BlockRace: A Big Data Approach to Dynamic Block-based Data Race Detection for Multithreaded Programs †

Xiupei Mei
Department of Computer Science
City University of Hong Kong
Hong Kong
xpmei2-c@my.cityu.edu.hk

Zhengyuan Wei
Department of Computer Science
City University of Hong Kong
Hong Kong
zywei4-c@my.cityu.edu.hk

Hao Zhang
Department of Computer Science
City University of Hong Kong
Hong Kong
hzhang339-c@my.cityu.edu.hk

W.K. Chan ‡
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

ABSTRACT
The advent of multicore systems and distributed frameworks enables distributed strategies to address challenges in large-scale divisible problems by decomposing them into small ones, processing the corresponding sub-solutions and aggregating these sub-solutions into the final result. However, dynamic online detection of data races in execution traces of multithreaded programs is challenging to be parallelized due to their inherent historic event sensitivity and incremental inference of happens-before transitive closure. To examine the extent of such detection to be transformed into parallel versions running on Big data infrastructure, in this paper, we present BlockRace, a novel dynamic block-based data race detection technique, which precisely detects data races in such traces and checks pairs of events blocks in parallel using its novel strategy. We evaluate BlockRace on 18 programs, and the results show that BlockRace achieves 1.96x to 5.5x speedups compared to its sequential counterparts. To the best of our knowledge, BlockRace is the first technique to detect races in block pairs where these block pairs can be run in parallel on Big Data frameworks.

CCS CONCEPTS
• Software and it's Engineering → Software Testing and Debugging

KEYWORDS
data race detection, parallelization, concurrency bug, multithreaded programs, Big Data infrastructure

† This research is supported in part by the GRF of HKSAR Research Grants Council (project nos. 11214116 and 11200015), the HKSAR ITF (project no. ITS/378/18), the CityU MF_EXT (project no. 9678180), the CityU SRG (project nos. 7004882 and 7005216) and the CityU SGS Conference Grant.

‡ Corresponding author.

AST'20, October 2020, Seoul, South Korea

ACM Reference format:
Xiupei Mei, Zhengyuan Wei, Hao Zhang, and W.K. Chan. 2020. BlockRace: A Big Data Approach to Dynamic Block-based Data Race Detection for Multithreaded Programs. In *Proceedings of the 1st IEEE/ACM International Conference on Automation of Software Test (AST'20). ACM, New York, NY, USA, 10 pages.*

1 Introduction
With the advent of multicore processors, many programs are developed in a multithreaded fashion. However, due to the non-deterministic nature of concurrency, software developers are difficult to reason their code, resulting in leaving concurrency bugs in their programs [25]. At the same time, concurrency bugs are notoriously hard to test and debug. In the past decades, both the academia and industry have put efforts to find, locate, and repair them. Among all kinds of concurrency bugs, *data race* is one of the most common and fundamental kinds of concurrency bugs. A data race is said to occur when two threads in the same execution access the same memory location without synchronization protection and one of the accesses is a write [20].

To detect data races, many static and dynamic techniques have been proposed [10][14][20]. Static techniques do not scale well and incur heavy false alarms, and yet they have the potential to analyze the whole codebase. Dynamic techniques can precisely detect races (i.e., without reporting false positives), but usually incur high runtime overheads and their detection ability is limited to the program executions under analysis. In this paper, we limit our scope to dynamic precise race detection because we are interested in make such detection practical.

To the best of our knowledge, many dynamic online race detectors are designed based on the notion of vector clocks (VC) or its variants [16]. Their basic ideas are to capture the current timestamps of each thread on performing various kinds of operations and use the happens-before relations (see Section 2 for details), which is a partial order relation, over these operations to

maintain the latest timestamps of other threads visible to the thread in their corresponding VCs. In this way, the historic timestamps can be discarded to lower memory and runtime overheads incurred by these dynamic techniques.

Nonetheless, the above sequential update strategy along the event sequence captured in an execution trace, as what is used in DJIT+ or FastTrack [20], also limits their timestamp processing to the form of sequential algorithms. Thus, even though there could be many processors available through a distributed framework (e.g., a Big Data infrastructure in a data center), the current state-of-the-art dynamic data race detectors cannot easily harvest the computational power of the underlying infrastructure in analyzing a trace.

Another challenge faced of state-of-the-art dynamic online data race detectors is that certain VCs for each memory location in an execution trace should be kept in a detector's analysis state. However, as the number of threads increases or the memory consumption of the application is already close to the limit of the physical memory of a single machine (or node in a distributed system), analyzing races on all memory locations using these algorithms become increasingly slow. Existing strategies using the current data race detectors are to serialize the execution as a trace file, and analyze the trace file repetitively and sequentially, which incurs additional overheads to map between the analysis states of different such runs, albeit automatable.

A naïve strategy is to divide a trace file into multiple segments, and processes these segments in parallel. However, this strategy does not work because succeeding segments depend on the VC state of the preceding segments to start with. Parallelizing such a trace file for precise data race detection should firstly decouple the happens-before relations among such segments without losing the precision in detecting data races.

To the best of our knowledge, in this paper, we propose the *first work*, referred to as BlockRace, in detecting data races through concurrent set intersection operations on data records between blocks based on happens-before relations among them. We prove by theorems that BlockRace is sound and complete in precise data race detection. Moreover, BlockRace is novel in that its block partitioning algorithm converts VC operations commonly used in traditional VC-based data race detectors into operations of set intersection over pairs of memory access events, which enables possible use of Big Data framework to perform parallel operations over those pairs of memory access events in the same pair of datasets (block pair).

We have evaluated BlockRace on 18 programs and compared it to two well-known data race detectors (referred to as HB and HBEpoch, which are modelled after DJIT+ or FastTrack, respectively). Experimental results show that BlockRace can scale up with different numbers of threads. The multi-threaded version can achieve 1.96x to 5.5x speedups compared to its sequential counterpart, i.e. the single-threaded version. The experiment also shows that BlockRace is more efficient than HB and HBEpoch in terms of elapsed time on large benchmark programs.

The main contribution of this paper is threefold.

1. This paper presents BlockRace based on a customized notion of block, which innovatively transforms data race detection as vector-clock operations into set intersection operations over blocks, reducing runtime overheads and complexity of the detection.
2. We show the feasibility of BlockRace by implementing it as a tool, and the experimental results show that BlockRace incurs significantly lower slowdown overheads than its sequential counterpart while keeping the same level of precision as backed by theorems.
3. This work is the first work to show the feasibility and efficiency of selective race detection on an arbitrary subset of the threads in an execution, significantly extending the applicability of dynamic precise data race detections, which previously can only detect data races by analysing the operations of all but not some threads in the execution under analysis.

The rest of the paper is organized as follow. Section 2 introduces the preliminaries. Sections 3 and 4 present the motivation of our work and BlockRace, respectively. Section 5 reports the evaluation on BlockRace. The related work is reviewed in Section 6 followed by a conclusion in Section 7.

2 Preliminaries

In this section, we review the preliminaries of our model.

A multithreaded program consists of a set of threads H, a set of locks L, and a set of memory locations M.

In an execution of a multithreaded program, each *thread* $t \in H$ performs its events sequentially. These events include memory access events, i.e., read and write operations to a memory location $m \in M$, and synchronization events, which are lock acquisition and release operations on a lock $l \in L$. An execution *trace* is the sequence of such events.

If two memory access events occur at the same memory location m, and at least one of them is a write operation, the two events *conflict*.

There are three rules to quantify the ***happens-before relation*** (HBR $\rightarrow$): (1) In a trace T, if two events, a and b, are carried out by the same thread, and a appears before b in T, then $a \rightarrow b$. (2) If a is a lock release event, and b is a lock acquire event on the same lock in T, then $a \rightarrow b$. (3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two events a and b are concurrent (denoted as $a \parallel b$) in trace T if and only if they do not form any HBR, i.e., neither $a \rightarrow b$ nor $b \rightarrow a$.

A happens-before race (***HB Race***) is defined as a pair of concurrent conflicting memory accesses. The objective of precise HB race detection is to identify a subset of all HB races in a given trace without reporting false positives. Vector Clock (VC) can be

Algorithm 1: Basic VC-based HB Race Detection Algorithm

Input: The event trace of an execution
Output: The HB races in the given trace

```
1   onRelease(Thread t, Lock l)
2       C_l := C_t
3       C_t [t.tid] := C_t [t.tid] + 1
4
5   onAcquire(Thread t, Lock l)
6       C_t := C_t ⊔ C_l
7
8   onAccess(Thread t, Memory Location m)
9       fetch the last conflict access of m as C_m
10      if not C_m → C_t
11          report a HB-Race on m
12          C_m [t.tid] := C_t [t.tid]
```

used to track partial orders, and is widely used in precise data race detectors to track HBRs among all the events to detect HB races.

A VC is an array of thread clocks, each of which is an integer that is updated when conducting synchronization operations. A VC contains the synchronization information of the past events of each thread visible to the entity that the VC is associating to. For instance, let C_{t1} be the vector clock maintained for thread t_1 during the detection. The element $C_{t1}[t_2.tid]$ keeps the last thread clock of t_2 that has been observed by t_1 before the current moment of t_1.

Algorithm 1 depicts a baseline VC-based HB race detection algorithm. During the execution of this algorithm, the detector assigns a vector clock for each thread t and lock l, denoted as C_t and C_l respectively. When a thread t releases a lock l, it assigns the vector clock C_t to C_l, then increases its own thread clock in C_t (line 2- 3). When a thread t acquires a lock l, its vector clock C_t joins with C_l (line 2) for each i in C_t, $C_t[i] = max\ (C_t[i], C_l[i])$. This operation obtains the HB information in C_l which is left by the thread that just releases the lock l (line 6).

To detect data races, the algorithm assigns a vector clock to each memory location m, denoted as C_m, to record the last access information. Noted that here in this baseline version, for brevity, we do not distinguish read and write to sake for presentation clarity of BlockRace to be presented in Section 4.

When a thread t conducts a memory access operation to a memory location m, the algorithm fetches the vector clock of the last conflicting access of m as C_m (line 9). If C_m and C_t are not ordered by HBR, i.e., $C_m[i]$ is not always smaller than or equals to $C_t[i]$ for each i, the last conflict access event should not form any HB relation with the current access. Hence a HB race on m is reported (line 11). At the final step of Algorithm 1, the algorithm updates the current access information to C_m (line 12).

3 Motivation

To make VC-based approach to HB race detection applicable for industrial use, there are still significant challenges. A VC-based detector normally incurs high time and memory overheads. As it needs to keep the access information of each memory location for later reference, significant memory costs can be anticipated, especially in analyzing large program executions.

Parallelization is a de facto and promising approach to handle a kind of problem that can be divided into small ones to be processed concurrently. However, as depicted in Algorithm 1, typical algorithms for VC-based detection are highly sequential due to the needs to maintain HB relations, which makes its HB race detection non-trivial to parallelize and be run efficiently.

To overcome the above concurrency challenge, in this paper, we propose a novel alternative formulation of the problem of race detection. In essence, the whole data race analysis procedure is decomposed into a set of independent tasks without losing detection precision. We refer to our technique as **BlockRace**.

4 BlockRace: Our Block-Based Race Detector

In this section, we present BlockRace, our novel approach for HB-race detection. BlockRace consists of three phases. The first phase is to collect the happens-before relations among threads and record which memory locations are accessed by each thread after which synchronization events. The second phase is a partition phase that splits the collected block traces into a set of independent detection tasks. And the last phase is a block-level data race detection which can be parallelly conducted and run on Big Data infrastructures.

BlockRace is built on top of the following insights on data race detection: (1) As a postmortem analysis technique, the timestamps in VCs at each synchronization event position of each thread can be computed and kept offline, (2) the ordering of synchronization events among threads can be used to precisely characterize which segments (called blocks) of the two threads have been run concurrently, (3) the presence of the same memory location in a pair of such blocks that run concurrently indicates a data race, and (4) the checking on such presences of common memory locations between records (memory accesses) in a pair of such blocks can be performed as a set intersection operation between the corresponding two sets of records.

As such, these pairs of blocks can be run independent and concurrently to one another in checking the presence of data races in them, and the union of their results constitutes the set of data races detected on the trace produced in the first phase of BlockRace.

In the sequel, we introduce a format of trace logging, referred to as *Non-Sync Block Trace* (***Block Trace*** for short), which stores the execution events into blocks together with their inter-block dependency information. Then, we show how the entire race detection over the block traces could be partitioned, by our novel partitioning schemes, into a set of independent tasks, so that these tasks could be checked in a parallel manner, enabling Big Data processing on race detection. Finally, we present our novel block-based race detection algorithm and its theoretical guarantee.

4.1 Non-Sync Block Trace

The concept of *synchronization-free regions* (SFR) or *non-sync block* (NS-Block) has been proposed in prior works [11][26]. In this paper, we follow the definition of Non-Sync Block in [11] and utilize the trace-logging format for our block-based detection.

In [11], a *Non-Sync Block* (NS Block) is defined as a sequence of consecutive memory access events conducted by a thread between two synchronization events such that no other synchronization event exists between these two synchronization events. For brevity, we refer to a Non-Sync Block as a ***block***.

We use this notion to define concurrent block pairs as follows:

Two blocks α and β are said to form a HB relation (denoted as $\alpha \rightarrow \beta$), if there are two events $a \in \alpha$ and $b \in \beta$ such that $a \rightarrow b$. That is, if there is a HB relation between two events, one for each block, then these two blocks form a HB relation. Similarly, we define two blocks α and β as *concurrent* if neither $\alpha \rightarrow \beta$ nor $\beta \rightarrow \alpha$ is satisfied, denoted as $\alpha \parallel \beta$. We refer to the pair α and β as a ***concurrent block pair***.

We next prove that concurrent block pairs have the following three properties.

Property 1. For any two blocks α and β belong to two different threads, if $\alpha \rightarrow \beta$, then for any two events a and b from α and β, respectively, the relation $a \rightarrow b$ should hold.

Proof: Since we know $\alpha \rightarrow \beta$, by definition, there is a pair of events $x \in \alpha$ and $y \in \beta$ such that $x \rightarrow y$. The relation $x \rightarrow y$ must be achieved by the synchronization events x' and y' via $x \rightarrow x' \rightarrow y' \rightarrow y$ where $x \rightarrow x'$ and $y' \rightarrow y$ are maintained by the program order. Since by definition of NS Block, there is no synchronization operation in the same block, x' should appear after any event in α and y' should appear before any event in β. Therefore, for any two-event pair $a \in \alpha$ and $b \in \beta$, the relation $a \rightarrow b$ holds. □

Property 2. For any two concurrent blocks α and β, any event in one block is **concurrent** with all the events in another block.

Proof: By the definition of concurrent block pair, if $\alpha \parallel \beta$, then neither $\alpha \rightarrow \beta$ nor $\beta \rightarrow \alpha$ is satisfied, i.e., for any event $a \in \alpha$ and $b \in \beta$, neither $a \rightarrow b$ nor $b \rightarrow a$ is satisfied. Hence, a is concurrent with b. □

Property 3. For any two blocks α and β, if there exist events $a \in \alpha$ and $b \in \beta$ such that $a \parallel b$, then $\alpha \parallel \beta$ is also satisfied.

Proof: If α is not concurrent with β, there must be a HB-relation between them. With the loss of generality, suppose that $\alpha \rightarrow \beta$ holds. Per the definition of HB relation between blocks, there are two events $x \in \alpha$ and $y \in \beta$ such that $x \rightarrow y$. According to Property 1, $a \rightarrow b$ should hold, which contradicts to $a \parallel b$. Therefore, we should have $\alpha \parallel \beta$. □

An *NS-block trace* records a sequence of blocks conducted by a thread. A ***multithreaded program execution*** is a set of block traces together with the HB relations between the blocks involved in these traces.

Algorithm 2: Tracking Non-Sync Block Traces

Input: The event stream from an execution
Output: The block traces of all threads

```
1  Initialize rMap and wMap to empty for each Thread t
2
3  onRead(Thread t, Memory Location m)
4     Fetch current event as e
5     t.rMap[m] := t.rMap[m] ∪ {e}
6
7  onWrite(Thread t, Memory Location m)
8     Fetch current event as e
9     t.wMap[m] := t.wMap[m] ∪ {e}
10
11 onAcquire(Thread t, Lock l)
12    persistBlock(t)
13    C_t := C_t ⊔ C_l
14    C_t[t.tid] := C_t[t.tid] + 1
15
16 onRelease(Thread t, Lock l)
17    persistBlock(t)
18    C_l := C_t
19    C_t[t.tid] := C_t[t.tid] + 1
20
21 onFork(Thread t, Thread u)
22    persistBlock(t)
23    C_u := C_t
24    C_t[t.tid] := C_t[t.tid] + 1
25
26 onJoin(Thread t, thread u)
27    persistBlock(t)
28    persistBlock(u)
29    C_t := C_t ⊔ C_u
30    C_t[t.tid] := C_t[t.tid] + 1
31
32 persistBlock(Thread ti)
33    create a new NS Block α
34    α.rMap := t.rMap
35    α.wMap := t.wMap
36    α.lastPred := C_t
37    append α to the t's block trace
38    reset t.rMap and t.wMap to empty
```

Algorithm 2 shows BlockRace's tracking algorithm over a set of block traces of a multithreaded program execution. Note that the tracking procedure uses different functions to track each event based on the event type.

For any thread t, the algorithm maintains two maps. Each map uses the accessed memory location as the key and the set of events as the values so that no duplicated access events in a block is kept.

Two maps, denoted as *t.wMap* and *t.rMap,* are used to temporarily store the read and write events within a block (lines 3–9). On handling events with synchronization operations (e.g., Acquire, Release, Fork and Join), the algorithm creates a new block

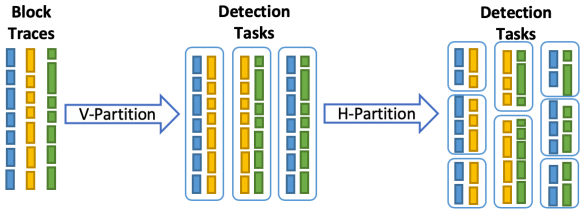

Figure 1. Overview of the partition schemes.

to keep the maintained events (lines 12, 17, 22, 27, 28). By doing so, it ensures that no synchronization events exist in between any two events in a block, which produces a NS block.

On a thread t producing a new block α, the current vector clock of t is also kept in α, (denoted as $\alpha.lastPred$, line37) to facilitate the determination of HB relations among blocks in a later stage of BlockRace.

More specifically, suppose that blocks α and β belong to thread i and j, respectively. The entry $\alpha.lastPred[j]$ keeps the last clock of thread j happening before α, and $\beta.lastPred[j]$ tracks the thread clock of β. If $\alpha.lastPred[j] > \beta.lastPred[j]$, then $\beta \rightarrow \alpha$ holds. Otherwise, β does not happen before α, i.e., $\beta \nrightarrow \alpha$.

The algorithm uses VCs to keep track of HB relations between blocks. The VCs of threads and locks are updated when synchronization events are handled (lines 13–14, 18–19, 23–24, 29–30). This procedure is similar with the baseline VC-based race detection algorithm (Algorithm 1), but is much more lightweight, as Algorithm 2 needs not to maintain VCs of any memory locations, which will be presented in the next sub-section.

4.2 Partition Schemes in BlockRace

To examine the entire multithreaded program execution, the block traces of all the threads need to be pairwise checked. In this subsection, we present how BlockRace covert the whole data race detection process into finer granularity through its novel vertical and horizontal partitioning schemes, summarized as V-Partition and H-Partition.

Figure 1 shows an overview of these two partition schemes that convert the entire detection work of a multithreaded program execution into a set of independent detection tasks (depicted by rounded rectangles in Figure 1). Each detection task involves a pair of block traces from two threads. V-Partition divides and pairs block traces of all the threads into a set of tasks of which each is responsible for examining a pair of threads. It is named V-partition as it works like slicing the trace for analysis vertically along the selected threads of control in the program execution. Moreover, based on V-Partition, each divided detection task on a thread pair can further be partitioned horizontally by H-partition, which splits each task into even smaller ones.

V-Partition: For a detection task on two threads, the involved computation only depends on their block traces and the HB relations between them, of which both information has already been profiled and stored (e.g., as datasets in the file system). All

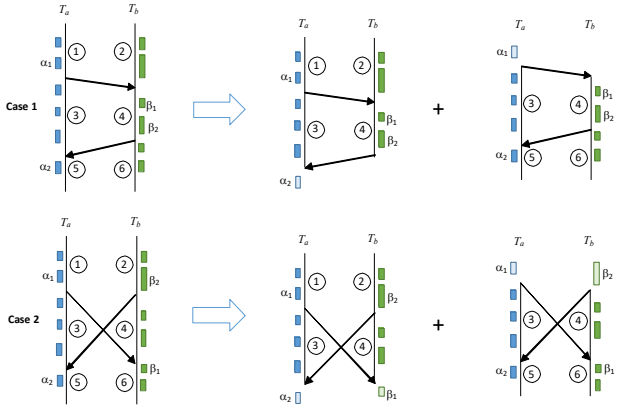

Figure 2. Two cases on how H-Partition split a V-Partition

detection tasks are independent to one another, which allows them to be conducted in an arbitrary order, sequentially or in parallel. Besides parallelization, V-Partition also brings in an interesting feature: Under particular scenarios, developers might only have interests over the HB races between a few specific threads instead of the whole executions. V-Partition can accommodate such customized examination.

H-Partition: For a detection task involving two threads, the task is further spitted via H-Partition into a set of smaller tasks for parallel processing, which is presented below.

First, we define the notion of ***round relation*** as follows. Given two thread traces T_a and T_b, *round relations* is a pair of HB relations $\alpha_1 \rightarrow \beta_1$ and $\beta_2 \rightarrow \alpha_2$ such that α_1 and α_2 are in T_a and β_1 and β_2 are in T_b, i.e., the two HB relations are not align in the same direction going from one thread to the other thread. If neither $\alpha_x \rightarrow \beta_x$ nor $\beta_x \rightarrow \alpha_x$ exists for all events (say α_x) in between α_1 and α_2 in T_a and for all events (say β_x) in between β_1 and β_2 in T_b, then we say that $\alpha_1 \rightarrow \beta_1$ and $\beta_2 \rightarrow \alpha_2$ are the pair of ***most adjacent round relations***.

Our insight on real-world program executions is that the condition required by *round relation* is fairly easy to be satisfied. Intuitively, H-partition divides a pair of thread trace into partitions such that each partition contains a pair of most adjacent round relations in the original program execution. Figure 2 depicts the scenarios in H-Partition.

In Figure 2, the two relations (i.e. $\alpha_1 \rightarrow \beta_1$ and $\beta_2 \rightarrow \alpha_2$) split the trace pair into six regions (indicated by ① – ⑥ in the figure). Suppose that $\alpha_1 \rightarrow \alpha_2$ in T_a. There are two cases to consider:

In Case 1, relation $\beta_1 \rightarrow \beta_2$ exists in T_b. The blocks in region ① form HB relations with the blocks in regions ④ and ⑥, and blocks in regions ② and ④ form HB relations with blocks in region ⑤. In Case 2, relation $\beta_2 \rightarrow \beta_1$ exists in T_b. Blocks in region ① form HB relations with blocks in region ⑥, and blocks in region ② form HB relations with blocks in region ⑤.

In both cases, blocks in regions ① and ② form HB relations with blocks in regions ⑤ and ⑥. Also, there does not exist any HB race in between these blocks. So, H-Partition can precisely

Algorithm 3: Race detection between two block traces

Input: Two block traces T_i, T_j from a detection task.
Output: All HB races between T_i and T_j.

```
1   detect(BlockTrace Ti, BlockTrace Tj)
2     Initialize queue Qi, Qj to empty
3
4     while Ti or Tj has block to process:
5       while Ti has block to process:
6         α := Ti.nextBlock
7         if Qj.tail ↛ α
8           deque Qj until Qj.head ↛ α
9           checkBlocks(α, β) for each block β in Qj
10          Qi.enqueue(α)
11        else
12          break
13
14      // Similar procedure in lines 5 – 12
15      while Tj has block to process:
16        α := Tj.nextBlock
17        if Qi.tail ↛ α
18          deque Qi until Qi.head ↛ α
19          checkBlocks(α, β) for each block β in Qi
20          Qj.enqueue(α)
21        else
22          break
```

Algorithm 4: Checking conflict events in two NS blocks

Input: Two concurrent NS blocks α and β.
Output: All HB races consist by events from α and β.

```
1   checkBlocks(Block α, Block β)
2     if α.wMap.keys ∩ β.wMap.keys is not empty
3       report W-W race warning(s)
4     if α.rMap.keys ∩ β.wMap.keys is not empty
5       report R-W race warning(s)
6     if α.wMap.keys ∩ β.rMap.keys is not empty
7       report W-R race warning(s)
```

partition the V-Partition detection task into two sub-tasks, i.e., the first sub-task stops after checking regions ③ and ④, and the second sub-task starts from checking regions ③ and ④. The two sub-tasks are independent to one another, and thus, can be processed in parallel.

4.3 HB Race Detection on NS Block Traces

This section presents BlockRace's race detection algorithm, which needs not to maintain any meta-information of memory locations.

At the heart of the second phase of BlockRace is to determine whether two blocks are concurrent which is computed via Algorithm 3. In Section 4.4, we present the theoretical guarantee that our algorithms can detect all HB races in a sound and complete manner.

Algorithm 3 maintains two queues for block traces T_i and T_j, denoted as Q_i and Q_j. Each queue maintains a sequence of consecutive blocks that are concurrent with the ones in another queue. The algorithm iterates blocks in each trace and stops once all the blocks are visited (line 4). When processing each block, say block α in one of the traces T_i, the algorithm first examines whether the last block in Q_j forms a HB relation from it to α (line 7). If this is the case, it dequeues the blocks in Q_j until $Q_j.head$ does not form any HB relation form it to α (line 8). As a result, all the blocks in Q_j are determined as concurrent with α and will be further examined for potential races via the *checkBlock*() function which invokes Algorithm 4 (line 9), and then the block α is added to Q_i for later checking from blocks in T_j (line 10). The checking logic for blocks in T_j (lines 15–22) is symmetrical to the part on processing blocks in T_i above.

Given two concurrent blocks α and β, any memory access conflict within α and β is a HB race. Algorithm 4 presents a novel approach to detect such races — It performs operation of set intersections instead of pairwise update of VCs to maintain HB relations when detecting data races. As mentioned in sub-section 4.1, in each block, its *rMap* and *wMap* are two maps storing the read and write events and using the memory location as the key. Hence, in Algorithm 4, once the intersection of the keys from two maps is not empty and at least one map is *wMap* (lines 2, 4, 6), there exists conflict events in these two maps. Then according to the intersection result, the corresponding access events can be fetched from the two maps and each pair of them is reported as a HB race.

4.4 Soundness and Completeness of Algorithms 3 and 4

In this section, we present the correctness of BlockRace.

Lemma 1. If $\alpha \parallel \beta$, either of them will not be dequeued until the other is loaded in queue by Algorithm 3.

Proof: Assume β is just loaded, if α has already been dequeued from Q_1, then there exists a block γ from T_2 such that $\alpha \rightarrow \gamma$ and $\gamma \rightarrow \beta$, hence $\alpha \rightarrow \beta$, which contradict the condition $\alpha \parallel \beta$.

Theorem 1. Given two block traces T_1, T_2, for any block $\alpha \in T_1$ and $\beta \in T_2$, the pair $<\alpha, \beta>$ will be checked for conflict events in Algorithm 3 if and only if $\alpha \parallel \beta$.

Proof: *Necessity*. Assume that $\alpha \parallel \beta$, and α is just loaded from T_1 (line 5). According to Lemma 1, block β could not have been dequeued from Q_2. Now there are two possible scenarios: Case 1: If β is already in Q_2, then α and β will be checked in line 9. Case 2: If β is not in Q_2, then α will be added to Q_1 in line 10. α will not be removed before β is loaded (Lemma 1). So, when β is loaded in later loop (in the while body under line 13), it will be checked with α. In both cases, α and β are checked in Algorithm 3.

Sufficiency. If $<\alpha, \beta>$ is checked during Algorithm 3, say it is checked in line 9, then a) β is in Q_2; and b) β is not dequeued in line 9. For a), it indicates that β is added to the queue in theprevious loop, and $\beta.lastPred \rightarrow Q_j.tail$. As α has not been loaded at that moment, $Q_j.tail \rightarrow \alpha$ satisfied, $\alpha \nrightarrow \beta$ holds. Otherwise, it will contradict with the definition of the last predecessor block of β. For b), it indicates $\beta \nrightarrow \alpha$ holds.

Table 1. Statistic information of benchmarks

Category	Program	Loc	#Thread	#Locks	#Var	#Events	#Blocks
Small	array	22	3	2	30	47	6
	critical	30	4	0	30	55	6
	account	37	4	3	41	130	10
	airlinetickets	35	7	0	44	148	12
	pingpong	36	6	0	51	153	20
	mergesort	52	5	4	317	675	19
	boundedbuffer	99	4	5	64	1400	105
	bubblesort	70	26	2	167	6856	194
	bufwriter	49	6	1	69	9796	954
	raytracer	204	3	8	3854	20065	64
	ftpserver	1455	11	160	5068	46152	5146
	moldyn	422	3	2	1172	181044	35
Medium	derby	6509	4	936	159423	1213530	31538
	jigsaw	2646	13	277	103582	3063863	5838
	montecarlo	337	3	3	876093	7283306	36
Large	eclipse	14449	15	8456	10092329	85580931	937538
	xalan	4261	6	2491	4399314	122039821	1043147
	lusearch	2042	7	123	4778326	216802287	206576

Therefore, while neither $\alpha \rightarrow \beta$ nor $\alpha \rightarrow \beta$ is satisfied, we conclude $\alpha \,||\, \beta$.□

Theorem 2. Algorithms 3 and 4 are sound and complete on detecting HB-races in trace *T*.

Proof: *Soundness*: *For all reported warnings, they are HB-races.*

For each reported warning pair <*a, b*> from Algorithm 3, i.e., there are two NS Blocks α and β from two threads such that: 1) $a \in \alpha, b \in \beta$ and 2) $\alpha \,||\, \beta$ (by Theorem 1).

By Property 2, *a* and *b* are concurrent. As *a* and *b* are also conflicting, the reported pair <*a, b*> is a HB-race in *T*.

Completeness: *All HB-races will be detected.*

For a HB-race pair <*a, b*> in trace *T*, since *a* is concurrent with *b*, and there must be two NS Blocks α and β such that $a \in \alpha$ and $b \in \beta$. By Property 3, α and β are also concurrent. By Theorem 1, α and β will be checked and reported during the detection task of the two involved threads. □

5 Experiment

5.1 Implementation

To evaluate BlockRace, we implemented it in Java, and compared it against the standard HB and *FastTrack* (referred to as HBEpoch), which are partial-order detectors based on vector clocks and epochs, respectively [20]. The implementations of the two detectors were obtained from Rapid [1], an open source framework implemented in Java. We implemented BlockRace with the fixed thread pool executor in Java. In this implementation, the whole detection task is partitioned and queued first, and each active worker thread executes a task at a time and keeps getting a new task from the queue until the queue is empty. All the worker threads run in parallel on our multi-core platform.

The thread pool size was configured by the parameter *w*. When *w* was set to one, all the detection tasks were conducted with one worker thread, i.e., the detector worked in the sequential mode. To store the block traces into trace files, we used the Kryo library to serialize block objects [4].

Note that, to avoid confounding effects brought by different languages and frameworks, we implement BlockRace as a Java program without employing Big data frameworks. However, as each detection task is independent from others, it enables BlockRace to be easily implemented with parallel programming paradigms (e.g. MapReduce), and executed on Big data infrastructures. More specifically, a typical MapReduce program contains two kinds of tasks, the map task and the reduce task [2]. The input dataset of the program is split into independent chunks which are parallelly processed by map tasks. And the outputs are then accumulated by the reduce task. For BlockRace detection, after H-Partition and V-Partition, the whole detection work is split into a set of independent tasks. Each map task can accept one of them and produce the corresponding detection results, and these results can be gathered by the reducer as the final report.

5.2 Benchmarks

Our benchmarks were selected to be those that were widely used in recent works on data race detection [22]. In particular, they were concurrent Java programs taken from standard benchmark suites: 1) the IBM Contest benchmark suite, 2) the Java Grande Forum

Table 2. Summary of Results of HB, HBEpoch and BlockRace

	HB		HBEpoch		BlockRace		
Program	**Time**	**#Races**	**Time**	**#Races**	**Time**	**#Races**	**#Races***
array	00:00.3	0	00:00.3	0	00:00.4	0	0
critical	00:00.3	3	00:00.3	2	00:00.4	8	6
account	00:00.3	4	00:00.3	2	00:00.4	4	5
airlinetickets	00:00.3	5	00:00.3	2	00:00.7	69	5
pingpong	00:00.5	5	00:00.4	4	00:00.9	20	5
mergesort	00:00.4	2	00:00.3	2	00:00.6	7	2
boundedbuffer	00:00.4	7	00:00.4	7	00:00.6	18	9
bubblesort	00:00.9	5	00:00.8	3	00:01.5	266	5
bufwriter	00:00.9	4	00:00.6	4	00:01.2	5	4
raytracer	00:01.0	4	00:01.0	4	00:00.6	3	5
ftpserver	00:01.5	30	00:01.4	30	00:01.9	47	48
moldyn	00:01.5	30	00:01.3	27	00:00.6	44	30
derby	00:05.8	20	00:05.3	16	00:02.6	27	29
jigsaw	00:16.6	9	00:11.3	7	00:08.4	13	13
montecarlo	00:19.0	5	00:16.5	5	00:05.0	13	20
eclipse	06:30.0	42	04:34.9	38	05:51.1	58	68
xalan	06:05.6	11	04:42.9	9	03:22.0	91	18
lusearch	09:47.0	35	06:51.4	28	01:55.4	141	59

benchmark suite, 3) some real-world software – *Apache FTPServer*, *W3C Jigsaw* web server and *Apache Derby*, and 4) the *DaCaPo* benchmark suite (version 9.12-MR1).

All the experiments were conducted on a machine equipped with 96 CPU cores and 1.5TB main memory. The operation system is 64-bit Ubuntu 16.04.5 LTS and the OpenJDK JVM with version 1.8.0_191.

The tool Rapid accepted trace files generated from the RV-Predict logging solution, while our tool BlockRace accepted the block traces as the input. We used RV-Predict to profile the trace files for HB and HBEpoch. and generated our block-based format trace files from the same traces. Hence all the detection algorithms were applied on trace files from the same program execution and their detection results were comparable.

Table 1 shows the information of the 18 benchmarks. The second and third columns are the name of each program and lines of codes touched by the execution. And the fourth to the last columns show the statistic information of the recorded execution traces.

According to the number of the profiled events in a trace (trace size), we divide these programs into three categories: small, medium, and large.

The small set contains 12 programs of which the trace sizes are less than one million). Three programs, *derby*, *jigsaw* and *montecarlo*, belong to the medium category where trace sizes range from one million to ten million. And the three benchmark programs that have more than ten million events fall into the large category.

5.3 Data Analysis

Our evaluation consists of two parts. First, we assess the performance and accuracy of our block-based detection approach. In this part, the concurrent parameter w was set to 1 for fair comparison (sequential mode). Second, we conducted experiments to evaluate the parallelization scalability of our approach. In the experiment, we recorded and compared the performance of our method with different values of w.

5.3.1 Performance and Accuracy

We compared the performance of BlockRace to HB and HBEpoch (with $w = 1$).

Table 2 summarizes the runtime and detection results of each detector. Columns 2, 4 and 6 are the time spent by each detector. For each benchmark program, we ran each detector 10 times and used the mean time spent as its final result. From the table, we can see that for smaller workloads, all three detectors completed their analyses quickly (within 2 seconds). But for the six workloads in the medium and large categories, BlockRace ran 1.1x to 5.1x faster than HB in all these six programs and 1.3x to 3.6x faster than HBEpoch in five out of six programs.

Columns 3, 5 and 7 summarize the reported race warnings of each detector. As shown, there is high variance in detection results for some programs such as *airlinetickets*, *bubblesort*, *xalan* and *lusearch*. To understand the variance, we studied the detailed implementation of Rapid. The variance was found to be caused mainly by the differences in between the race warning definitions of the implementations. More specifically, for a HB race pair $\langle e_1, e_2 \rangle$ and e_1 occurs early than e_2 in the trace, during the processing of

Table 3. Time spent and memory usage of BlockRace on analyzing programs with different numbers of thread

Program	k	1	2	4	8	16
lusearch	Time	01:55.4	01:10.5	00:51.1	00:42.5	00:43.1
	Mem (MB)	16893	16869	20968	23777	26598
xalan	Time	03:22.0	01:45.4	01:08.5	00:49.6	00:36.8
	Mem (MB)	27039	35010	24486	26915	31819
eclipse	Time	05:51.1	03:56.7	02:59.4	03:30.2	03:36.0
	Mem (MB)	26887	48031	66803	69606	87334

event e_2, HB and HBEpoch detectors find e_2 involved in a race and report e_2's line number in the program source code as a race warning. This design does not report those data races where e_2 appears in multiple race pairs. In contrast, BlockRace reports the thread id and line number of both events in a race pair, therefore reporting more race warnings than HB and HBEpoch.

To further examine the differences, we converted the warnings of block-based detector into the same format being used in Rapid. Results are shown in Column 8. A closer examination reveals that all the race warning events reported by HB and HBEpoch were found by BlockRace.

Another cause of the varied results is that VC-based detectors cannot deal with successive access events in one thread. For instance, assume that there are two adjacent write memory events, e_1 and e_2, both in one thread, as well as a third event e_3, which is a concurrent write event in another thread. All the three events access to the same memory location, so $\langle e_1, e_3\rangle$ and $\langle e_2, e_3\rangle$ are two race pairs. Now assume that e_1 and e_2 occur early than e_3, VC-based detectors will record e_2 only for later reference, missing the pair $<e_1, e_3>$ in their results. However, BlockRace does not miss such races, as both e_1 and e_2 are maintained in the same block.

We recorded the maximum resident set size of each execution using the GNU time command with the *-verbose* option enabled. The overall results show that BlockRace incurs smaller memory overheads than the other two detectors. Owing to page limit, for brevity, we leave the report of the memory overhead in our future work.

5.3.2 Scalability

The largest three workloads (*eclipse*, *xalan* and *lusearch*) were utilized to assess the parallelization ability of BlockRace. We executed BlockRace with different numbers of parallel workers (w = 1, 2, 4, 8, 16), so that the detection tasks were completed by different numbers of concurrent workers. Theoretically, as the whole detection tasks are same, the detection will be accelerated with multiple workers. Table 3 summaries the results.

Though we did not present the detection result in Table 3 due to the page limit, we did find the same number of races being detected in the same programs across different numbers of threads used. The finding is in accordance to the design of BlockRace that the detection tasks remained independent to each other.

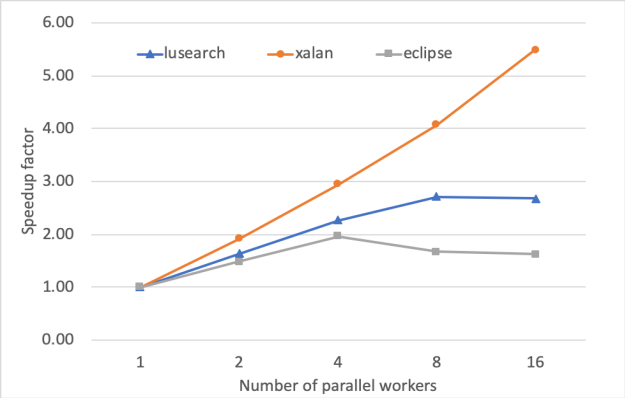

Figure 3. Speedup factors achieved by assigning different numbers of parallel workers

The plot in Figure 3 shows the performance speedup of BlockRace with different numbers of parallel workers. On the three programs, BlockRace achieved noticeable speedup when being conducted with more parallel workers. On *lusearch*, BlockRace reached its peak performance when w = 8, achieving 2.7x speedup. It achieved 5.5x speedup with w = 16 on *xalan* and 1.96x speedup when w = 4 on *eclipse*.

From our data analysis, we further formulated a few reasons that impeded the parallelization scalability of BlockRace in the experiment. The most important reason lies in the I/O process. The I/O performance became worse when multiple tasks tried to access the same trace file concurrently from the disk, which in turn affected the detection efficiency. However, in distributed environment such as Hadoop, the trace files split into blocks and replicated across multiple machines. When detection tasks are executed using frameworks such as Spark or Hadoop, tasks can be parallelly executed on multiple machines and access replicas of the same trace file without interfering each other, which will likely mitigate the limitation brought by I/O. Another major factor affecting the results is the caching optimization strategy applied by the operating systems. We found that when a trace file was loaded multiple times within a short period, the loading procedure for late accesses were faster than the first one. Therefore, with our approach, the overall performance could vary with randomly assigned execution order of the detection tasks.

6 Related Work

Concurrent programming is difficult and error-prone. Concurrency bugs are widely existed in applications [25]. Many detection techniques to detect data races [20][21][22][27], deadlocks [13], and atomicity violations [6][12] have been developed in the past few decades. To detect data races, both static and dynamic techniques are proposed. Static methods [14] could analyze the whole program but usually report false alarms. Dynamic techniques utilize information of happens-before relations [10][20], locking discipline [27] or combination of the two [16] for race detection.

However, the accuracy of data race detection is often achieved in exchange of inefficiency and high runtime overheads [15]. Prior

research has attempted to balance the tradeoff between execution performance and detection accuracy. For example, to avoid high overhead, sampling strategies were proposed and applied to monitor a set of memory access [11][17]. There are also techniques using special hardware or operating system support to lower the runtime overheads [18][19].

With the prevalence of multi-core systems, works on parallelizing the detection analysis tasks to better utilize the hardware resource and to improve the performance of race detectors were proposed [7][8][10]. Our present work is along this line of research.

Wester et.al [8] presented a strategy in which each thread execution is defined as a series of timeframes such that in the same timeframe, one core is designated to execute all code blocks of all threads. The strategy speeds up the detection and scales well with multiple cores. However, it relies on a new multithreading paradigm called *uniparallelism* [7], which is different from the task parallel paradigm supported by typical thread libraries. Also, their detector requires heavy modifications on OS and shared libraries and needs customization of the detection algorithms.

ParallelFT is proposed to increase the parallel execution of the FastTrack detector on multicore architectures [10]. The approach relies on the independency of the race analysis on each memory location, and partitions the access events by their target memory locations and distributes the analysis workload to a set of independent detection threads.

Sakurai et.al [9] found that evenly divided memory space tends to skew the race analysis workloads to a few detection threads. They extended *ParallelFT* by proposing *POI* which provides load-balancing among the task distribution.

However, a major limitation of *ParallelFT* is its assumption on the analysis independency on each memory location: The independency does not always hold. For more recent order-based race detection algorithms such as CP, WCP, M2 [21][22][23], the relation of events could be established by multiple variables. which makes it impossible to applying *ParallelFT*'s partition approach on these algorithms. As a comparison, our approach could be extended to handle these algorithms as our trace generation is orthogonal to the order maintaining mechanism used in these algorithms.

7 Conclusions and Future Work

In this paper, we have presented BlockRace, a novel block-box parallelizable detection to dynamically detect data races from multithreaded programs. It involves a novel scheme (including V-Partition and H-Partition) to transform an execution trace into a set of independent pairs of blocks of thread operations, where all these pairs can be subject to race detection in parallel using set intersection operations. The partition scheme is suitable for Big Data processing. We have conducted an evaluation on BlockRace on 18 subjects and compared it to two precise sequential detectors and on different numbers of parallel workers to examine its parallelization capability. The results have shown that BlockRace is precise and efficient, and scales well. There are limitations in BlockRace. One major limitation is that the number of detection tasks could be high, i.e. polynomial to the number of threads in the trace under analysis. Hence, each trace file would be loaded into the memory multiple times by various tasks. We plan to address it in our future work.

REFERENCES

[1] Rapid, https://github.com/umangm/rapid
[2] Apache Hadoop, https://hadoop.apache.org/
[3] Apache Spark, https://spark.apache.org/
[4] Kryo, https://github.com/EsotericSoftware/kryo
[5] RVPredict, https://runtimeverification.com/predict/
[6] F. Sorrentino, A. Farzan, and P. Madhusudan, "PENELOPE: Weaving Threads to Expose Atomicity Violations," in Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 2010, pp. 37–46.
[7] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing sequential logging and replay. In Proc. 2011 International Conference on Architectural Support for Programming Languages and Operating Systems, pages 15–26.
[8] B. Wester, D. Devecsery, P. M. Chen, J. Flinn, and S. Narayanasamy, "Parallelizing Data Race Detection," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 2013, pp. 27–38.
[9] Y. Sakurai, Y. Arahori, and K. Gondow, "POI: Skew-Aware Parallel Race Detection," in Proceedings of IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2018, pp. 215–224.
[10] Y. W. Song and Y.-H. Lee, "A Parallel FastTrack Data Race Detector on Multi-core Systems," in Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 387–396.
[11] Y. Cai, J. Zhang, L. Cao, and J. Liu, "A deployable sampling strategy for data race detection," in In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 810–821.
[12] S. Wu, C. Yang, and W. K. Chan, "ASR: Abstraction Subspace Reduction for Exposing Atomicity Violation Bugs in Multithreaded Programs," in 2015 IEEE International Conference on Software Quality, Reliability and Security, 2015, pp. 272–281.
[13] Y. Cai and W. K. Chan, "Magiclock: Scalable Detection of Potential Deadlocks in Large-Scale Multithreaded Programs," IEEE Transactions on Software Engineering, vol. 40, no. 3, pp. 266–281, Mar. 2014.
[14] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static Race Detection on Millions of Lines of Code," in Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, New York, NY, USA, 2007, pp. 205–214.
[15] B. P. Wood, M. Cao, M. D. Bond, and D. Grossman, "Instrumentation Bias for Dynamic Data Race Detection," Proc. ACM Program. Lang., vol. 1, no. OOPSLA, pp. 69:1–69:31, Oct. 2017.
[16] J. Yang, C. Yang, and W. K. Chan, "HistLock: Efficient and Sound Hybrid Detection of Hidden Predictive Data Races with Functional Contexts," in 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2016, pp. 13–24.
[17] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-Race Detection," In ACM Sigplan notices (Vol. 44, No. 6). ACM. pp. 134-143.
[18] E. Vlachos et al., "ParaLog: Enabling and Accelerating Online Parallel Monitoring of Multithreaded Applications," in Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 2010, pp. 271–284.
[19] T. Zhang, C. Jung, and D. Lee, "ProRace: Practical Data Race Detection for Production Use," in ACM SIGOPS Operating Systems Review 51, no. 2, 2017, pp. 149-162.
[20] C. Flanagan and S. N. Freund, FastTrack: efficient and precise dynamic race detection. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09). ACM, New York, NY, USA, 121-133.
[21] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound Predictive Race Detection in Polynomial Time," in Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, 2012, pp. 387–400.

[22] D. Kini, U. Mathur, and M. Viswanathan, "Dynamic Race Prediction in Linear Time," in ACM SIGPLAN Notices, vol. 52, no. 6, Jun. 2017, pp. 157–170.

[23] A. Pavlogiannis, "Fast, Sound and Effectively Complete Dynamic Race Detection," arXiv:1901.08857 [cs], Jan. 2019.

[24] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants," IEEE Micro, vol. 27, no. 1, pp. 26–35, Jan. 2007.

[25] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics," in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 2008, pp. 329–339.

[26] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia, "Valor: efficient, software-only region conflict exceptions," in ACM SIGPLAN Notices, vol. 50, no. 10, 2015, pp. 241–259.

[27] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. "Eraser: A dynamic data race detector for multi-threaded programs. " In ACM Transactions on Computer Systems (TOCS) 15, no. 4, 1997, pp. 391–411.