ran each experiment 10 times and averaged the results.

We implemented the detectors using PIN dynamic binary instrumentation (DBI) tool [14]. PIN allows us to work with binaries of applications rather than their source code, which may be crucial in commercial settings. In our implementation, we utilized the just-in-time compiler JIT mode of PIN. This allows instrumentation to be done at different granularities such as instruction, trace, image or routines. We used routine instrumentation for the synchronization function calls. For instance, a callback is inserted after pthread mutex lock function so that our hybrid algorithm updates the caller thread's writer lockset. For memory accesses we used trace instrumentation. We know that instruction instrumentation enables to insert one analysis call for every instruction executed. In fact, instruction instrumentation could be useful for inserting an analysis call for every read and write instruction so that data race detection algorithms' memory access algorithms could be executed. However, reducing the number of analysis calls results in efficient instrumentation. Therefore, instead of instruction instrumentation we prefer trace instrumentation. A trace is composed of a sequence of Basic Block (BBL). A BBL is a single entrance and single exit sequence of instructions. Thus, by trace instrumentation we firstly iterate over BBLs that forms the trace that is instrumented. Then, for each BBL we iterate over instructions and call a function for each read or write instruction that is executed. The number of analysis calls is reduced from the executed instruction count to the number of executed traces. We developed a tool, called Dyndatarace, that incorporates our solution and can be accessed online[1].

In our implementations, we utilized the same implementation for the common parts of different detectors as much as possible. For instance, we use the same implementation of vector clocks for both the hybrid algorithm and the happens-before algorithm. Similarly, we used the same implementation of locksets for both the hybrid algorithm and the lockset algorithm.

While potential data races are being searched, it is crucial to track the dynamic allocations and deallocations to prevent false alarms. In our implementations, we overcome this problem by tracking all memory allocations and deallocations. Whenever a deallocation is executed, all the shadow memory state that is kept for the corresponding allocation call is cleaned up.

Benchmark	Base Time (sec)	# Thread	# Memory Addr.	# Segment	Execution Time				Potential Data Races		
					Empty	Lockset	Hybrid	Happens-Before	Lockset	Hybrid	Happens-Before
×264	0.18	15	4 579 993	6 225	2.92×	13.80×	20.26×	25.20×	184	27	1
freqmine	0.27	15	18 525 725	1 350	2.42×	117.93×	332.41×	261.56×	76	23	7
vips	0.22	18	22 883 398	32 044	3.67×	103.69×	250.21×	315.43×	174	64	0
swaptions	0.14	4	47 837	320	1.91×	72.43×	134.98×	202.15×	0	0	0
bodytrack	0.18	5	5 803 300	8 702	2.72×	18.67×	31.24×	50.37×	47	5	4
fluidanimate	0.25	4	1 303 047	2 163 828	1.43×	89.42×	85.10×	113.52×	281	35	0
streamcluster	0.25	8	166 555	163 641	1.73×	75.28×	133.05×	178.56×	19	14	0
canneal	1.90	4	3 012 749	1 380	4.13×	16.41×	20.88×	26.71×	1	1	0
pbzip2	1.40	16	1 050 268	157 60	2.32×	41.92×	44.27×	50.30×	29	23	0
httpd	26.11	22	251 943	229 900	2.21×	13.32×	24.11×	26.75×	1777	1151	0
average	3.09	11.10	5 762 481	262 315	2.54×	56.29×	107.65×	125.06×	258	134	1

Table 2. Results for three race detectors

[1] `https://github.com/onderkalaci/dyndatarace`

In our experiments we compare the behavior of detectors in terms of execution overhead as well as the number of potential data races. Table 2 displays our experimental results. In the table we show the overhead (slowdown) of dynamic binary instrumentation over the uninstrumented execution in the column denoted by *Empty*, which does not perform any computations or analysis related to data race detection but the overhead of instrumenting memory accesses. We also display the slowdown of each data race detector with respect to the *Empty* implementation. On average, execution slow down is 125× for happens-before detector, 107× for hybrid detector and 56× for lockset detector. Memory requirements are 2 084 M for happens-before detector, 1 258 M for hybrid detector and 916 M for lockset detector. Hence, the slow downs are arranged as *Happens-before* > *Hybrid* > *Lockset* as expected. Lockset slowdown is less than half of the happens-before and hybrid. This is acceptable since heavy memory and execution overhead of vector clocks are not present in the lockset. On the average, happens-before is about 15 % slower than hybrid.

In terms of the number of data races detected by each algorithm, the results are again expected and are as follows, *Lockset* > *Hybrid* > *Happens-before*. On average, the potential number of data races detected by happens-before, hybrid and lockset detectors are 1, 134, and 258, respectively. As discussed in the previous sections, lockset detectors produce too many false positives. On the contrary, happens-before detectors do not produce any false positives, but they can miss some of the potential data races. Hybrid algorithm poses characteristics from both approaches.

We performed experiments to measure the impact of increasing number of threads on the number of data races. For some of the applications, we were able to change the number of the threads that run concurrently without changing the inputs. Such applications include *x264*, *freqmine*, *fluidanimate*, *canneal* and *pbzip2*. For these applications, we performed three different experiments with different number of threads as shown in Table 3. Our experiments show that although the execution time depends on the thread count, the total number of potential data races detected are not affected.

Application	Original Execution	Experiment-1	Experiment-2	Experiment-3
x264	15	18	20	32
freqmine	15	18	20	32
fluidanimate	4	6	8	12
canneal	4	6	8	12
pbzip2	16	12	20	32

Table 3. Number of thread counts for different experiments

6.1 Results of Optimizations on Segment-Based Hybrid Algorithm

In this section, we explore the experimental results of the optimizations that are applied to the segment-based hybrid algorithm.

Since our goal is to find the best performance due to optimizations, we explore a combination of them rather than exploring optimizations individually. We first check the combination of Optimization 1 and 2 since these optimizations do not affect the number of data races and the best combination parameters are searched. Then we add either one of Optimization 3 and 4 to these two, since Optimization 3 and 4 cannot be combined. The reason is that both Optimization 3 and 4 lead to false negatives and affect the performance. Therefore, it would be difficult to infer the effect of each optimization on results when they are combined. The presented execution time values are relative to the execution of segment-based hybrid algorithm with no optimization applied.

Application	Execution Time				
History Size	0	1	10	50	250
×264	0.92×	0.95×	0.88×	0.89×	0.99×
freqmine	0.98×	0.85×	0.86×	0.83×	0.84×
vips	0.98×	0.9×	0.86×	0.97×	0.97×
swaptions	1.16×	1.44×	1.42×	1.33×	1.42×
bodytrack	1.09×	0.99×	0.95×	0.98×	1.03×
fluidanimate	0.96×	0.78×	0.77×	0.78×	0.79×
streamcluster	0.91×	0.68×	0.68×	0.68×	0.68×
canneal	0.99×	0.95×	0.99×	0.94×	0.97×
pbzip2	1.02×	1.0×	0.99×	1.0×	0.98×
httpd	0.71×	0.61×	0.62×	0.67×	0.64×
average	0.97×	0.91×	0.90×	0.91×	0.93×

Table 4. Optimization 1 and Optimization 2

6.2 Optimization 1 and Optimization 2

Table 4 shows the performance change by the addition of Optimization 1 when Optimization 2 is always enabled, since Optimization 2 almost always improves performance. On the average, when the vector clock history size is set to 10, the maximum performance is achieved.

For some of the applications such as swaptions, Optimization 1 reduces the performance, independent of the history size. In this case, unsuccessful searches in the vector clock history utilize CPU so much that the gain of successful searches cannot compensate the unsuccessful ones.

6.3 Optimization 1, Optimization 2 and Optimization 3

Table 5 displays the effect of Optimization 3 when Optimizations 1 and 2 are applied with a vector clock history size of 10. The table shows the effect of limiting the total segment count on both the execution time and the number of potential data races.

Since each application executes a different number of segments during execution, the parameter that is altered is the percentage of number of segments with respect to all segments in the original execution, displayed in row Limit. For instance, when the total number of segments are limited to 50 % of the original execution for that application, the performance is increased 17 % while only 4 of the potential data races are missed. We observe that even when the limit is set to 4 % the number of data races remain similar to the original and the performance improves by 31 %. However, the execution time does not decrease in a linear fashion. A potential reason for this is that the operation of destroying a segment requires a lot computation, that is the destroyed segment must be removed from all of the segment sets that it is a member of.

Limit	100 %		50 %		32 %		18 %		12 %		4 %		1 %	
Application	Time	Race	Time	Race	Time	Race	Time	Race	Time	Race	Time	Race	Time	Race
×264	0.91×	27	0.90×	27	0.86×	27	0.86×	27	0.88×	27	0.80×	26	0.66×	4
freqmine	0.88×	23	0.41×	23	0.40×	23	0.36×	23	0.31×	23	0.29×	19	0.88×	19
vips	0.87×	64	0.88×	63	0.88×	59	0.87×	55	0.79×	53	0.77×	38	0.61×	0
swaptions	1.44×	0	1.40×	0	1.34×	0	0.98×	0	0.95×	0	0.87×	0	1.32×	0
bodytrack	0.97×	5	0.83×	3	0.73×	0	0.77×	0	0.80×	0	0.71×	0	0.64×	0
fluidanimate	0.81×	35	0.67×	34	0.68×	34	0.66×	34	0.61×	34	0.59×	33	0.79×	22
streamcluster	0.72×	14	0.61×	14	0.59×	14	0.57×	14	0.55×	14	0.44×	14	0.50	14
canneal	1.0×	1	1.01×	1	0.92×	1	0.90×	1	0.95×	1	0.91×	1	0.98×	1
pbzip2	0.99×	23	0.98×	23	1.00×	23	0.98×	20	0.98×	17	0.97×	17	0.96	0
httpd	0.68×	1 151	0.62×	1 121	0.58	1 113	0.61×	1 113	0.58×	1 097	0.57×	1 097	0.68×	56
average	0.93×	134	0.83×	130	0.80×	129	0.76×	128	0.74×	126	0.69×	124	0.80×	11

Table 5. Results of Optimization 1, Optimization 2 and Optimization 3. Limit between 100 %–1 %.

6.4 Optimization 1, Optimization 2, and Optimization 4

Table 6 displays the effect of Optimization 4 when Optimizations 1 and 2 are applied with a vector clock history size of 10. The table shows that on average execution times and the number of potential data races roughly converge to the sampling rate, until 16 % sampling rate. When the sampling rate is decreased further, due to the overhead of creating and managing segments, performance does not converge to the sampling rate. Furthermore, for *pbzip*2, the execution time is not converging to the sampling rate since this application makes too many I/O operations, which is a bottleneck.

6.5 Vector Clock Operation Performance

In this section, we compare the average number of vector clock operations per memory access between the happens-before and hybrid detectors. In the happens-before Algorithm 1, the number of vector clock comparisons is one or two depending on the access type as can be seen from lines 3 and 8, respectively. As the proportion of reads increases, the average number of vector clock comparisons per memory access is expected to decrease. In the segment-based hybrid approach, the number

Sample Rate	100 %		71 %		50 %		16 %		5 %		2 %	
Application	Time	Race	Time	Race	Time	Race	Time	Race	Time	Race	Time	Race
×264	0.92×	27	0.66×	23	0.49×	19	0.17×	12	0.11×	9	0.10×	3
freqmine	0.88×	23	0.21×	23	0.13×	23	0.04×	12	0.02×	11	0.06×	4
vips	0.87×	64	0.67×	53	0.49×	35	0.12×	17	0.07×	1	0.06×	0
swaptions	1.54×	0	0.74×	0	0.67×	0	0.23×	0	0.21×	0	0.19×	0
bodytrack	1.01×	5	0.91×	0	0.52×	1	0.16×	0	0.08×	0	0.07×	0
fluidanimate	0.82×	35	0.56×	28	0.44×	27	0.08×	12	0.06×	2	0.06×	2
streamcluster	0.74×	14	0.42×	13	0.33×	12	0.12×	5	0.04×	2	0.05×	0
canneal	1.02×	1	0.82×	1	0.65×	1	0.37×	0	0.28×	0	0.31×	0
pbzip2	1.04×	23	0.97×	13	0.81×	6	0.74×	1	0.72×	0	0.75×	0
httpd	0.71×	1 151	0.41×	797	0.29×	478	0.08×	111	0.12×	23	0.04×	15
average	0.96×	134	0.64×	95	0.48×	60	0.21×	17	0.17×	4	0.17×	2

Table 6. Results of Optimization 1, Optimization 2 and Optimization 4. Sample rate between 100 %–2 %.

of vector clock operations depends on the size of the segment set for each memory access which increases as the number of concurrent accesses to a memory address increases. The comparisons are done in two different points in segment-based hybrid algorithm in lines 3, 4, 6 of Algorithm 4 and line 8 of Algorithm 5.

Application	Hybrid	Happens-Before
x264	2.12	1.07
freqmine	14.87	1.16
vips	7.99	1.11
swaptations	4.78	1.04
bodytrack	2.85	1.13
fluidanimate	1.85	1.08
streamcluster	2.58	1.07
canneal	2.54	1.34
Firefox	0.34	1.07
pbzip2	0.99	1.07
httpd	22.80	1.37
average	5.70	1.13

Table 7. Average number of vector clock comparison per memory access

The number of vector clock operations cannot be determined statically. Therefore, to calculate the average number of vector clock operations in both algorithms, we create two variables (*m_access_cnt*, *vc_compare_cnt*) and on each memory access, we increment the value of *m_access_cnt* and on each vector clock comparison, we increment the value of *vc_compare_cnt*. Then, after the execution completes, we calculate the vector clock comparison per memory access, $vc_{avg} = vc\_compare\_cnt/m\_access\_cnt$. Table 7 shows vc_{avg} for both algorithms. As expected,

vc_{avg} is 1.13 for happens-before algorithm, whereas, for segment-based hybrid algorithm, vc_{avg} is 5.70. Therefore, we can conclude that on average, happens-before algorithm performs better than the hybrid segment-based algorithm in terms of the average number of vector clock operations per memory access.

6.6 Memory Performance

We show the memory requirements of lockset, happens-before and segment-based hybrid algorithms for each application in Table 8. The results show that hybrid detector's memory requirement is almost 40 % less than happens-before detector and 30 % more than lockset detector. These results are expected since utilization of vector clocks increases memory requirements remarkably. Happens-before detectors are entirely based on vector clocks, lockset detectors do not utilize vector clocks and hybrid detectors are partially based on vector clocks.

Application	Lockset	Hybrid	Happens-Before
x264	794	884	2 785
freqmine	1 444	2 201	3 873
vips	1 502	2 220	3 564
swaptations	304	574	744
bodytrack	752	1 408	2 062
fluidanimate	674	886	1 240
streamcluster	513	590	574
canneal	750	904	1 766
pbzip2	721	910	1 104
Firefox	1 066	1 271	1 971
httpd	1 562	1 990	3 244
average	916	1 258	2 084

Table 8. Memory requirements for data race detection algorithms (MB)

7 SUMMARY OF RESULTS

We proposed four different optimizations on our algorithm to improve performance. The first two improvements do not alter the number of data races that are detected in any of the benchmark applications. When the first two optimizations are combined they reduce the execution time by 10 %, if the vector clock comparison history cache size is set to 10. The last two improvements alter both the execution time and data races detected relative to the original application. For the third optimization, when the total number of segments is limited to 50 % of the number of segments in the original application, execution time decreases by 17 % where the number of potential races almost remains the same. For the last optimization, average execution times and the number of detected data races roughly converge to a user given sampling rate. We also performed experiments which show that the memory requirements of happens-before algorithm is more than segment-based hybrid algorithm, whereas, hybrid algorithm executes more vector clock comparisons on the average than the happens-before.

We showed that a hybrid race detector has several advantages over other types of race detectors as described above. However, due to the use of instrumentation there is a big slowdown for all detectors that can make it impractical for online execution purposes. We discuss potential improvements in future work.

8 RELATED WORK

Dynamic data race detection algorithms proposed in the literature are based on lockset approach, happens-before approach or combination of both approaches. In this section we explore these approaches and their differentiated variants. *Eraser* [25] is the pioneer of the lockset based detectors. Since lockset based detectors only recognize locks, the synchronization formed among threads by other primitives such as condition variables or barriers are ignored. Thus, false positives are inevitable with lockset detectors.

Happens-before detectors inspect data races by verifying the happens-before relation among memory accesses. The happens-before relation is represented by vector clocks in many happens-before based detectors such as *DJIT+* [21] and *LiteRace* [15]. Contrary to lockset detectors, happens-before detectors do not produce any false positives. However, they miss some potential data races. In other words, pure happens-before based detectors produce false negatives. *DJIT*$^+$ and other traditional happens-before detectors suffer from high execution time and memory overhead. The size of vector clocks are proportional to the number of threads in the system. Therefore, memory requirements and comparison complexity of vector clocks are proportional to the thread count. In order to overcome these performance issues several enhancements have been proposed. *FastTrack* [5] implements a scalar data structure, called *epoch*, consisting of two integers, and replaces vector clocks with epochs whenever possible. This replacement does not affect the precision of the happens-before algorithm. iFT [7] represents an improvement over the FastTrack method. It reduces the average runtime and memory overhead to 84 % and 37 %, respectively, of those of FastTrack.

PACER [2] proposes a sampling method on FastTrack algorithm. It makes a proportionality guarantee such that it detects potential data races at a rate equal to the sampling rate. *LiteRace* [15] is another happens-before data race detection algorithm that applies a different sampling approach than ours, and detects 70 % of data races by sampling only 2 % of memory accesses. This work is orthogonal to ours. With low sampling rates, Carisma [32] can detect race conditions also.

Hybrid data race detector algorithms combine happens-before and lockset detectors. One of the main purposes of these detectors is to solve the false positive problem of lockset approach and false negative problem of happens-before approach.

In [20], the combination of both approaches is implemented where their algorithm incurs a higher overhead than lockset detector, but the number of false positives is decreased. AccuLock [30] is a hybrid detection algorithm that combines FastTrack and a new Lockset analysis. *RaceTrack* [31] implements a hybrid adaptive algorithm that automatically pays more attention to the more suspicious code. The algorithm aims to increase the precision while decreasing the overhead. *Threadsanitizer* [26] and *Helgrind+* [11] decrease the execution overhead by inspecting data races among segments instead of memory addresses similar to ours. We implemented several other optimizations in this work that do not exist in those works and present a more formal treatment of segments, which was not done previously.

There are other approaches for race detection as well. Race detection can be considered as a safety verification problem for concurrent programs. Model checking is a formal verification technique that can find and prove the absence of races. Safety or liveness can be given in the form of temporal logics such as LTL or CTL. Model checking has been used in the context of race detection [8, 24]. However, due to the exponentiality of both the program path and the scheduling space model checking does not scale well to large programs. Runtime verification, similar to testing, deals only with observed execution of a program, whereas model checking considers all possible executions. Similar to model checking, in runtime verification temporal logic specifications can be used and one can generate trace reorderings under scheduling constraints to find bugs unseen in the observed execution [10].

In [27], the authors propose a new relation, called casually precedes, which is a more generalized relation than happens-before relation. This new relation does not sacrifice from the precision of happens-before relation, instead, it enables the detector to produce fewer false negatives. In [10], the authors present a sound predictive race detection technique based on a new foundation of maximal causal model incorporating the control flow information. There is also work on parallelizing data race detection [29] which shows that with 4 cores as the original application, they can speed up the median execution time by 4.4 for a happens-before detector and by 3.3 for a lockset race detector.

9 CONCLUSION

We presented a new segment-based hybrid data race detection algorithm with optimizations, which is a combination of happens-before and lockset algorithms. Data race detectors suffer from low performance and may produce many false alarms. We use the concept of segments as well as several other optimizations to improve its performance and reduce false alarms. We implemented our algorithms using a dynamic binary instrumentation platform. We compared our results with traditional lockset-based and happens-before based data race detection algorithm on several multithreaded applications and obtained favorable results. Our hybrid detector is 15 % faster than the happens-before detector and produces 50 % less potential data races than the lockset detector.

As a future work, the segment method can be applied to happens-before algorithm where it could improve the performance without sacrificing the precision. Although our dynamic binary instrumentation allows us to work with binaries of applications and does not require the source code, we will investigate other efficient instrumentation techniques, such as compiler based instrumentation, to improve the applicability of detectors in industrial settings.

Acknowledgment

This research was supported in part by Bogazici University Research Fund 13662.

REFERENCES

[1] Bienia, C.—Kumar, S.—Singh, J. P.—Li, K.: The PARSEC Benchmark Suite: Characterization and Architectural Implications. International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008, doi: 10.1145/1454115.1454128.

[2] Bond, M. D.—Coons, K. E.—McKinley, K. S.: PACER: Proportional Detection of Data Races. Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10), 2010, pp. 255–268, doi: 10.1145/1806596.1806626.

[3] Engler, D.—Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. ACM SIGOPS Operating Systems Review – SOSP '03, Vol. 37, 2003, No. 5, pp. 237–252, doi: 10.1145/945445.945468.

[4] Fidge, C.: Logical Time in Distributed Computing Systems. IEEE Computer, Vol. 24, 1991, No. 8, pp. 28–33, doi: 10.1109/2.84874.

[5] Flanagan, C.—Freund, N. S.: FastTrack: Efficient and Precise Dynamic Race Detection. Communications of the ACM, Vol. 53, 2010, No. 11, pp. 93–101, doi: 10.1145/1839676.1839699.

[6] Gilchrist, J.: Parallel Data Compression with bzip2. Proceedings of the International Conference on Parallel and Distributed Computing and Systems, 2004.

[7] Ha, O.-K.—Jun, Y.-K.: An Efficient Algorithm for On-the-Fly Data Race Detection Using an Epoch-Based Technique. Scientific Programming, 2015, Art. No. 205827, doi: 10.1155/2015/205827.

[8] Henzinger, A. T.—Jhala, R.—Majumdar, R.: Race Checking by Context Inference. ACM SIGPLAN Notices, Vol. 39, 2004, No. 6, pp. 1–13, doi: 10.1145/996841.996844.

[9] The Apache HTTP Server Project – 2.2.22, 2014. Accessed: May 2014.

[10] Huang, J.—O'Neil Meredith, P.—Rosu, G.: Maximal Sound Predictive Race Detection with Control Flow Abstraction. ACM SIGPLAN Notices, Vol. 49, 2014, No. 6, pp. 337–348.

[11] Jannesari, A.—Bao, K.—Pankratius, V.—Tichy, F. W.: Helgrind+: An Efficient Dynamic Race Detector. Proceedings of the IEEE International

Symposium on Parallel and Distributed Processing (IPDPS 2009), 2009, doi: 10.1109/IPDPS.2009.5160998.

[12] Kahlon, V.—Yang, Y.—Sankaranarayanan, S.—Gupta, A.: Fast and Accurate Static Data-Race Detection for Concurrent Programs. In: Damm, W., Hermanns, H. (Eds.): Computer Aided Verification (CAV '07). Lecture Notes in Computer Science, Vol. 4590, 2007, pp. 226–239.

[13] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM (CACM), Vol. 21, 1978, No. 7, pp. 558–565, doi: 10.1145/359545.359563.

[14] Luk, C.-K.—Cohn, R.—Muth, R.—Patil, H.—Klauser, A.—Lowney, G.—Wallace, S.—Reddi, V. J.—Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05), 2005, pp. 190–200, doi: 10.1145/1065010.1065034.

[15] Marino, D.—Musuvathi, M.—Narayanasamy, S.: LiteRace: Effective Sampling for Lightweight Data-Race Detection. Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09), 2009, pp. 134–143, doi: 10.1145/1542476.1542491.

[16] Matsumoto, M.—Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation (TOMACS), Vol. 8, 1998, No. 1, pp. 3–30, doi: 10.1145/272991.272995.

[17] Mattern, F.: Virtual Time and Global States of Distributed Systems. Proceedings of the Workshop on Distributed Algorithms (WDAG), 1989, pp. 120–131.

[18] Naik, M.—Aiken, A.—Whaley, J.: Effective Static Race Detection for Java. Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06), 2006, pp. 308–319, doi: 10.1145/1133981.1134018.

[19] Netzer, R. H. B.—Miller, B. P.: What Are Race Conditions? Some Issues and Formalizations. ACM Letters on Programming Languages and Systems (LOPLAS), Vol. 1, 1992, No. 1, pp. 74–88, doi: 10.1145/130616.130623.

[20] O'Callahan, R.—Choi, J.-D.: Hybrid Dynamic Data Race Detection. Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03), 2003, pp. 167–178, doi: 10.1145/781498.781528.

[21] Pozniansky, E.—Schuster, A.: Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03), 2003, pp. 179–190.

[22] Pratikakis, P.—Foster, S. J.—Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06), 2006, pp. 320–331, doi: 10.1145/1133981.1134019.

[23] POSIX Pthreads, IEEE Std 1003.1, 2013 Edition, `http://www.unix.org/version4/ieee_std.html`, 2014.

[24] Qadeer, S.—Wu, D.: KISS: Keep It Simple and Sequential. ACM SIGPLAN Notices – PLDI '04, Vol. 39, 2004, No. 6, pp. 14–24, doi: 10.1145/996841.996845.

[25] Savage, S.—Burrows, M.—Nelson, G.—Sobalvarro, P.—Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems, Vol. 15, 1997, No. 4, pp. 391–411, doi: 10.1145/265924.265927.

[26] Serebryany, K.—Iskhodzhanov, T.: ThreadSanitizer: Data Race Detection in Practice. Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09), 2009, pp. 62–71, doi: 10.1145/1791194.1791203.

[27] Smaragdakis, Y.—Evans, J.—Sadowski, C.—Yi, J.—Flanagan, C.: Sound Predictive Race Detection in Polynomial Time. Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12), 2012, pp. 387–400, doi: 10.1145/2103656.2103702.

[28] Voung, J. W.—Jhala, R.—Lerner, S.: RELAY: Static Race Detection on Millions of Lines of Code. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE '07), 2007, pp. 205–214, doi: 10.1145/1287624.1287654.

[29] Wester, B.—Devecsery, D.—Chen, P. M.—Flinn, J.—Narayanasamy, S.: Parallelizing Data Race Detection. Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13), 2013, pp. 27–38, doi: 10.1145/2451116.2451120.

[30] Xie, X.—Xue, J.: Acculock: Accurate and Efficient Detection of Data Races. Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11), 2011, pp. 201–212.

[31] Yu, Y.—Rodeheffer, T.—Chen, W.: Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05), 2005, pp. 221–234, doi: 10.1145/1095810.1095832.

[32] Zhai, K.—Xu, B.—Chan, W. K.—Tse, T. H.: CARISMA: A Context-Sensitive Approach to Race-Condition Sample-Instance Selection for Multithreaded Applications. Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), 2012, pp. 221–231, doi: 10.1145/2338965.2336780.

Alper Sen received his B.Sc. and M.Sc. degrees in electrical and electronics engineering from Middle East Technical University, Ankara, Turkey, in 1995 and 1997, respectively, and his Ph.D. degree in electrical and computer engineering from the University of Texas at Austin, Austin, in 2004. He was Technical Staff Member with Freescale Semiconductor, Austin, and an Adjunct Faculty Member with the University of Texas at Austin, until 2009. Currently he is Associate Professor with the Department of Computer Engineering, Bogazici University. His current research interests include verification of hardware and software systems, parallel programming, embedded systems, and system-level designs.

Onder Kalaci received his B.Sc. degree in computer engineering from Middle East Technical University, Ankara, Turkey, and his M.Sc. degree in computer engineering from the Bogazici University. He is currently Ph.D. student at the Bogazici University.