Dynamic Race Detection with O(1) Samples

MOSAAD AL THOKAIR, University of Illinois at Urbana-Champaign, USA

MINJIAN ZHANG, University of Illinois at Urbana-Champaign, USA

UMANG MATHUR, National University of Singapore, Singapore

MAHESH VISWANATHAN, University of Illinois at Urbana-Champaign, USA

Happens before-based dynamic analysis is the go-to technique for detecting data races in large scale software projects due to the absence of false positive reports. However, such analyses are expensive since they employ expensive vector clock updates at each event, rendering them usable only for in-house testing. In this paper, we present a sampling-based, randomized race detector that processes only *constantly many* events of the input trace even in the worst case. This is the first *sub-linear* time (i.e., running in $o(n)$ time where n is the length of the trace) dynamic race detection algorithm; previous sampling based approaches like Pacer run in linear time (i.e., $O(n)$). Our algorithm is a property tester for HB-race detection — it is sound in that it never reports any false positive, and on traces that are far, with respect to hamming distance, from any race-free trace, the algorithm detects an HB-race with high probability. Our experimental evaluation of the algorithm and its comparison with state-of-the-art deterministic and sampling based race detectors shows that the algorithm does indeed have significantly low running time, and detects races quite often.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Concurrency - Shared memory, Dynamic program analysis, Property testing, Happens-before race detection

ACM Reference Format:

Mosaad Al Thokair, Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. 2023. Dynamic Race Detection with O(1) Samples. *Proc. ACM Program. Lang.* 7, POPL, Article 45 (January 2023), 30 pages. https://doi.org/10.1145/3571238

1 INTRODUCTION

A concurrent program is said to have a data race (or simply a race) if it has an execution in which a pair of threads access a shared memory location consecutively and in which one of the accesses writes a value to the shared memory location. Data races are one of the most common source of bugs in concurrent programs and are the cause of more serious problems like data corruption [Boehm 2011; Kasikci et al. 2013; Narayanasamy et al. 2007]. Absence of data races is often a pre-requisite

Authors' addresses: Mosaad Al Thokair, mosaada2@illinois.edu, University of Illinois at Urbana-Champaign, Urbana, USA; Minjian Zhang, minjian2@illinois.edu, University of Illinois at Urbana-Champaign, Urbana, USA; Umang Mathur, umathur@comp.nus.edu.sg, National University of Singapore, Singapore, Singapore; Mahesh Viswanathan, vmahesh@illinois.edu, University of Illinois at Urbana-Champaign, Urbana, USA.

2475-1421/2023/1-ART45
https://doi.org/10.1145/3571238

for the semantics of programs to be well defined [Boehm and Adve 2008; Zhivich and Cunningham 2009] and for compiler optimizations to be sound [Ševčík 2011; Ševčík and Aspinall 2008]. Sound dynamic data race prediction is a popular approach to identify such bugs in concurrent programs. Here one observes a single execution of the program, and the goal of the analysis is to see if the execution provides evidence for the presence of a data race in the program. This requires reasoning about alternate re-orderings of the events of the observed execution to determine if an execution with a data race is possible.

The simplest and most commonly used dynamic data race detection technique is based on Lamport's happens-before (HB) partial order [Lamport 1978]. The idea is to (implicitly) compute the HB partial order on the events of the observed execution and check if there are a pair of conflicting accesses to a shared memory location that are unordered by HB. Happens before based data race detection is known to be *sound* — the presence of HB unordered memory accesses is proof that the program has a data race. Early vector clock based algorithms [Fidge 1991; Mattern 1988] for happens before race detection have been improved over the years to the extent that it is the go-to data race detection in practice [Serebryany and Iskhodzhanov 2009; Serebryany et al. 2011]. However, the analysis has high runtime costs [Biswas et al. 2017; Bond et al. 2010; Marino et al. 2009] due to expensive metadata updates at each event, despite optimizations introduced [Flanagan and Freund 2009; Pozniansky and Schuster 2003]. This makes dynamic race detection suitable only for in-house testing, and makes an otherwise lucrative premise of data races *exceptions* [Adve 2010; Elmas et al. 2007] rather impractical.

One approach that attempts to reduce the analysis cost in HB race detection uses *sampling* [Bond et al. 2010]. The informal idea behind this approach is to sample some events of the observed trace, and analyze only the sampled subset of events to determine if the program has data races, with the hope being that the sampled subset will be small compared to the whole trace and that often it will be sufficient to find a race. Though the algorithm presented in [Bond et al. 2010] (Pacer) is experimentally shown to run faster than known deterministic race detection algorithms, its expected running time is linear and in the worst case it can be shown to be no better than a deterministic algorithm — there are examples on which, with non-zero probability, Pacer will analyze *all* the events in the trace.

The motivation behind this work is to explore the possibility of a sound race detection algorithm that analyzes, even in the worst case, a *sub-linear* number of events in an execution (i.e., "little o" of the length of the trace), but nonetheless has provable mathematical guarantees of precision. The hope is that such an algorithm will help scale sound dynamic race detection beyond in-house testing.

To achieve our goal, we investigate the design of a *property tester* [Goldreich 2017] for HB race detection. A (one-sided) (ϵ, δ)-property tester for a decision problem L is an algorithm A that meets the following obligation: on an input $x \in L$, A answers "yes" with probability 1, while on an input that is "ϵ-far" from any input in L, A answers "no" with probability at least $1 - \delta$. Notice that the definition of a property tester is based on a distance metric on input strings. The standard notion of distance used in property testing is hamming distance. Therefore, rephrasing and specializing to HB race detection, we have the following. A property tester for HB race detection is an algorithm A such that on any HB race-free execution σ, A answers "yes", and on any execution σ in which at least ϵ fraction of the events must be modified to obtain a race-free execution, A answers "no" with probability at least $1 - \delta$. Any sound and complete algorithm for HB race detection is, by definition, a property tester for race detection since it distinguishes between race-free and racy executions. However, a property tester has weaker obligations and solves a decision problem approximately —

on executions σ that have an HB race but at the same time are very close to a race-free execution, a property tester has no obligation to correctly classify them as having an HB race. This flexibility has enabled computer scientists to design extremely fast, but nonetheless useful, algorithms for a variety of decision problems [Goldreich 2017] and has led to sub-linear algorithm design being a vibrant field of study for the past 25 years.

In this paper we present a (ϵ, δ)-property tester **RPT** (Race Property Tester) for HB-race detection that provably examines only *constantly many* events in the observed program execution. More precisely, let t be an upper bound on the number of threads and h be the maximum number of locks held at any point in the trace of a concurrent program. Our property tester analyzes only $\widetilde{O}(t+h)$ events of a trace σ and correctly classifies them as having an HB-race with probability at least $1-\delta$, when σ is ϵ-far from race-free executions. Here $\widetilde{O}(\cdot)$ hides constant, $\ln(1/\delta)$ and $(1/\epsilon)$ factors. Notice that the number of events examined by our property tester only depends on the parameters t and h, which are often very small, regarded as constants, and is independent of the length of the input trace σ. Also note that, by design, **RPT** is a sound race detector —- whenever it flags the presence of a race, the execution has a real race.

Our property tester is a very simple, almost naïve, algorithm, which maybe a feature when it comes to implementing it and deploying it in practice. It works as follows. If the input trace is "short" (defined precisely in Algorithm 3), run a deterministic race detector such as FastTrack [Flanagan and Freund 2009]. Otherwise, sample, uniformly at random, $O(\frac{\ln(1/\delta)}{\epsilon})$ sub-traces of input σ, each of length $O(\frac{t+h}{\epsilon})$, and check that none of the sampled sub-traces contain an HB race. If they do, the algorithm declares the input trace σ to be racy, and otherwise, declares it to be race-free. To check whether any of the sampled sub-traces contain an HB-race, we could use any HB race detection algorithm. In our experiments, we use the FastTrack algorithm [Flanagan and Freund 2009] that uses vector clocks and employs the *epoch* optimization.

The challenge, as for most randomized algorithms, is to prove that this simple algorithm is correct. This means we need to show that, if the input σ, observed by running a multi-threaded program, is ϵ-far from every race-free execution, then our algorithm will find an HB race with high probability. The crux of our correctness proof is in the following observation. We show that any trace σ that is ϵ-far from every race-free execution, has *many, short* (of length $\widetilde{O}(t+h)$) sub-traces that contain an HB race. Thus, by sampling a few different sub-traces independently, using standard arguments, one can show that the algorithm's answer is correct with high probability.

We expect that the promise of a constant runtime overhead race detector will be useful for practitioners. Given that the formal guarantee of our property tester is parameterized by ϵ and δ, it is natural to ask how a practitioner should use our algorithm. After all, on the face of it, it seems like we need to know how far an observed execution σ is from race-free traces! **RPT**, like most dynamic techniques, is primarily an approach to find bugs. Therefore, our recommendation is to view ϵ and δ as adjustable parameters that a software developer can progressively decrease based on resource availability and past experiences rather than obligated parameters that one must decide for each program. If at any stage a race is discovered then debugging can begin. On the other hand, if no race is discovered even as ϵ and δ are decreased, then the software developer can be more confident about the reliability of the code based on the mathematical statements that back the correctness of the property tester. Finally, our experimental results show that even when the ϵ used in the algorithm is a poor measure of the actual distance of the input σ from race-free executions, the algorithm detects HB races reasonably often.

RPT has been implemented. We have evaluated the performance of **RPT** on benchmark examples and compare it against the state-of-the-art deterministic (FastTrack) and sampling-based (Pacer) HB race detector, to see if the theoretical promises hold. We choose not to compare against techniques which employ a two-phase hybrid analysis [Choi et al. 2002; Jeong et al. 2019; Kasikci et al. 2013] because our innovations are primarily in dynamic analysis which is orthogonal to these approaches. Our techniques can be modularly plugged into hybrid race detection techniques to reduce the running time of the dynamic analysis phase and like RaceMob, can benefit from an additional static analysis phase (see Section 5 for more details).

Preliminary results suggest that **RPT** is a promising approach. When compared with FastTrack and Pacer, **RPT** has the lowest running time among the 3. Moreover, **RPT**'s competitive advantage grows as the length of the trace increases. In fact, **RPT**'s running time flattens out as the trace length grows in our experiments. Despite that, our results show that **RPT** reports a race quite often. This is especially true when considering traces that have a large proportion of race warnings — when the number of race warnings reported by FastTrack divided by trace length is at least 10^{-5}, **RPT** detects races with at least the same probability if not better than Pacer. This is despite the fact that **RPT** in these experiments was run with a large value for ϵ, namely 0.01. Detailed experimental results are presented in Section 4.

2 BACKGROUND AND PRELIMINARIES

In this section we discuss preliminary notations and also recap relevant background on data race detection and property testing.

2.1 Traces and Data Races

Concurrent Program Traces and Events. The focus of our work is dynamic race detection, where one monitors the execution *trace* of a concurrent program, observing *events* generated by different threads, and analyzing it to infer the presence of a data race. Each event is labeled with a tuple $\langle t, o\rangle$ (denoted simply as $e = \langle t, o\rangle$), where t is the unique identifier of the thread that performs e and o represents the operation associated with e. For our exposition, the operation o can be one of [1]: (a) read/write access to a memory location x (i.e., $o = \mathtt{r}(x)$ or $o = \mathtt{w}(x)$), or (b) lock-based synchronization — acquire/release of a lock ℓ (i.e., $o = \mathtt{acq}(\ell)$ or $o = \mathtt{rel}(\ell)$). We use the notation $\mathsf{thr}(e) = t$ and $\mathsf{op}(e) = o$ for the event $e = \langle t, o\rangle$. A trace σ can thus be viewed as a sequence of such events (denoted Events_σ). We denote by $\mathsf{Threads}_\sigma$, Locks_σ and Mem_σ to denote the set of threads, locks and memory locations that appear in the trace σ. We use $|\sigma|$ to denote the length of σ.

Sub-traces. Traces, as mentioned, are a sequence of events. We will adopt the convention that the first event in the sequence has index 0. Thus, a trace of length n is of the form $\sigma = e_0 e_1 \cdots e_{n-1}$. The i^{th} event of trace σ (namely e_i) will also be denoted as $\sigma[i]$. A *sub-trace* $\sigma[i, j] = e_i e_{i+1} \cdots e_{j-1}$ is the subsequence of σ of length $j - i$ from index i to index $j - 1$. When $j \leq i$, we adopt the convention that $\sigma[i, j]$ is the empty sequence ε. The *concatenation* of traces $\sigma_1 = e_0 \cdots e_{n-1}$ and $\sigma_2 = f_0 \cdots f_{m-1}$ is the sequence $e_0 \cdots e_{n-1} f_0 \cdots f_{m-1}$ of length $n + m$ and will be denoted by $\sigma_1\sigma_2$.

Well formed Traces and Sub-traces. Executions of concurrent programs, in addition to being a sequence of events of the form described above, satisfy some properties. A trace $\sigma = e_0 \cdots e_{n-1}$ is *well formed* if critical sections on the same lock do not overlap. That is, for every $j < n$, if $e_j = \langle t, \mathtt{rel}(\ell)\rangle$ releases lock ℓ, then there is an $i < j$ such that $e_i = \langle t, \mathtt{acq}(\ell)\rangle$ and further, for

[1] We omit other synchronizations like forks and joins or wait-notify, for simplicity of presentation. It is straightforward to accommodate them, and all our results apply to the more general setting too. Further, our experiments do account for such events in the benchmarks.

every $i < k < j$, $\mathsf{op}(e_k) \notin \{\mathtt{acq}(\ell), \mathtt{rel}(\ell)\}$[2]. Henceforth, we will assume traces to be well formed. A sequence η is a *well formed sub-trace* if there is a well formed trace σ and indices i and j such that $\eta = \sigma[i, j]$. Finally, in a well formed sub-trace $\eta = e_0 e_1 \cdots e_{n-1}$, we say that lock ℓ is *held* by thread t at j (for $0 \leq j \leq n$) if either (a) there is an $i < j$ such that $e_i = \langle t, \mathtt{acq}(\ell)\rangle$ and for every $i < k < j$, $\mathsf{op}(e_k) \neq \mathtt{rel}(\ell)$, or (b) there is an $i \geq j$ such that $e_i = \langle t, \mathtt{rel}(\ell)\rangle$ and for every $j \leq k < i$, $\mathsf{op}(e_k) \neq \mathtt{acq}(\ell)$. We say that lock ℓ is held at the beginning (resp. end) of a non-empty well-formed sub-trace η if ℓ is held at 0 (resp. $|\eta|$).

Data Races. A trace is said to have a data race if two different threads access the same memory location without explicit synchronization in between. This is formalized in terms of Lamport's Happens-Before (HB) partial order [Lamport 1978], which we recap next, while generalizing this notion to well formed sub-traces.

Definition 1 (Happens-Before)**.** For a well formed sub-trace σ, the happens-before partial order induced by σ, denoted $<^{\sigma}_{\mathsf{HB}}$, is the smallest binary relation on Events_σ such that for any two events $e_1 \neq e_2 \in \mathsf{Events}_\sigma$, we have $e_1 <^{\sigma}_{\mathsf{HB}} e_2$ if e_1 occurs before e_2 in σ, and one of the following holds:

(program-order) $\mathsf{thr}(e_1) = \mathsf{thr}(e_2)$.

(lock synchronization) e_1 releases a lock ℓ, which e_2 acquires (i.e., $\mathsf{op}(e_1) = \mathtt{rel}(\ell)$ and $\mathsf{op}(e_2) = \mathtt{acq}(\ell)$), or

(transitivity) there is an event e_3 such that $e_1 <^{\sigma}_{\mathsf{HB}} e_3 <^{\sigma}_{\mathsf{HB}} e_2$.

A pair of events (e_1, e_2) of σ is said to be *conflicting* if $\mathsf{thr}(e_1) \neq \mathsf{thr}(e_2)$, both are memory access events on a common location (say) x with at least one of them being a write. An *HB-race* in σ is a pair (e_1, e_2) of conflicting events in σ such that neither $e_1 <^{\sigma}_{\mathsf{HB}} e_2$ nor $e_2 <^{\sigma}_{\mathsf{HB}} e_1$. Finally, σ is said to have a data race if there is an HB-race in it; otherwise σ is said to be race-free. [3]

An important observation about the HB partial order is that it is 'context-free', i.e., whether a pair of events is ordered by HB only depends on the events that appear between the two events in the trace. This is formalized in Proposition 2.1 and will be crucially exploited by our randomized algorithm.

Proposition 2.1. *Let $\sigma = e_0 e_1 \cdots e_{n-1}$ be a well formed sub-trace. For any pair of indices $i < j$, $e_i <^{\sigma}_{HB} e_j$ if and only if $e_i <^{\sigma[i,j+1]}_{HB} e_j$.*

Proof. Follows from the definition of HB. □

2.2 Dynamic Data Race Detection

Data races can be detected in a streaming fashion, by processing events one-by-one, updating meta-data and checking for races at each event of interest, as shown in the general outline Algorithm 1. Most of the algorithms known for detecting data races dynamically [Elmas et al. 2007; Flanagan and Freund 2009; Itzkovitz et al. 1999; Pozniansky and Schuster 2003] adhere to this generic outline, and differ only on the precise details of the data maintained by the algorithm or the implementation of the functions `InitMetadata` and `checkRaceAndUpdateMetadata`.

[2]We assume that locks are not re-entrant; all our results can nevertheless be extended in the presence of such locks.
[3]In general, the absence of HB-races does not imply the absence of predictive data races. The notion of race-freedom in this work means the absence of HB-races.

Algorithm 1: *Outline for dynamic data race detection*

Input: Trace σ

```
1 InitMetadata()
2 for e in σ do  checkRaceAndUpdateMetadata(e)
```

Algorithm 2: *Vector clock updates*

```
1  function InitMetadata()
2      for t ∈ Threads · do
3          C_t := ⊥[1/t]
4      for x ∈ Mem · do
5          R_x := ⊥; W_x := ⊥
6      for ℓ ∈ Locks · do
7          L_ℓ := ⊥
8  handler acquire(t, ℓ)
9      C_t ← C_t ⊔ L_ℓ
10 handler release(t, ℓ)
11     L_t ← C_t
12 handler read(t, x)
13     check W_x ⊑ C_t
14     R_x ← C_t
15 handler write(t, x)
16     check R_x ⊑ C_t
17     check W_x ⊑ C_t
18     W_x ← C_t
```

Here, we discuss the most popular algorithm Djit [Itzkovitz et al. 1999] which uses vector clocks for assigning vector timestamps [Fidge 1991; Mattern 1988] to events and uses them to check for HB-races; Djit has further been optimized in subsequent works including Djit+ [Pozniansky and Schuster 2003] and FastTrack [Flanagan and Freund 2009]. A vector timestamp is a map $V : \mathsf{Threads}_\sigma \to \mathbb{N}$ that assigns a natural number to every thread of the trace σ being analyzed. The ordering on two timestamps is defined as $V_1 \sqsubseteq V_2 \triangleq \forall t \in \mathsf{Threads}_\sigma, V_1(t) \leq V_2(t)$. In essence, the Djit algorithm computes a vector timestamp V_e for each event e such that for any two events $e_1 \neq e_2$, $e_1 \leq^\sigma_{\mathsf{HB}} e_2$ iff $V_{e_1} \leq^\sigma_{\mathsf{HB}} V_{e_2}$. However, instead of actually storing the timestamps of each event, the algorithm uses *vector clocks* to store a small number of timestamps. Overall, the algorithm maintains, a vector clock $\mathbb{C}_t$, $\mathbb{L}_\ell$, $\mathbb{R}_x$ and $\mathbb{W}_x$ for every $t \in \mathsf{Threads}_\sigma$, $\ell \in \mathsf{Locks}_\sigma$ and $x \in \mathsf{Mem}_\sigma$.

Algorithm 2 summarizes how vector clocks are initialized and updated — the function `checkRaceAndUpdateMetadata` calls the appropriate handler based on the operation performed in the event (the timestamp $\bot$ is $\lambda u, 0$). The lines 13, 16 and 17 perform the race detection checks. We omit the increment '$\mathbb{C}_t \leftarrow \mathbb{C}_t[\mathbb{C}_t(t) + 1/t]$' at the end of each handler.

While HB-based dynamic analysis is considered the go-to method for data race detection in practice [Serebryany and Iskhodzhanov 2009; Serebryany et al. 2011], it is known to add high runtime costs [Biswas et al. 2017; Bond et al. 2010; Marino et al. 2009] due to expensive metadata updates at each event, despite optimizations introduced [Flanagan and Freund 2009; Pozniansky and Schuster 2003]. This makes dynamic race detection suitable only for in-house testing.

2.3 Property Testing

A property tester [Goldreich 2017] is an algorithm that solves a decision problem under a promise setting. Another way to think about it is that it solves a decision problem "approximately". It is typically a randomized algorithm. A property tester for a decision problem characterized by a language L is an algorithm that provides the following guarantees: on a input $x \in L$ the algorithm answers *yes* with high probability, and on an input x that is "far" from anything in L, it answers *no* with high probability. Thus, to define a property tester precisely, we need to identify a notion

of distance between elements of the space of inputs. The most commonly used distance metric is *hamming distance* which we define first.

Definition 2 (Hamming Distance)**.** For sequences $u, v \in \Sigma^*$ over alphabet Σ, the hamming distance between u and v ($d_{\mathsf{h}}(u, v)$) is defined as follows:

$$d_{\mathsf{h}}(u, v) = \begin{cases} |\{i \mid u[i] \neq v[i]\}| & \text{if } |u| = |v| \\ \infty & \text{otherwise} \end{cases}$$

For $u \in \Sigma^*$ and $L \subseteq \Sigma^*$, $d_{\mathsf{h}}(u, L) = \inf_{v \in L} d_{\mathsf{h}}(u, v)$.

Note that for $u, v \in \Sigma^*$, either $d_{\mathsf{h}}(u, v) = \infty$, when $|u| \neq |v|$, or $d_{\mathsf{h}}(u, v) \leq |u| = |v|$, when $|u| = |v|$.

Having defined the notion of distance between an input and a language, we can define precisely what a property tester is. In this paper, we only consider *one-sided* testers and so we specialize the definition to this case.

Definition 3 (Property Tester)**.** A (one-sided) (ϵ, δ) property tester for a problem L is a randomized algorithm A such that on any input x, A's output satisfies the following property.

(a) If $x \in L$, $A(x) =$ yes with probability 1.

(b) If $d_{\mathsf{h}}(x, L) \geq \epsilon|x|$, $A(x) =$ no with probability at least $1 - \delta$.

We note some observations about Definition 3. On inputs x that are not in L but are "close" (i.e. $d_{\mathsf{h}}(x, L) < \epsilon|x|$), the property tester may answer either yes or no, without violating its obligation. In this sense, a property tester is an approximate algorithm for a decision problem. Second, since we are considering one-sided testers, we can arrive at the following conclusions about an input based on the tester's response. If $A(x) =$ no then we can conclude that $x \notin L$. On the other hand, if $A(x) =$ yes then we cannot conclude anything definite about the membership of x in L.

3 A PROPERTY TESTER FOR RACE DETECTION

Our randomized algorithm **RPT** for dynamic race detection is simple and straightforward:

(1) Sample uniformly at random r sub-traces, each having length k, of the observed trace σ.

(2) If any of the sampled sub-traces has a data race, declare σ to have a data race.

(3) If none of the sampled sub-traces have a data race, declare σ to be race-free.

To complete the description of **RPT**, we need to answer the following questions. How many sub-traces should the algorithm sample (parameter r)? What should the length of sampled sub-traces be (parameter k)? Obviously these parameters are set to ensure that the resulting algorithm is a property tester for race detection. The correctness proof is the most critical piece, and also the most technically challenging part of the algorithm itself. Our description in this section will use vague terms like "many", "very far", "short", "long" etc. These terms will be precisely characterized by parameters in our formal theorem and lemma statements, but our use of these informal terms in the text helps illuminate the main ideas behind the correctness argument without getting lost in the technical details.

Before we begin outlining the correctness proof, let us examine our algorithm template and establish some straightforward facts. First notice that no matter what values we set for parameters r and k, the algorithm is *sound* — if the algorithm declares a trace σ to have a data race then it does

indeed have a data race. This is because of the "context-free" property of HB-races articulated in Proposition 2.1 — whether events e_1 and e_2 of σ are in HB-race only depends on the events in the sub-trace of σ that starts with e_1 and ends with e_2. This means that a race-free execution σ will be declared to be correct with probability 1. Thus, to establish correctness, our obligation is to find values for r and k that ensure that on traces which are very far from any race-free execution, the algorithm discovers a sub-trace with an HB-race with high probability. Second, to check if a sampled sub-trace has a data race, we could use any algorithm to check for HB-races [Elmas et al. 2007; Flanagan and Freund 2009; Itzkovitz et al. 1999; Kini et al. 2018; Kulkarni et al. 2021; Pozniansky and Schuster 2003]. In our implementation, we use FastTrack [Flanagan and Freund 2009], but this could be replaced by any improvements to HB-race checking to yield faster running times.

3.1 Proof of Correctness

Let us now present an overview of our correctness proof. It crucially relies on our main theorem (Theorem 3.3) which says that if the input trace σ is far from any race-free execution (with respect to hamming distance), then there are many short sub-traces that contain an HB-race. The measure that characterizes "short" in this theorem will be taken to be the value of k. Observe that if Theorem 3.3 guarantees that presence of many k-length sub-traces that have a race, then by analyzing a randomly chosen k-length sub-trace guarantees that we will discover a race with some probability. Therefore if we repeat this experiment a few times (namely r times), we can ensure that we discover a race with high probability. Here the number of samples r will be set based on the chance that a single k-length sub-trace is racy, using standard counting arguments. The main theorem (Theorem 3.3) itself is established by first observing that a trace σ which is far from any race-free execution has many (not necessarily short) *disjoint* sub-traces that have an HB-race. This is the content of Lemma 3.2 which is proved as follows. We show that if σ has very few disjoint sub-traces that are racy, then σ can be transformed by changing very few events into a race-free execution. Since we know σ is far, we can conclude that it has many disjoint races. Finally, to show that few disjoint racy sub-traces means closeness to a race-free execution, we need Lemma 3.1 which proves that any pair of race-free sub-traces σ_1 and σ_2 can be combined into a larger race-free sub-trace, provided we paste a short sub-trace μ between σ_1 and σ_2.

Having provided an overview of our proof, we are ready to present the technical details. We start with a technical lemma that shows that for any well formed sub-traces σ_1 and σ_2, there is a *short* trace μ such that the concatenated sub-trace $\sigma_1\mu\sigma_2$ is well formed with the property that every event in σ_1 is HB-before any event in σ_2. This will be used later to show that in traces that are far from race-free executions, there are many disjoint racy sub-races.

Lemma 3.1. *Let σ_1 and σ_2 be well formed subtraces over threads T. Let h be an upper bound on the number of locks held at the end of σ_1 and at the beginning of σ_2. There exists a sub-trace $\mu(\sigma_1, \sigma_2)$ such that $|\mu(\sigma_1, \sigma_2)| \leq 4|T| + 2h$, and $\sigma = \sigma_1\mu(\sigma_1, \sigma_2)\sigma_2$ is well formed. Moreover, for any events $e_1 \in \mathsf{Events}_{\sigma_1}$ and $e_2 \in \mathsf{Events}_{\sigma_2}$, we have $e_1 <^{\sigma}_{HB} e_2$.*

Proof. Let LH_1 be the set of locks held at the end of σ_1 and let LH_2 be the set of lock held at the beginning of σ_2. Notice that $|\mathsf{LH}_1| \leq h$ and $|\mathsf{LH}_2| \leq h$. Without loss of generality, let us assume that $\ell_* \in \mathsf{Locks}_{\sigma_1} \cup \mathsf{Locks}_{\sigma_2}$. $\mu(\sigma_1, \sigma_2)$ is the following sequence of events in the given order.

(1) If ℓ_* is held by thread t_* at the end of σ_1 then start with $\langle t_*, \mathtt{rel}(\ell_*)\rangle$.

(2) For each thread $t \in T$, add the sequence $\langle t, \mathtt{acq}(\ell_*)\rangle\langle t, \mathtt{rel}(\ell_*)\rangle$. After adding such a sequence for each thread, *repeat* this sequence again. That is, once again, for every thread $t \in T$, add the sequence $\langle t, \mathtt{acq}(\ell_*)\rangle\langle t, \mathtt{rel}(\ell_*)\rangle$.

(3) For each lock $\ell \in \mathsf{LH}_1 \setminus \{\ell_*\}$ that is held by thread t at the end of σ_1, add the event $\langle t, \mathtt{rel}(\ell)\rangle$.

(4) For each lock $\ell \in \mathsf{LH}_2$ held by thread t at the beginning of σ_2, add the event $\langle t, \mathtt{acq}(\ell)\rangle$.

Observe that the number of events added in step 1 + step 3 is at most h. Similarly, the number events added in step 4 is at most h. Finally, the number of events added in step 2 is $4|T|$. Putting all of this together, proves that $|\mu(\sigma_1, \sigma_2)| \leq 4|T| + 2h$. Next, the order in which events are added in $\mu(\sigma_1, \sigma_2)$ ensures that in σ each lock is held by at most one thread at any given time, which means that σ is well formed. Moreover, the set of locks held at the beginning of σ_2 in σ is exactly LH_2. Finally, the events added in step 2 ensure that for any events $e_1 \in \mathsf{Events}_{\sigma_1}$ and $e_2 \in \mathsf{Events}_{\sigma_2}$, we have $e_1 <^{\sigma}_{\mathsf{HB}} e_2$. □

Remark. It is worth noting an important consequence of Lemma 3.1 that we will exploit in our proof. Observe that the sub-trace $\mu(\sigma_1, \sigma_2)$ constructed in the proof has no data access events. Thus, if σ_1 and σ_2 are race-free, then so is $\sigma_1\mu(\sigma_1, \sigma_2)\sigma_2$.

For the rest of this section, let fix a set of threads T, a set of locks L, and a set of memory locations M. Let RF be the set of all well formed traces over T, L and M that are race free. That is,

$$\mathsf{RF} = \{\sigma \text{ race free} \mid \mathsf{Threads}_\sigma \subseteq T,\ \mathsf{Locks}_\sigma \subseteq L,\ \mathsf{Mem}_\sigma \subseteq M\}.$$

Observe that for any trace σ, $d_{\mathsf{h}}(\sigma, \mathsf{RF}) \leq |\sigma|$. This is because we can always pick $\eta \in \mathsf{RF}$ of the same length as σ, by ensuring that either η only events performed by a single thread, or has no write events, etc.

Let us also assume that h is an upper bound on the number of locks held at any point in a trace; in the worst case $h = |L|$, but typically h is much smaller than $|L|$. Finally, let us fix $m = 4|T| + 2h$.

Lemma 3.1 allows one to show that if a trace σ is very far from the set RF (as measured by parameter ϵ), then there are many *disjoint* sub-traces of σ that contain a pair of events that are in HB-race. In other words, if σ is far from any race-free trace, then there are many disjoint witnesses that demonstrate that σ has a race.

Lemma 3.2. *Let σ be a trace of length n such that $d_{\mathsf{h}}(\sigma, \mathsf{RF}) \geq \epsilon n$. There is an integer $u \geq \frac{\epsilon n}{m}$ and an increasing sequence of indices $0 = i^1_1 < i^2_1 < i^1_2 < i^2_2 < \cdots < i^1_u < i^2_u \leq n$ of length $2 \cdot u$ such that each sub-trace $\sigma[i^1_j, i^2_j]$ $(1 \leq j \leq u)$ has an HB-race.*

Proof. Let us construct an increasing sequence of indices as follows. Take $i^1_1 = 0$. The remaining indices are inductively defined as follows. Assuming $i^1_1, i^2_1, \ldots i^1_j$ have been defined. Then,

$$i^2_j = \min\{k \leq n \mid \sigma[i^1_j, k] \text{ has a race}\}.$$

In the above equation, if the set over which we are taking a minimum is empty (i.e., $\sigma[i^1_j, n]$ is race-free) then our construction of the sequence ends. Next, assuming $i^1_1, i^2_1, \ldots i^1_j, i^2_j$ are defined, we take $i^1_{j+1} = i^2_j + m - 1$, provided $i^2_j + m - 1 < n$; again if $i^2_j + m - 1 \geq n$, then we stop the construction.

Notice that by definition, our sequence is increasing and each sub-trace $\sigma[i^1_j, i^2_j]$ has an HB-race. To complete the proof of the lemma, all we need to argue is that the sequence we have constructed

is long, i.e., if i_u^2 is the last index constructed by the above sequence, then $u \geq \frac{\epsilon n}{m}$. We will use the fact that $d_{\mathsf{h}}(\sigma, \mathsf{RF}) \geq \epsilon n$ to establish this.

Suppose we have constructed the sequence $0 = i_1^1 < i_1^2 < \cdots < i_u^1 < i_u^2$ as above. Since we stopped at i_u^2, it means that either $i_u^2 + m - 1 \geq n$ or $\sigma[i_u^2 + m - 1, n]$ is race-free. Let us define the sub-trace σ_j as $\sigma[i_j^1, i_j^2 - 1]$. Notice by definition of i_j^2 this means that σ_j is race free. Consider the trace σ' defined as follows.

$$\sigma' = \sigma_1 \mu(\sigma_1, \sigma_2) \sigma_2 \mu(\sigma_2, \sigma_3) \cdots \sigma_u \mu_*.$$

Here $\mu(\sigma_j, \sigma_{j+1})$ is the sequence guaranteed by Lemma 3.1 for σ_j and σ_{j+1}, and μ_* is defined as follows: if $i_u^2 + m - 1 \geq n$, then μ_* is some race-free trace of length $n - i_u^2 + 1$ and if $\sigma[i_u^2 + m - 1, n]$ is race-free then $\mu_* = \mu'_* \sigma[i_u^2 + m - 1, n]$ where $\mu'_* = \mu(\sigma_u, \sigma[i_u^2 + m - 1, n])$. Without loss of generality, we will assume that sub-traces of the form $\mu(\sigma_j, \sigma_{j+1})$ guaranteed by Lemma 3.1 are of length *exactly* m — if they are shorter, we can pad them with events.

Notice that $|\sigma'| = n = |\sigma|$ and σ' is (by construction) race-free because of the remark after Lemma 3.1. Moreover $d_{\mathsf{h}}(\sigma, \sigma') \leq um$, since at most m events are changed in each μ-sub-trace. Since σ' is race-free and $d_{\mathsf{h}}(\sigma, \mathsf{RF}) \geq \epsilon n$, we have $um \geq \epsilon n$ which means that $u \geq \frac{\epsilon n}{m}$. This completes the proof of the lemma. □

Lemma 3.2 guarantees the presence of many disjoint, racy sub-traces. However, that by itself is not enough to get an efficient property tester for data race detection. In particular, Lemma 3.2 provides no bounds on the length of the racy sub-traces it identifies. If we do not strengthen Lemma 3.2, the only bound we can get on the sample length k in our template algorithm would be $|\sigma|$, which would make our property tester no more efficient than a deterministic race detector. Our main theorem, established next, shows that, for sufficiently long traces, there are many racy sub-traces of *short* length, when a trace σ is far from any race-free trace. The proof uses Lemma 3.2. This will enable us to bound k and get good asymptotic bounds.

Theorem 3.3. *Let σ be a trace of length n such that $d_{\mathsf{h}}(\sigma, \mathsf{RF}) \geq \epsilon n$. In addition, let $n \geq (12m)/\epsilon$. Then there are at least $2(\epsilon n)/15$ sub-traces of σ of length $4m/\epsilon$ that contain an HB-race.*

Proof. Let $0 = i_1^1 < i_1^2 < i_2^1 < \cdots i_u^1 < i_u^2 \leq n$ be the increasing sequence of indices guaranteed by Lemma 3.2 such that each subtrace $\sigma[i_j^1, i_j^2]$ has a data race. Consider the set

$$\mathsf{long} = \{\sigma[i_j^1, i_j^2] \mid i_j^2 - i_j^1 \geq (2m/\epsilon)\}.$$

Since each element of long is a sub-trace of σ, the sum of the lengths of such sub-traces is $\leq n$. On the other hand, each sub-trace in long is of length at least $2m/\epsilon$ and so the sum of the lengths is at least $\frac{2m|\mathsf{long}|}{\epsilon}$. Putting it together, we get

$$n \geq \sum_{\rho \in \mathsf{long}} |\rho| \geq \frac{2m|\mathsf{long}|}{\epsilon} \quad \Rightarrow \quad |\mathsf{long}| \leq \frac{\epsilon n}{2m}.$$

Since $u \geq (\epsilon n)/m$, we have $|\mathsf{short}| \geq (\epsilon n)/(2m)$, where

$$\mathsf{short} = \{\sigma[i_j^1, i_j^2] \mid i_j^2 - i_j^1 < (2m/\epsilon)\}.$$

The above counting argument guarantees that the number of disjoint short racy traces is large. However, we can improve the bound further if we allow sub-traces to overlap. This improvement will help improve the running time of our property tester in turn.

Consider a sub-trace $\eta = \sigma[i_j^1, i_j^2] \in \text{short}$. Let $|\eta| = s$; we know $s \leq (2m/\epsilon)$. Each such sub-trace η (unless $j = 1$ or $j = u$) is a sub-trace of $(4m/\epsilon - s)$ sub-traces of σ of length $4m/\epsilon$. The reason is because any sub-trace of σ of length $4m/\epsilon$ that starts at a position in the interval $[i_j^2 - 4m/\epsilon, i_j^1]$ contains η. Since $s \leq 2m/\epsilon$, we have each such η is contained in at least $4m/\epsilon - s \geq (2m/\epsilon)$ sub-traces of length $4m/\epsilon$. Note, that each sub-trace of length $4m/\epsilon$ that contains such an $\eta \in \text{short}$ has a data race. On the other hand, any sub-trace ρ of σ of length $4m/\epsilon$ can contain at most $(5m/\epsilon)/m$ sub-traces from short. This can be argued as follows. Suppose ρ contains a sub-traces in short. Since each sub-trace in short is separated by m positions (see proof of Lemma 3.2), the sum of the lengths of all short sub-traces plus their intervening gaps is at least $(a-1)m$. Now $|\rho| = 4m/\epsilon$. Thus, $(a-1)m \leq 4m/\epsilon$, which means that $a \leq (4/\epsilon) + 1 \leq 5/\epsilon$. In other words, each sub-trace of σ of length $4m/\epsilon$ contains at most $5/\epsilon$ of the sub-traces in short. Putting these observations together, we see that the number of sub-traces ρ of σ of length $4m/\epsilon$ that contain a data race is at least

$$\frac{2m}{\epsilon} \cdot \frac{\epsilon}{5} \cdot [|\text{short}| - 2] \geq \frac{2m}{5} \left[\frac{\epsilon n}{2m} - 2\right].$$

In the above equation, "-2" is to discount $\sigma[i_1^1, i_1^2]$ and $\sigma[i_u^1, i_u^2]$ if they belong to short. Assuming $n \geq (12m)/\epsilon$, we have $(\epsilon n)/(2m) - 2 \geq (\epsilon n)/(3m)$. Thus, the number of sub-traces of length $4m/\epsilon$ that contain a data race is at least $\frac{2m}{5} \cdot \frac{\epsilon n}{3m} = \frac{2\epsilon n}{15}$. □

Theorem 3.3 helps complete the description of our algorithm **RPT**. Our property tester will pick sub-traces of length $4m/\epsilon$, the parameter used in Theorem 3.3. There are n such sub-traces, since each starting position identifies such a sub-trace. From Theorem 3.3 it follows that the probability that a random sub-trace of length $k = 4m/\epsilon$ has a data race (when σ is ϵ-far from RF), is at least $2\epsilon/15$. If we pick $r = \frac{15\ln(1/\delta)}{2\epsilon}$, the probability we will not detect is data race is at most

$$(1 - 2\epsilon/15)^r < e^{-\ln(1/\delta)} = \delta.$$

3.2 Pseudocode for RPT

Let us conclude this section by presenting a pseudo-code for our property tester (Algorithm 3). Recall that the algorithm samples $r = \frac{15\ln(1/\delta)}{2\epsilon}$ sub-traces of input σ of length $k = 4m/\epsilon$, and checks if any of the sampled sub-traces have an HB-race. Notice that sampling a sub-trace of length k is the same as picking a starting index i with the understanding that the sampled sub-trace is $\sigma[i, i+k]$. Thus, sampling r sub-traces is the same as picking r starting indices (Line 6). Consider two sub-traces $\sigma[i_1, i_1+k]$ and $\sigma[i_2, i_2+k]$ of σ that overlap. That is, wlog $i_1 < i_2 < i_1+k$. Notice that by the definition of HB partial order, if the sub-trace $\sigma[i_1, i_2+k]$ is race-free then both $\sigma[i_1, i_1+k]$ and $\sigma[i_2, i_2+k]$ are race-free. Thus, we can merge sampled sub-traces that are overlapping without sacrificing on our ability to detect races. Therefore, in Line 7, we merge the overlapping sub-traces to get a smaller set of sampled sub-traces, but with the possibility of the sampled sub-traces being longer that k. This step reduces the total number of events our algorithm will process. After this initial pre-processing step, the algorithm proceeds as follows. When an event is the start of a sampled sub-trace, the meta-data is reset so that there is a fresh start to race detection. In addition, whenever an event is in our sampled sub-trace we call the function `checkRaceAndUpdateMetadata` which in turn calls the appropriate handler in Algorithm 2 based on the operation performed by the event. When an event is not in any of our sampled sub-traces, no checking and meta-data updates take place.

We remark that **RPT** can be easily extended to synchronisation primitives such as those due to atomic/volatile accesses, as well as synchronizations such as fork-join, wait-notify or barriers. This is because for such synchronizations, the "context-free" property (Proposition 2.1) holds and the

Algorithm 3: *Property Tester for Checking HB-races*

Parameters: $\epsilon, \delta \in [0, 1]$

Input: Trace σ

```
1  InitMetadata()
2  m ← 4|T| + 2h ; k ← 4m/ε; r ← 15 ln(1/δ)/(2ε)
3  if |σ| < 12m/ε then
4      Run Algorithm 1 on σ
5  else
6      I ← Sample r indices in [0, n − k]
7      S ← mergeSubtraces ({σ[i, i + k]}_{i∈I})
8      for e in σ do
9          if e is the start of a sub-trace in S then
10             resetData()
11         if e is in an sub-trace in S then
12             checkRaceAndUpdateMetadata(e)
```

classic FastTrack algorithm can be easily extended for such synchronizations (as also observed in [Flanagan and Freund 2009]).

4 EXPERIMENTAL EVALUATION

We evaluated the practical feasibility of our property tester by implementing **RPT** and comparing against the go-to deterministic race detection algorithm FastTrack due to Flanagan and Freund [Flanagan and Freund 2009], and against Pacer [Bond et al. 2010], which is the state-of-the art sampling based race detection algorithm. Each of these tools has a different philosophy and solves a slightly different problem — FastTrack processes every event in the trace and reports *all* HB races; Pacer promises that every HB race has an equal, non-zero probability of being reported; and in contrast, **RPT** is engineered to sample just enough to ensure that we can mathematically prove that *at least one race* will be reported with high probability, when the trace is far from being race free. Their comparison is not a like-for-like comparison. As a consequence, the experiments in this section are not to suggest that one tool is better than another. They are there to merely help one understand the likely performance of **RPT** on practical programs: is the running time low as promised by the theory, how often does it report at least one race, does it work only when there are many races, what types of races arise in practice and are there some that **RPT** will always fail on. If we only report the performance of **RPT** it becomes difficult to gauge how reasonable it is, and therefore, we report the performance of both FastTrack and Pacer to serve as a baseline.

The rest of this section is organized as follows. After explaining our implementation and setup (Section 4.1), and characteristics of our extracted traces (Section 4.2), we present our experimental analysis in two parts. In the first part (Section 4.3), we present experiments that help understand how effective **RPT** is in reporting at least one race. We look at how the running time of **RPT** changes with trace characteristics like length and number of threads + locks held. Next, our theoretical analysis only guarantees reporting at least one race with high probability whenever the observed execution is far from a race-free execution. Therefore, we ask how does **RPT** perform on traces with very few races, since the number of races in a trace is an upper bound on the distance of a trace from a race-free trace. Finally, recall that **RPT** relies on sampling short sub-traces and can

only detect races between events that are not far apart. So we ask, how often do traces have races that are close by, and how does **RPT** perform on traces where most or all races are between events that far apart (when compared to the length of sub-traces sampled by **RPT**). The second part of our analysis (Section 4.4) reports on the performance of **RPT** as measured by the number of races detected. Note that **RPT** is not engineered to report all races or most races or even every race with some probability. It only guarantees reporting *some* race with high probability, when the trace is far from being race-free. Thus, these experiments are not consistent with the design of **RPT**, but our objective here is to understand if there are some types of races that will escape detection with **RPT**.

4.1 Implementation and Setup

We have implemented our algorithm **RPT**, the FastTrack algorithm and Pacer's sampling algorithm in Java. Our implementation is designed to run all three algorithms on the exact same trace to allow for a fair comparison and reduce noise in the results introduced by the runtime thread-scheduler.

Pacer. Pacer is the state-of-the art sampling based race detection technique for detecting data races. At a high level, Pacer partitions the observed execution trace into *sampling* and *non-sampling* phases, similar to **RPT**. In each sampling period, Pacer monitors all events, by performing metadata updates as in FastTrack, similar to **RPT**. But unlike **RPT**, the expected total size of the sampling periods is $r \cdot n$ where r is a sampling rate set by Pacer, and n is the total size of the execution trace. The more stark difference is that Pacer also performs metadata updates on (a subset of) events in a non-sampling period —- all synchronization operations, as well as all memory locations that were accessed in prior sampling periods are tracked. This means that, in general, Pacer effectively can analyze a large number of events, much more than what is determined by its proportionality constant r. On the other hand, our approach guarantees that the number of events analyzed by **RPT** over the course of the entire execution is bounded by a constant which is determined by the number of threads, lock nesting depth and the chosen values for the parameters ϵ and δ.

The publicly available implementation of Pacer [Bond 2021] is built on top of Jikes RVM-3.1.0, which is only compatible with an old version of Java 1.6.0. Further, this implementation does not support comparing the same execution trace against different runtime techniques. A distinguishing feature of the Jikes-RVM implementation of Pacer's algorithm is the use of the runtime garbage collector to implement a periodic random sampler. In our implementation, we simulate the effect of a periodic sampler by invoking our sampler in a periodic fashion. Our implementation of Pacer in RAPID closely mimics the algorithm's description in the original paper [Bond et al. 2010], including the use of version clocks, shallow copies and optimizations in vector clock joins.

Benchmarks. Our benchmark programs are primarily derived from prior works which evaluate the performance of different race detection techniques [Flanagan and Freund 2009; Huang et al. 2014; Kini et al. 2017; Mathur et al. 2022, 2021]. These include Java benchmarks from the DaCaPo benchmark suite [Blackburn et al. 2006], Java Grande Forum [Smith et al. 2001], microbenchmarks from [von Praun and Gross 2003] and SIR [Do et al. 2005], and OpenMP benchmarks derived from DataAccelerator [Schmitz et al. 2020], DataRaceBench [Liao et al. 2017], OpenMP source code repository [Dorta et al. 2005] and the NAS parallel benchmarks [Bailey et al. 1991] as well as from HPC applications including CORAL benchmarks [Advanced Simulation and Computing, LLNL 2022a,b], ECP proxy applications [LLNL 2022], and Mantevo project [Sandia National Laboratories 2022].

Experimental Setup. The execution traces of Java benchmarks were logged using the RoadRunner [Flanagan and Freund 2010] dynamic analysis framework and traces from OpenMP benchmarks

Table 1. Characteristics of traces. We aggregate benchmark traces in 6 clusters, based on the values of parameter $m = 4|T| + 2h$. Column 1 shows the range of m values in each cluster and Column 2 shows the number of traces in each such cluster. Columns 3-9 show the average, min, max and different percentiles of the lengths of the traces in each cluster. The last row shows these metrics for the entire dataset.

1	2	3	4	5	6	7	8	9
m	Num. of	Trace Length Distribution						
(range)	traces	Average	Min	20 %-ile	40 %-ile	60 %-ile	80 %-ile	Max
(0, 29]	12	650.2M	40.0M	134.3M	291.7M	539.6M	607.8M	2.8B
(29, 59]	10	391.3M	1.0M	47.0M	124.6M	253.4M	323.0M	2.4B
(59, 69]	57	165.9M	3.1M	104.9M	112.3M	135.0M	168.9M	1.3B
(69, 95]	14	541.1M	11.7M	90.2M	265.4M	533.9M	771.1M	1.6B
(95, 231]	49	294.2M	11.7M	106.8M	132.9M	175.1M	360.0M	2.1B
(231, 1000]	7	158.7M	39.1M	65.2M	177.8M	199.9M	207.5M	259.1M
All traces	149	297M	1.0M	102M	127M	172M	349M	2.8B

were logged using ThreadSanitizer [Serebryany and Iskhodzhanov 2009]. There were 90 programs in the benchmark suite and we ran some programs on two different inputs. This resulted in a total of 149 benchmark traces, after filtering out short traces (with $< 1M$ events). We analyzed each of these traces against the three algorithms we have implemented. We use the parameter combination ($\epsilon = 0.01, \delta = 0.1$) for **RPT** and set Pacer's sampling rate to be 3% as suggested in [Bond et al. 2010]. On each trace and for each combination of parameters, we ran FastTrack 20 times, while **RPT** and Pacer 50 times, to account for randomization and variations in system load; FastTrack was run fewer times because it is a deterministic algorithm. We report the average performance of each tool in our tables and graphs. Our experiments were conducted on machine with 2.6GHz 64-bit Linux machine, using Java-1.8 as the JVM and 30GB heap space.

4.2 Trace Characteristics

Since the number of traces in our collection is large, we will summarize the data by clustering traces according to the values of several parameters. We choose two metrics for clustering traces — lengths of the traces (or number of events), and the value of $m = 4|T| + 2h$ which governs the length of each sample **RPT** extracts, and thus the total number of sampled events. The choice of the second metric allows us to focus on intrinsically *similar* traces at the same time. Table 1 summarizes our set of traces, in overall terms, as well as with the details of each m-based cluster. Observe that the total number of events go as high as 2.8 billion, the average trace length is 297 million and the median trace length is around 135 million. The trace lengths are diverse overall, as well as within each cluster. In total there are 5 benchmarks that are race free. Categorized by trace length, there are 30 benchmarks in $(0, 100M]$, 70 in $(100M, 200M]$, 24 in $(200M, 400M]$, 14 in $(400M, 700M]$ and 11 in $(700M, 3B]$ [4]. We also provide a table with more detailed information for the benchmarks in the appendix.

4.3 Detecting at Least One Race

RPT is designed to approximately solve the decision problem of race detection, i.e., answer the question whether an observed trace has a race. The innovation in the algorithm is to identify how little sampling will still allow one to mathematically guarantee reporting a race (with high

[4] $100M$ denotes 100 million or 10^8, while $1B$ denotes 1 billion or 10^9.

probability) in a trace that is far from being race-free. In this section, we explore how effective the theoretical claims are in practice by answering the following questions.

- **RPT** is designed to sample minimally so that its running time is low. Does this hold in practice? We answer this in the affirmative.
- Theoretical claims about **RPT**'s correctness guarantee detecting a race only when the trace is ϵ-far from being race-free. In practice, how does **RPT** perform when the distance of the observed trace from race-free traces is much less than ϵ? Does **RPT** successfully report races in such traces? We find that **RPT** does report races often even when the observed trace is very close to being race free.
- **RPT** samples short sub-traces and checks for races within these sub-traces. Thus, a race pair that is far apart will not be detected by **RPT** since the sub-traces sampled will not have both the events forming the race pair. How often do traces contain race pairs that are not far apart? Second, is **RPT** able to report races, when almost all the races are at distance greater than the sample length used by **RPT**? In our benchmark suite, we observe that most traces have races between nearby events. Moreover, even in traces where almost all races are far apart, **RPT** successfully reports races often.

Running Time. Our implementation of each of FastTrack, Pacer and **RPT** analyzes trace logs, and we measure the running time of race detection by simply measuring the time taken by each of the algorithms to analyze each trace. The running times are computed as the average of the time taken during each run of an algorithm on a given trace. In order to be able to present our results for the large benchmarks visually, we cluster traces by their lengths. For each cluster, we compute the weighted average of the running times and speedups(FastTrack as baseline) for FastTrack, Pacer, and **RPT**, where the weight for a trace is the reciprocal of its length.

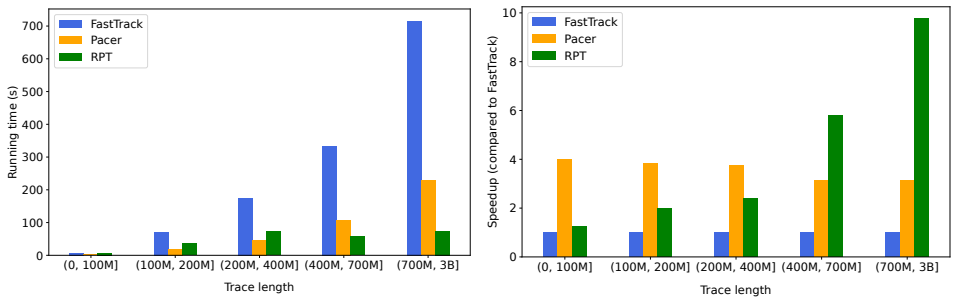
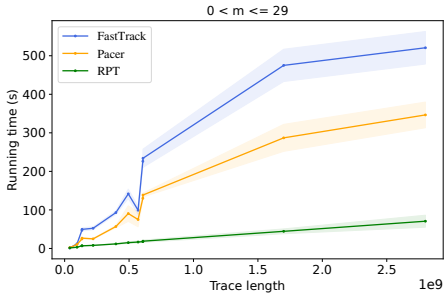


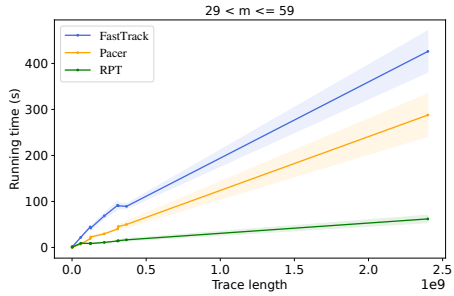
Fig. 1. Weighted average running time (left) as function of trace length (range) and weighted average speedup over FastTrack (right) as function of trace length(range)

Not surprisingly, the running time of **RPT** is pretty low, and in fact, the lowest for the larger traces. Further, Pacer is also significantly faster than FastTrack. As trace lengths increase, **RPT**'s competitive advantage over Pacer and FastTrack becomes more significant, and **RPT**'s running time does not grow as fast.

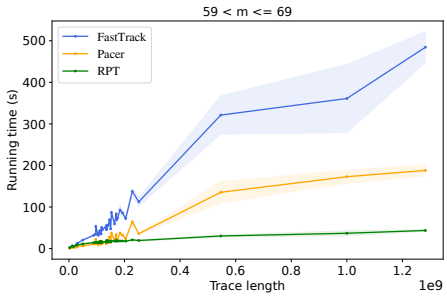
We next 'zoom-in' into the traces to understand the running times better, instead of aggregating the running times for several traces. Indeed, the number of traces in the extremal buckets (for example, for traces with length > 700M) averaging the running times smoothes out interesting behavior that we would like to otherwise understand. For a fine grained analysis, we cluster the traces according



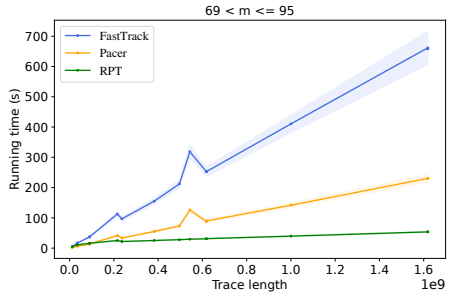
(a) Running times for traces with $m \in (0, 29]$



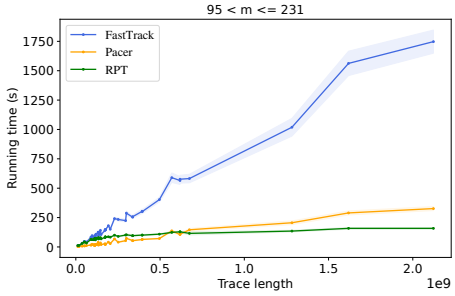
(b) Running times for traces with $m \in (29, 59]$



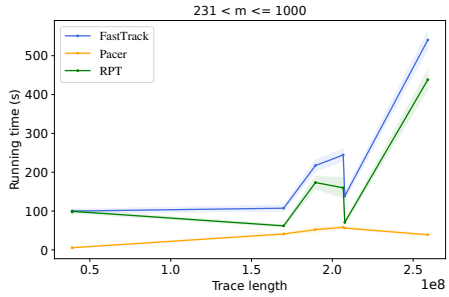
(c) Running times for traces with $m \in (59, 69]$



(d) Running times for traces with $m \in (69, 95]$



(e) Running times for traces with $m \in (95, 231]$



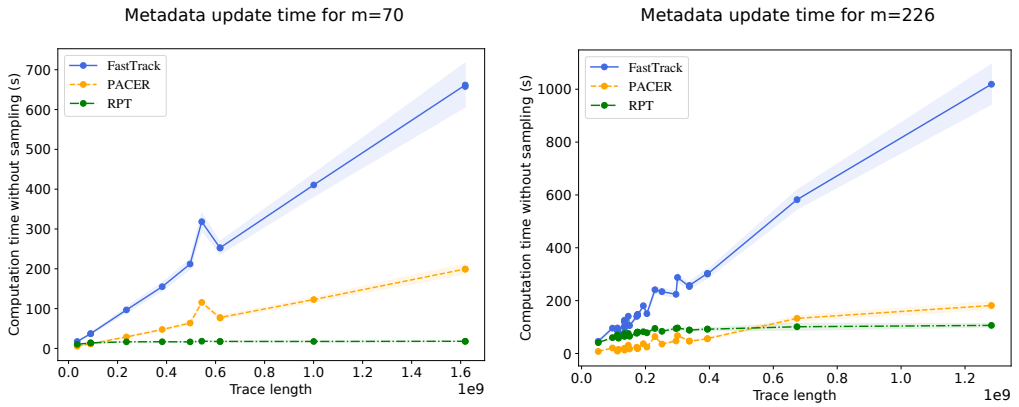
(f) Running times for traces with $m \in (231, 1000]$

Fig. 2. Running time as a function for trace length in each cluster.

to the parameter m and analyze each such cluster individually. Fig. 2 shows how the running time varies with trace lengths, in each cluster. The first observation we make is that the exact runtimes vary widely across clusters (even for similar trace lengths); see for example the clusters corresponding to $m \in (29, 59]$ and $m \in (95, 231]$ where the time taken varies significantly for traces of similar lenghts. However, inside a given cluster, the times increase, roughly linearly with the lengths of the traces. This justifies our choice of m as a measure for clustering traces. Indeed, the number of threads (and thus m) governs the size of the vector clocks and also the running time, and further, also governs the the number of events sampled by **RPT**. Finally, the time taken by **RPT**, in each cluster, is much lower than Pacer, which is much lower than the deterministic algorithm of FastTrack where every event in the trace is analyzed for detecting data races.

Observe that, even though our theoretical analysis of **RPT** (Section 3) guarantees *constant* running time, the trend for **RPT** in any of the figures in Fig. 2 does not completely 'flatten' out. This is due to the cost introduced by the random number generator and in detecting when sampling is switched on or off. In the next subsection, we investigate the running times in further detail to highlight this. Overall, **RPT** introduces much lower analysis cost than PACER and FASTTRACK.

Constant Running Time. Recall that the number of events sampled by **RPT** is $\widetilde{O}(m)$ and is



(a) Traces with $m = 70$ (b) Traces with $m = 226$

Fig. 3. Time to update metadata as a function of trace length, when the parameter m is constant.

independent of the length of the trace. A natural question to ask is if that is reflected in **RPT**'s running time experimentally. We study two collections of traces from our benchmarks to better understand this. The first set consists of all traces (12 in total) with $m = 70$, and the second consists of all traces (28 in total) with $m = 226$. Trace lengths vary in each collection to help understand overheads of each algorithm with increasing trace length. For both these sets, we plot the overhead due to processing meta-data for FASTTRACK, PACER, and **RPT** for the corresponding set of benchmarks in Fig. 3. This excludes the time taken by the sampling-based algorithms in generating the random numbers. Since FASTTRACK performs meta-data operations on all events, we report the total time taken for it. For **RPT** and PACER, we only report the time for processing meta-data on the chosen events. We see that as expected, **RPT** spends constant amount of time for analyzing the sampled part of the trace. On the other hand, both PACER and FASTTRACK spend time that increases with trace length.

Precision. We want to evaluate how **RPT** performs in terms of its ability to expose data races. Since **RPT** samples only a constant number of events, it is bound to not report every dynamic *warning*, i.e., those events that appear as the second event e_2 in a data race *event* pair (e_1, e_2). At the same time, the (ϵ, δ)-guarantee of **RPT** ensures that whenever the observed execution is sufficiently 'racy', it will report a race with a high probability. In the following, we report on the precision of **RPT** and compare it with the precision of PACER.

Fig. 4 shows the probability with which PACER and **RPT** detect at least one race for each benchmark; we call this the success rate of each benchmark:

$$\text{success} = \frac{\#\text{ runs with} \geq 1 \text{ warnings}}{\text{Total number of runs}}$$

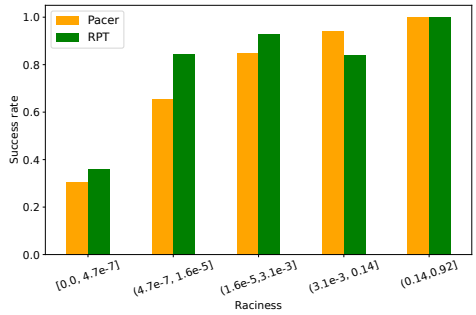


Fig. 4. Success rate as a function of raciness (average)

Recall that since **RPT** is a property tester, it guarantees to detect a race with high probability only when the trace being analyzed is far from race-free traces with respect to hamming distance. This is difficult to measure for a trace. We computed an approximation to the hamming distance that we call the raciness of a trace:

$$\text{raciness} = \frac{\text{avg. \# warnings reported by FastTrack}}{\text{Trace length}}$$

Raciness of a trace σ is an upper bound on the hamming distance of σ from any race-free trace. In other words, if a trace has low raciness, then it is very close to being race-free with respect to the hamming distance. However, it could be a poor overestimate. We expect that the success rate of **RPT** will increase as the raciness of the trace increases. In Fig. 4, we cluster traces based on their raciness, and aggregate the success rates for each bucket, and then evaluate both **RPT** and Pacer based on their success rates. Observe that over most of the clusters, **RPT**'s probability of detecting a race is similar, if not better, than Pacer's probability of race detection. Overall, we conclude that **RPT** successfully flags an execution racy with very high probability, even if the number of warnings in the trace is small (about 10^{-7} times the number of events in the trace). Recall that in our experiments, we run **RPT** with $\epsilon = 0.01$ and $10^{-7} \ll 0.01 = \epsilon$.

Distance between racy pairs. RPT samples short sub-traces, and can only report races between

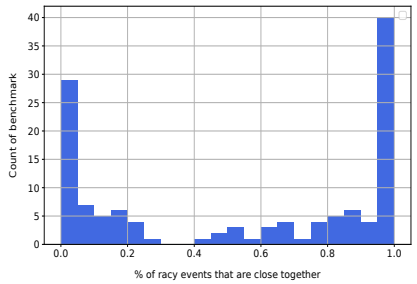


Fig. 5. Distribution of number of benchmarks over ratios of racy second events over all racy second events.

pairs of events that both belong to a sampled sub-trace. Thus, **RPT** cannot report races between pairs of events that are far apart. How often do traces have races separated by a small number of events? Let us say that a race pair (e_1, e_2) in trace σ is "short", if the distance between e_1 and e_2 is less than the length of the sub-traces sampled by RPT. We cluster our benchmark traces based

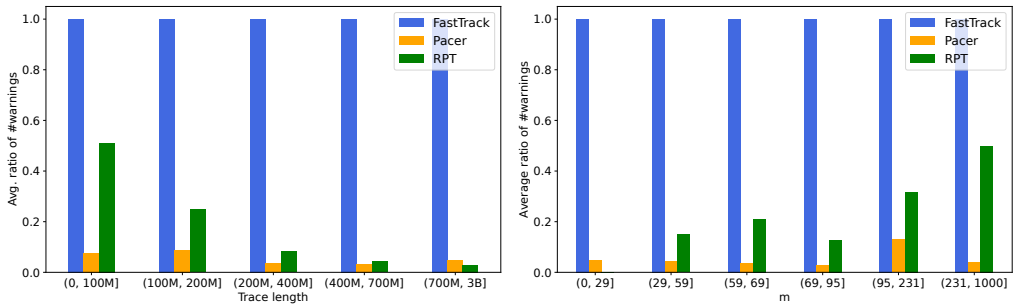
on the number of short races as a fraction of the total number of races in the trace, and this is plotted as a histogram in Fig. 5; when plotting this histogram, we drop the one race-free trace in our suite. We observe that in 52 traces the percentage of short races is $< 25\%$, while in 96 traces (approximately 2/3rds of our suite), the percentage of short races is at least 40%. Thus, short races are quite common in practice.

Next, we study how **RPT** does on traces where the number of short races is very small, i.e., $< 5\%$. There are 29 such traces (about 1/5th of our suite). Surprisingly **RPT**'s success rate is very good even on this set. On average **RPT** reports at least one race 84% of the time on these traces. On 21 (out of 29) traces, **RPT** reports a race 100% of the time, and it reports a race at least half the time on 25 of these traces. In some of these examples, the percentage of short races is less than 0.01%. Among the remaining 4 examples, there was one trace where **RPT** never reported a race, and on the remaining 3 examples whose percentage of short races is $0.003 - 0.02\%$, **RPT** reported a race $20 - 25\%$ of the time. In particular, the one example on which **RPT** never reports a race, there are no short races and Pacer also fails to report any race in any of its runs on this trace.

4.4 Detecting Various Races

FastTrack reports every race pair and Pacer guarantees to report every race pair with an equal and non-zero probability. **RPT**, in contrast, does not provide such strong guarantees with it when comes to reporting an arbitrary race pair. It only promises to report some race on traces that are far from race-free traces. Nonetheless, we would like to understand how many races **RPT** reports and whether there are races that escape detection with **RPT**. We report the results of investigation in this section.

Average ratios of warnings per run. In Fig. 6, we depict how the number of warnings reported



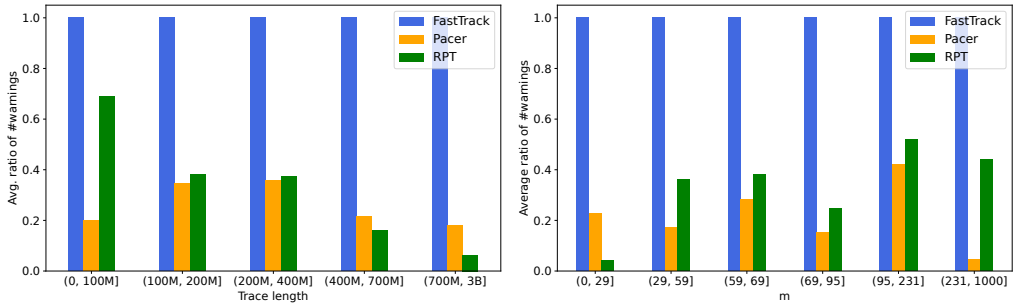
(a) Warning ratios v/s trace length

(b) Warning ratios v/s m

Fig. 6. Ratio of warnings (with warnings of FastTrack) as a function of trace length and m.

varies in our suite of traces. As before, to visualize the data, we cluster it as per trace length and m. For each cluster, we consider the ratio of warnings reported by an algorithn (**RPT**, Pacer or FastTrack) and the number of warnings in the trace (namely those reported by the deterministic algorithm FastTrack). For each cluster, we compute the average of these ratios. Fig. 6a, we report how the ratio varies across clusters of different trace lengths. For smaller traces, **RPT** reports a large fraction of the warnings, as compared to Pacer. This is expected, because **RPT** samples constantly many events, which for the case of smaller traces, amounts to sampling a large fraction of the trace. As a result, it reports many warnings. On the other hand, Pacer misses races for smaller traces due to its proportional sampling. For the large traces, Pacer is able to find more races as expected

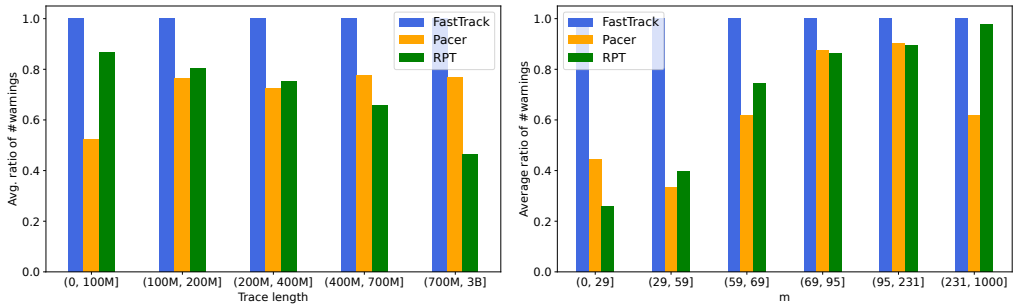
because of its proportional nature. Fig. 6b plots these ratios when the traces are clustered by the value of m. The reason to study these plots clustered by m is because the number of samples drawn by **RPT** on a trace grows as a function of m; as shown in Table 1, clustering by m and trace length are different ways to slice up our examples, and there is no correlation between these measures.



(a) Racy memory locations v/s trace length (b) Racy memory locations reported v/s m

Fig. 7. Number of memory locations (normalized by those reported by FastTrack) as a function of trace length and m.

Exposing racy memory locations. Our benchmarks exhibit HB-races on enormous memory locations. Here we evaluate the following question — can **RPT** detect each racy memory location? Or, there are a large number of variables with data races that are inherently difficult for **RPT** to discover? Admittedly, reports on unique memory locations are more insightful for developers using a race detector, as excessive number of repeated warnings (on the same location) are known to easily overwhelm developers. As before, we compute the aggregated ratios of memory locations that **RPT**, Pacer and FastTrack report (as compared with those reported by FastTrack), where the aggregation is performed according to trace lengths and m. This is shown in Fig. 7. While the trends here are similar to those for number of warnings, these figures show that, in fact, when we focus on the number of unique memory locations flagged as racy, **RPT** is able to correctly flag a good ratio of memory locations to be racy.



(a) Number of racy source code lines v/s trace length (b) Number of racy source code lines v/s m

Fig. 8. Number of source code locations (normalized by those reported by FastTrack) as a function of trace length and m.

Exposing racy source code locations. We next focus on the source code locations that these race detection algorithms report. From the standpoint of a software developer using a race detector,

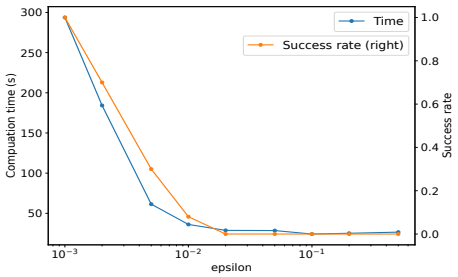


Fig. 9. The running time and success rate of **RPT** on xalan as ϵ varies.

this metric is even more valuable since developers are interested in localizing the data races and thereafter fix them. We report the number of source code locations flagged to be racy by each of the three tools, and present them, as before, by clustering according to trace lengths and m, in Fig. 8. We observe that both Pacer and **RPT** report reasonably many source code locations. This shows the power of sampling based approaches. Notably, for traces with higher m (that is, higher number of threads), **RPT** can report almost all locations in the source code that are flagged to be racy by the baseline determinsitic algorithm FastTrack.

4.5 Choosing ϵ

The performance of our property testing algorithm **RPT**, both in terms of its runtime and its ability to detect races, depends on parameters ϵ and δ. Changes to δ do not significantly affect the number of sampled events (which vary as $\ln(\frac{1}{\delta})$). Therefore, we keep δ fixed at 0.1 and see how things change as we vary ϵ. We ran **RPT** on one of the DaCaPo benchmark xalan, with values of $\epsilon \in \{0.5, 0.2, 0.1, 0.05, 0.02, 0.01, 0.005, 0.002, 0.001\}$. The choice of xalan was determined by a desire to pick a long trace with reasonable raciness. In Fig. 9, we plot how running time of **RPT** changes with ϵ and how the probability of detecting a race changes with ϵ (right axis). As expected, both the running time and the probability of detecting a race increase as ϵ is decreased.

5 RELATED WORK

Data races form the most common [Lu et al. 2008] as well difficult to detect [Musuvathi et al. 2008] class of concurrency bugs. Extensive research on developing techniques for automatically detecting data races has led to the development of many influential static and dynamic analysis techniques for race detection. Static techniques such as [Abadi et al. 2006; Blackshear et al. 2018; Flanagan and Freund 2000; Naik et al. 2006; Voung et al. 2007] employ type based analysis or interprocedural data and control flow analysis to infer if two accesses on a common memory location may not be protected by a common lock. Besides the inherent unsoundness in this criteria (memory accesses may still not be racy even when they are not protected by a common lock), computability limits imply that static techniques either report many false races, do not scale or require extensive manual annotation in the software to be analyzed.

Dynamic analysis techniques, on the other hand, do not detect races in the entire source code, but instead limit their attention to single executions, and are typically *sound* and completely automated. Popular techniques include Eraser-style lockset analysis [Savage et al. 1997], happens-before (HB) [Lamport 1978] based race detection [Elmas et al. 2007; Itzkovitz et al. 1999; Pozniansky and Schuster 2003] or hybrid techniques [O'Callahan and Choi 2003]. Amongst these, only HB-based techniques are sound (upto the first race reported [Mathur et al. 2018]), and are implemented in

popular tools like ThreadSanitizer [Serebryany and Iskhodzhanov 2009; Serebryany et al. 2011], Helgrind [Müehlenfeld and Wotawa 2007] or Intel Inspector [int 2021] using vector clocks. Even though such techniques are fully automated, their use is typically limited to in-house testing to avoid the large overhead of tracking metadata required for analysis.

Reducing runtime overhead. Our work aims to to improve the performance of HB-race detectors. The most prominent work with similar goal is the FastTrack algorithm that reduces the overhead of race checking, simplifying it using the *epoch* optimization. In our evaluation, we compare against this algorithm and show that sampling approaches like ours and Pacer's can indeed complement its performance, without compromising soundness. The overhead due to vector clocks updates can also be systematically reduced for programs whose execution graphs exhibit special structures [Agrawal et al. 2018; Cheng et al. 1998; Dimitrov et al. 2015; Feng and Leiserson 1997; Raman et al. 2012; Surendran and Sarkar 2016]. Other works with similar goals include the use of hardware support [Devietti et al. 2012], optimizing metadata synchronization [Bond et al. 2013; Wood et al. 2017], or static analysis for optimizing race check placement [Flanagan and Freund 2013; Rhodes et al. 2017].

Sampling Based Dynamic Analysis. The Pacer algorithm due to Bond, Coons and McKinley [Bond et al. 2010] is closest in spirit to our approach, although has significant differences. First the Pacer algorithm has stronger guarantees. The algorithm has a sampling parameter ρ and it guarantees that every race in the input trace will be detected with probability at least ρ. A property tester, as presented in this paper, however has weaker guarantees. It promises to detect races in traces that are far, in terms of hamming distance, from race-free executions with high probability. However, in order obtain the stronger guarantees, Pacer needs to sample significantly more events than the algorithm presented here. The expected number of samples drawn by Pacer is ρn (where input trace σ has length n), and it can be easily shown that there are executions on which with non-zero probability Pacer will sample *all* events in the trace. In contrast, the number of events sampled by our algorithm is at most $\widetilde{O}(t + h)$, which is independent of the length of the input trace σ. This means that our algorithm will typically have lower overheads than Pacer, at the cost of finding fewer races. This is borne out by our experiments. Other sampling based approaches that are closely related include LiteRace [Marino et al. 2009] also resorts to sampling for reducing runtime overhead guided by the 'cold-region hypothesis' that data races are more likely exhibited in parts of the source code that are not exercised frequently during the execution. DataCollider [Erickson et al. 2010] samples memory locations to detect data races on, in order to reduce runtime overhead of dynamic race detection. However, most of these algorithms provide no formal mathematical guarantees for the likelihood of reporting races or any upper bound on the number of events sampled.

Sampling Based Hybrid Analysis. The work in RaceMob [Kasikci et al. 2013] deploys a two-phase hybrid analysis. The first static analysis phase identifies a more precise set of memory locations that may potentially be racy, and in the second pass, performs sampled dynamic analysis while focusing on the memory locations identified from during the first static analysis phase. This is in line with the work of Choi et al [Choi et al. 2002], and is also more recently deployed for kernel race bugs [Jeong et al. 2019]. We remark that the fundamental algorithmic improvements proposed in our paper are orthogonal to such analyses and can be modularly plugged into such hybrid race detection techniques to reduce the overhead of the dynamic analysis phase. Similar to RaceMob's analysis, **RPT** can also benefit from an additional static analysis phase which can identify a focused set of memory locations. Such a set can help reduce space usage due to redundant metadata and the performance cost due to updates on such metadata. Further, if a candidate set of memory locations

is known prior to the dynamic analysis, the core algorithm of **RPT** can also be adopted so that it only samples sub-traces that begin at access events of memory locations belonging to this candidate set. In our paper, we focus on experimental comparison with purely dynamic analysis techniques to precisely distill the algorithmic benefits our approach offers.

Randomized scheduling. Another popular set of race detection techniques that employ randomization include those that drive the thread schedulers using randomization [Burckhardt et al. 2010; Luo and Demsky 2021; Sen 2008; Xu et al. 2020], with the goal of enhancing the likelihood of covering a racy location. Such techniques fall into the general class of controlled concurrency testing techniques [Thomson et al. 2016] that employ heuristics such as bounding context switches [Kahlon et al. 2005; Musuvathi and Qadeer 2007; Sorrentino et al. 2010], thread speed control [Chen et al. 2018] or reinforcement learning to determine a buggy thread schedule [Mukherjee et al. 2020]. Our approach is orthogonal to all such approaches and can potentially reduce the overhead of intensive exploration.

Other Approaches. Dynamic race detection has been a topic of interest, and recent advances such as *predictive* analysis aim to enhance coverage of HB-based race detectors by considering alternate reorderings that exhibit a race, without explicitly re-executing the program, and have been shown to be effective in catching hidden races. These include explicit enumeration based techniques [Sen et al. 2005], symbolic techniques [Huang et al. 2014; Said et al. 2011], graph based analyses [Pavlogiannis 2019] and partial order based techniques [Kini et al. 2017; Mathur et al. 2021; Roemer et al. 2018, 2020; Smaragdakis et al. 2012].

Property Testing. Property testing [Goldreich 2017] is a widely studied sub-field in theoretical computer science, where the relaxation to solve a decision problem is exploited to design sublinear algorithms for a variety of problems. Our algorithm and its proof, draws heavily from the ideas presented in [Alon et al. 2001], where it is shown that regular languages can be property tested with a constant number of samples. The proof we present here specializes the ideas in [Alon et al. 2001] for race detection. In particular, we have paid particular attention to the constants involved in our analysis, because even small factor changes can impact the performance of our algorithm on benchmarks we experiment with.

6 CONCLUSIONS AND FUTURE WORK

We presented a randomized property tester (**RPT**) for detecting HB-races. The algorithm is sound, i.e., never reports a race when there is none, and guarantees to detect a race if the input trace is far from any race-free trace. Moreover, the algorithm samples and processes only constantly many events even in the worst case. This is in sharp contrast to previously proposed sampling based approaches for race detection. **RPT** was implemented and compared against two well known HB race detectors: FastTrack and Pacer. Experimental evaluation showed that **RPT** did indeed have the lowest running time among all the algorithms, and detects races often.

There are several interesting avenues for future work. Our work demonstrates the effectiveness of property testing based algorithms for detecting data races. We think that this paradigm will also be promising for improving the effectiveness of detecting other errors such as deadlocks [Havelund 2000] and atomicity violations [Farzan et al. 2009; Flanagan et al. 2008; Mathur and Viswanathan 2020] for which dynamic analysis is often the preferred method of detection. Second, it would be interesting to design property testing algorithms for more predictive notions of data races including polynomial time algorithms [Kini et al. 2017; Mathur et al. 2021; Smaragdakis et al. 2012] as well as more exhaustive ones [Huang et al. 2014; Mathur et al. 2020; Said et al. 2011]. Finally, we envision that developing a sampling algorithm (like **RPT** or Pacer) that performs sampling in a distributed

manner, and introduces minimal additional synchronizations on events will help further lower the overhead introduced due to sampling, but is expected to require new algorithmic insights.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive feedback on an earlier drafts of this manuscript. Umang Mathur was partially supported by the Simons Institute for the Theory of Computing, and by a Singapore Ministry of Education (MoE) Academic Research Fund (AcRF) Tier 1 grant. Minjian Zhang and Mahesh Viswanathan are partially supported by NSF SHF 1901069 and NSF CCF 2007428.

A APPENDIX

Trace characteristics, sampling statistics and races reported. $\delta = 0.1$ for **RPT**. PACER uses sampling ratio=3%.

1	2	3	4	5	6	7	8	9	10	11
Benchmark	trace length	M	PACER sampled	**RPT** sampled	# Warnings			# Warning variables		
					FT	PACER	**RPT**	FT	PACER	**RPT**
zero-reversal-logs-final-logs	1.0M	56.0	358.9K	499.0K	29.6K	3.5K	18.4K	988.00	163.53	634.67
zero-reversal-logs-final-logs1	1.4M	32.0	136.5K	814.5K	72.00	4.58	39.63	24.00	1.22	12.83
zero-reversal-logs-final-logs2	3.1M	60.0	747.3K	2.6M	14.00	0.29	8.80	13.00	0.27	8.20
zero-reversal-logs-final-logs3	11.7M	72.0	1.5M	11.1M	252.00	7.13	123.98	28.00	1.02	17.05
HPCBench-NPBS-DC.S-12M-events	11.7M	228.0	2.9M	11.3M	5.8K	119.09	5.4K	574.00	17.62	534.35
HPCBench-NPBS-DC.S-12M-events1	11.7M	68.0	3.5M	11.1M	82.00	2.84	18.17	82.00	2.84	18.17
sunflow	16.8M	68.0	1.3M	15.1M	252.00	2.35	109.96	28.00	0.24	15.90
misc-hsqldb-hsqldb	18.8M	184.0	3.0M	18.1M	284.00	5.40	258.87	5.00	0.75	4.55
DRACC-DRACC-OMP-017-Counter-wr	27.0M	68.0	11.6M	21.8M	15.00	1.09	13.12	15.00	1.09	13.12
OmpSCR-v2.0-c-testPath-30M-eve	30.2M	68.0	18.0M	23.4M	181.00	12.89	26.37	16.00	0.89	11.73
OMPRacer-Lulesh-35M-events-16	35.3M	70.0	8.0M	26.0M	8.8M	253.2K	1.9M	65.4K	36.5K	51.4K
OmpSCR-v2.0-c-testPath-37M-eve	37.5M	228.0	19.8M	35.8M	289.00	8.80	207.83	57.00	2.05	52.48
misc-tradesoap-tradesoap	39.1M	904.0	4.8M	38.8M	7.4K	247.87	7.2K	396.00	9.60	387.88
misc-tradebeans-tradebeans	39.1M	908.0	4.6M	39.0M	7.2K	22.09	7.1K	401.00	7.93	399.45
series	40.0M	18.0	22.3K	10.3M	0.00	0.00	0.00	0.00	0.00	0.00
DataRaceBench-DRB155-missingor	50.0M	68.0	20.6M	29.7M	17.00	0.04	7.60	17.00	0.04	7.60
DataRaceBench-DRB155-missingor1	50.0M	228.0	27.3M	46.5M	58.00	55.11	38.85	58.00	55.11	38.85
OMPRacer-Lulesh-52M-events-56	52.1M	226.0	11.0M	49.4M	12.1M	379.2K	6.8M	86.4K	48.6K	86.3K
zero-reversal-logs-final-logs4	58.5M	42.0	8.8M	22.8M	93.00	1.71	0.98	4.00	0.20	0.32
tomcat	63.2M	212.0	4.8M	56.6M	1.2M	35.4K	498.9K	17.8K	9.2K	16.4K
OmpSCR-v2.0-cpp-sortOpenMP-cpp	88.9M	66.0	35.9M	35.0M	32.1M	986.0K	240.7K	8.0K	5.4K	7.9K
DRB177-fib-taskdep-yes-90M-eve	90.2M	70.0	44.3M	37.2M	3.8K	112.52	1.2K	1.5K	102.50	692.37
DataRaceBench-DRB176-fib-taskd	90.2M	70.0	44.3M	37.0M	9.9K	334.80	3.3K	2.3K	256.53	1.4K
DRB177-fib-taskdep-yes-90M-eve1	90.3M	230.0	44.2M	74.8M	7.8K	452.84	4.8K	3.9K	425.11	2.7K
DataRaceBench-DRB176-fib-taskd1	90.3M	230.0	44.3M	74.6M	26.6K	934.31	19.5K	7.2K	783.18	6.3K
fop	96.0M	8.0	9.9M	5.4M	0.00	0.00	0.00	0.00	0.00	0.00
OmpSCR-v2.0-c-LoopsWithDepende	96.4M	66.0	6.7M	36.1M	1.7K	113.79	551.70	31.00	14.43	21.02
OmpSCR-v2.0-c-LoopsWithDepende1	96.4M	66.0	6.8M	35.9M	2.9K	97.56	569.07	31.00	14.55	20.35
OMPRacer-XSBench-97M-events-16	96.6M	66.0	23.0M	36.4M	27.00	5.09	7.45	27.00	5.09	7.45
OMPRacer-XSBench-97M-events-56	96.6M	226.0	21.4M	77.5M	89.00	6.69	61.02	89.00	6.69	61.02
DRACC-DRACC-OMP-014-Counter-wr	104.9M	68.0	54.2M	37.5M	15.00	0.27	6.87	15.00	0.27	6.87
DRACC-DRACC-OMP-020-Counter-wr	104.9M	68.0	52.9M	37.9M	15.00	0.27	5.28	15.00	0.27	5.28
DRACC-DRACC-OMP-019-Counter-wr	104.9M	68.0	3.9M	37.3M	94.6M	2.8M	33.2M	527.00	512.27	515.25
DRACC-DRACC-OMP-018-Counter-wr	104.9M	68.0	3.9M	37.8M	93.9M	2.8M	33.3M	527.00	512.55	518.25
DRACC-DRACC-OMP-013-Counter-wr	104.9M	68.0	3.9M	36.6M	96.0M	2.9M	33.0M	527.00	512.00	517.40
DRACC-DRACC-OMP-012-Counter-wr	104.9M	68.0	3.9M	37.8M	95.6M	2.8M	33.9M	527.00	512.27	517.25
OmpSCR-v2.0-cpp-sortOpenMP-cpp1	106.7M	228.0	74.1M	82.3M	193.00	7.65	146.87	56.00	3.91	44.08
OmpSCR-v2.0-cpp-sortOpenMP-cpp2	106.8M	228.0	74.2M	82.3M	177.00	5.38	140.02	56.00	1.95	48.65
OmpSCR-v2.0-cpp-sortOpenMP-cpp3	107.0M	68.0	41.7M	38.0M	47.00	1.85	14.50	16.00	0.89	5.13
OmpSCR-v2.0-cpp-sortOpenMP-cpp4	107.1M	228.0	74.5M	82.4M	199.00	5.98	150.23	56.00	0.96	42.52
OmpSCR-v2.0-cpp-sortOpenMP-cpp5	107.5M	68.0	42.3M	38.0M	48.00	1.38	15.97	16.00	0.60	6.28
DataRaceBench-DRB154-missinglo	112.0M	68.0	56.4M	38.3M	15.00	0.27	5.78	15.00	0.27	5.78
DataRaceBench-DRB152-missinglo	112.0M	68.0	34.4M	37.8M	16.00	0.09	5.00	16.00	0.09	5.00
DataRaceBench-DRB150-missinglo	112.0M	68.0	34.3M	38.3M	16.00	0.05	6.00	16.00	0.05	6.00
DataRaceBench-DRB122-taskundef	112.0M	66.0	61.6M	37.4M	15.00	0.82	4.00	15.00	0.82	4.00
DataRaceBench-DRB123-taskundef	112.0M	66.0	31.6M	37.4M	14.0M	409.2K	4.7M	227.00	212.00	217.75
DataRaceBench-DRB122-taskundef1	112.0M	226.0	61.2M	84.2M	55.00	55.00	24.02	55.00	55.00	24.02
DataRaceBench-DRB123-taskundef1	112.0M	226.0	36.6M	84.0M	13.9M	406.6K	10.4M	712.00	712.00	681.83
OMPRacer-Kripke-119M-events-56	119.2M	228.0	44.6M	87.3M	10.4M	999.4K	640.1K	116.3K	31.4K	49.6K
DataRaceBench-DRB110-ordered-o	120.0M	68.0	55.4M	38.7M	15.00	0.27	4.25	15.00	0.27	4.25

1	2	3	4	5	6	7	8	9	10	11
Benchmark	trace length	M	Pacer sampled	**RPT** sampled	# Warnings			# Warning variables		
					FT	Pacer	**RPT**	FT	Pacer	**RPT**
DataRaceBench-DRB110-ordered-o1	120.0M	228.0	68.9M	87.2M	55.00	55.00	25.90	55.00	55.00	25.90
zero-reversal-logs-final-logs5	122.5M	36.0	27.5M	22.1M	77.00	3.69	11.98	34.00	1.73	5.47
crypt	126.0M	30.0	105.8M	18.9M	0.00	0.00	0.00	0.00	0.00	0.00
OMPRacer-QuickSilver-133M-even	132.6M	228.0	47.1M	90.6M	1.1M	33.0K	344.2K	121.3K	9.1K	11.3K
DataRaceBench-DRB105-taskwait	134.0M	66.0	45.2M	38.6M	6.6K	209.44	1.6K	1.3K	172.93	719.35
DataRaceBench-DRB106-taskwaitm	134.0M	66.0	45.5M	38.1M	1.3K	36.47	176.00	877.00	35.82	160.47
DataRaceBench-DRB105-taskwait1	134.0M	226.0	41.4M	91.5M	40.9K	1.5K	23.8K	4.6K	1.1K	4.2K
DataRaceBench-DRB106-taskwaitm1	134.0M	226.0	43.5M	90.7M	3.3K	278.91	1.3K	2.5K	275.85	1.1K
zero-reversal-logs-final-logs6	134.1M	22.0	14.1M	14.3M	33.1M	595.9K	87.9K	185.8K	22.3K	14.6K
OmpSCR-v2.0-c-QuickSort-134M-e	134.3M	66.0	19.3M	38.6M	419.9K	19.8K	3.50	34.9K	6.1K	3.50
OmpSCR-v2.0-c-QuickSort-134M-e1	134.3M	226.0	19.8M	92.3M	419.9K	20.7K	33.92	35.0K	6.6K	33.92
lufact	135.0M	18.0	13.6M	11.9M	33.1M	515.8K	55.7K	185.8K	21.4K	11.2K
DRACC-DRACC-OMP-017-Counter-w1	135.0M	68.0	58.4M	39.7M	15.00	0.00	5.50	15.00	0.00	5.50
DRACC-DRACC-OMP-015-Counter-wr	135.0M	68.0	35.9M	39.6M	5.1M	154.2K	1.5M	16.00	1.27	6.25
DataRaceBench-DRB148-critical1	135.0M	68.0	36.1M	39.3M	5.4M	161.3K	1.6M	16.00	2.36	5.75
DRACC-DRACC-OMP-010-Counter-wr	135.0M	68.0	35.9M	39.5M	5.2M	155.1K	1.5M	16.00	1.82	5.58
DRACC-DRACC-OMP-009-Counter-wr	135.0M	68.0	36.1M	38.7M	5.3M	160.9K	1.5M	16.00	2.09	4.50
DRACC-DRACC-OMP-016-Counter-wr	135.0M	68.0	35.9M	39.6M	5.4M	161.7K	1.6M	16.00	1.55	5.43
OmpSCR-v2.0-c-LUreduction-136M	136.4M	66.0	10.7M	38.7M	42.2M	1.2M	175.1K	89.4K	71.9K	1.8K
OmpSCR-v2.0-c-LUreduction-137M	136.9M	226.0	86.3M	93.1M	37.2M	1.2M	2.7M	89.1K	64.3K	22.9K
DataRaceBench-DRB144-critical	140.0M	68.0	43.0M	39.9M	16.00	0.87	4.52	16.00	0.87	4.52
OmpSCR-v2.0-cpp-sortOpenMP-cpp	141.7M	68.0	96.6M	39.6M	16.00	0.55	2.50	16.00	0.55	2.50
HPCBench-OmpSCR-v2.0-c-fft6-14	146.0M	66.0	56.0M	39.0M	30.00	0.69	5.25	30.00	0.69	5.25
OmpSCR-v2.0-c-fft6-146M-events	146.0M	226.0	56.1M	94.5M	74.00	7.75	40.17	74.00	7.75	40.17
OmpSCR-v2.0-c-Pi-150M-events-1	150.0M	66.0	103.0M	39.3M	27.00	7.89	4.85	27.00	7.89	4.85
OmpSCR-v2.0-c-Pi-150M-events-5	150.0M	226.0	149.6M	95.9M	91.00	84.45	25.13	91.00	84.45	25.13
HPCBench-NPBS-IS.W-153M-events	152.9M	66.0	37.9M	39.4M	51.4M	2.6M	12.4K	2.0M	553.2K	11.55
OmpSCR-v2.0-cpp-sortOpenMP-cpp1	164.0M	68.0	65.6M	40.8M	667.6K	19.8K	7.2K	99.0K	6.9K	1.3K
OMPRacer-amg2013-170M-events-1	169.9M	358.0	21.8M	130.3M	24.0M	703.6K	4.3M	595.2K	43.7K	68.8K
SimpleMOC-170M-events-16-threa	170.2M	68.0	43.1M	41.0M	25.6K	723.89	28.13	2.1K	55.38	26.27
HPCBench-graph500-171M-events	171.3M	66.0	49.2M	40.0M	86.0M	2.6M	340.3K	115.2K	18.3K	11.5K
HPCBench-graph500-172M-events	172.5M	226.0	49.8M	102.8M	83.9M	2.8M	4.0M	118.5K	18.0K	19.4K
CoMD-CoMD-omp-task-deps-174M-e	174.1M	66.0	24.1M	39.1M	124.3M	3.4M	94.0K	16.5K	15.8K	1.9K
CoMD-CoMD-openmp-174M-events-1	174.1M	66.0	24.7M	40.1M	124.3M	3.7M	95.9K	16.5K	15.9K	1.8K
CoMD-CoMD-openmp-175M-events-5	175.1M	226.0	30.4M	103.4M	128.5M	3.5M	3.5M	16.9K	15.5K	6.2K
CoMD-CoMD-omp-task-175M-events	175.1M	226.0	29.9M	103.4M	128.5M	3.1M	3.4M	16.9K	15.7K	6.2K
CoMD-CoMD-omp-task-174M-events	175.1M	226.0	30.5M	103.3M	128.5M	3.1M	3.3M	16.9K	15.6K	6.3K
CoMD-CoMD-omp-task-deps-175M-e	175.1M	226.0	30.5M	103.3M	128.5M	3.5M	3.3M	16.9K	15.6K	6.3K
DataRaceBench-DRB062-matrixvec	183.9M	66.0	90.3M	38.7M	33.9M	1.1M	3.5M	31.00	16.82	19.53
OMPRacer-amg2013-190M-events-5	189.6M	518.0	25.1M	159.6M	28.0M	928.1K	8.9M	718.4K	65.3K	155.0K
OmpSCR-v2.0-c-LoopsWithDepende	192.6M	66.0	25.0M	40.5M	2.8K	88.75	52.42	32.00	15.82	17.45
DataRaceBench-DRB062-matrixvec1	193.2M	226.0	99.5M	106.9M	36.0M	1.4M	17.5M	116.00	111.07	72.32
OmpSCR-v2.0-c-MolecularDynamic	204.3M	66.0	77.8M	40.8M	83.2M	2.5M	51.7K	1.6K	1.6K	814.13
OmpSCR-v2.0-c-MolecularDynamic1	204.4M	226.0	86.9M	109.1M	87.0M	2.7M	569.4K	1.7K	1.7K	1.7K
OMPRacer-miniFE-207M-events-58	206.7M	518.0	49.8M	165.7M	27.8M	1.3M	6.3M	992.2K	51.0K	22.9K
OMPRacer-miniFE-208M-events-18	207.7M	358.0	48.3M	144.2M	19.4M	816.5K	1.5M	968.5K	41.2K	18.0K
misc-graphchi-graphchi	215.8M	86.0	51.7M	52.0M	1.8M	42.2K	1.2K	318.5K	10.1K	83.08
zero-reversal-logs-final-logs	217.5M	38.0	95.8M	24.7M	750.6K	9.4K	33.07	177.00	5.18	4.67
misc-biojava-biojava	221.0M	22.0	59.8M	14.7M	2.00	0.00	0.00	2.00	0.00	0.00
HPCBench-HPCCG-228M-events-16	228.1M	66.0	24.0M	41.3M	30.8M	1.1M	310.2K	15.0K	14.8K	1.0K
HPCBench-HPCCG-230M-events-56	229.5M	226.0	35.5M	113.3M	41.8M	2.0M	3.6M	15.8K	15.7K	13.8K
DRB177-fib-taskdep-yes-382M-ev	236.2M	70.0	115.8M	43.7M	2.1K	60.58	274.45	1.2K	56.64	226.92
CoMD-CoMD-omp-taskloop-251M-ev	251.5M	66.0	52.7M	41.5M	981.7K	35.6K	224.43	30.8K	1.1K	4.80
CoMD-CoMD-omp-taskloop-251M-ev1	251.5M	226.0	53.5M	115.9M	798.4K	55.1K	2.5K	64.9K	2.1K	38.37
misc-cassandra-cassandra	259.1M	704.0	150.3M	218.8M	42.1K	3.7K	30.7K	9.4K	255.93	6.9K
OmpSCR-v2.0-cpp-sortOpenMP-cpp2	295.5M	226.0	167.5M	120.9M	144.5M	4.5M	1.3M	6.1K	5.9K	6.0K
HPCBench-NPBS-IS.W-300M-events	300.1M	226.0	60.5M	121.6M	117.2M	12.1M	81.3K	2.1M	597.6K	324.48
zero-reversal-logs-final-logs1	307.3M	42.0	282.9M	27.7M	10.4M	401.6K	614.85	1.00	0.62	0.88
tsp	312.0M	38.0	270.1M	25.2M	10.4M	430.7K	480.34	1.00	0.52	0.90
OmpSCR-v2.0-c-LoopsWithDepende1	337.2M	226.0	180.8M	123.5M	7.3K	239.65	2.0K	118.00	54.33	75.37
OmpSCR-v2.0-c-LoopsWithDepende2	337.3M	226.0	181.0M	124.8M	8.5K	274.67	2.0K	116.00	55.82	77.00
pmd	367.0M	54.0	104.5M	35.4M	144.8K	3.5K	0.96	1.6K	50.43	0.96
DRB177-fib-taskdep-yes-211M-ev	382.1M	70.0	187.4M	45.3M	5.7K	188.85	488.33	1.8K	164.80	353.00
OmpSCR-v2.0-c-LoopsWithDepende3	394.0M	226.0	207.5M	128.8M	9.6K	250.02	483.77	118.00	55.55	75.58
OmpSCR-v2.0-c-LoopsWithDepende4	394.0M	226.0	205.3M	128.7M	9.4K	127.56	370.32	113.00	46.31	65.63
zero-reversal-logs-final-logs2	397.8M	24.0	297.6M	16.0M	1.00	0.02	0.00	1.00	0.02	0.00
montecarlo	494.0M	18.0	102.0M	12.3M	57.2K	1.6K	129.52	5.00	1.15	1.00
OmpSCR-v2.0-c-fft-496M-events	496.0M	70.0	115.8M	45.9M	2.2M	62.2K	28.10	2.0M	57.5K	1.73
OmpSCR-v2.0-c-fft-496M-events1	496.0M	230.0	102.6M	135.9M	2.1M	63.8K	302.18	2.1M	63.8K	18.53
OMPRacer-Lulesh-543M-events-16	543.4M	70.0	91.2M	46.2M	119.8M	3.6M	209.5K	326.6K	203.3K	840.27

1	2	3	4	5	6	7	8	9	10	11
Benchmark	trace length	M	Pacer sampled	**RPT** sampled	# Warnings			# Warning variables		
					FT	Pacer	**RPT**	FT	Pacer	**RPT**
misc-zxing-zxing	546.4M	64.0	97.8M	42.4M	10.1M	264.2K	9.40	27.2K	754.49	1.58
OMPRacer-Lulesh-569M-events-56	569.5M	230.0	114.1M	138.3M	145.3M	4.1M	2.5M	470.3K	309.7K	21.8K
luindex	570.0M	24.0	298.8M	16.3M	1.00	0.07	0.00	1.00	0.07	0.00
zero-reversal-logs-final-logs3	606.9M	22.0	51.7M	15.0M	0.00	0.00	0.00	0.00	0.00	0.00
sor	608.0M	18.0	49.0M	12.3M	0.00	0.00	0.00	0.00	0.00	0.00
OmpSCR-v2.0-c-LoopsWithDepende2	112.6M	66.0	14.2M	37.4M	2.9K	57.60	106.07	30.00	14.16	19.00
OmpSCR-v2.0-c-LoopsWithDepende3	112.6M	66.0	14.1M	37.0M	2.9K	70.86	89.90	30.00	13.98	18.93
OmpSCR-v2.0-cpp-sortOpenMP-cpp6	114.2M	228.0	75.4M	85.3M	1.1M	72.6K	133.0K	99.7K	9.3K	15.7K
OmpSCR-v2.0-cpp-sortOpenMP-cpp7	115.4M	228.0	83.9M	84.9M	56.00	2.00	40.58	56.00	2.00	40.58
OmpSCR-v2.0-c-Mandelbrot-116M	115.7M	66.0	55.8M	37.4M	26.00	0.95	5.03	26.00	0.95	5.03
OmpSCR-v2.0-c-Mandelbrot-116M1	115.7M	226.0	115.7M	85.6M	87.00	55.98	28.78	87.00	55.98	28.78
OMPRacer-Kripke-117M-events-16	117.5M	68.0	13.1M	38.6M	12.2M	998.9K	91.4K	175.1K	68.0K	22.8K
DRB177-fib-taskdep-yes-618M-ev	618.3M	70.0	303.0M	46.5M	9.6K	297.89	522.72	2.2K	251.45	375.13
DataRaceBench-DRB176-fib-taskd	618.3M	70.0	303.9M	46.5M	7.9K	203.64	489.02	2.4K	174.24	353.95
DRB177-fib-taskdep-yes-618M-ev1	618.3M	230.0	303.0M	139.8M	27.5K	1.1K	4.6K	6.8K	934.60	2.5K
DRB176-fib-taskdep-no-341M-eve	618.3M	230.0	303.6M	139.8M	49.3K	1.8K	9.2K	9.7K	1.4K	4.3K
OmpSCR-v2.0-c-LoopsWithDepende5	674.2M	226.0	349.6M	137.1M	6.7K	198.91	187.25	116.00	50.64	64.22
xalan	1000.0M	60.0	225.0M	40.5M	2.3K	7.77	8.98	202.00	7.27	0.82
DRB177-fib-taskdep-yes-552M-ev	1.0B	70.0	490.6M	47.2M	7.6K	240.22	241.97	2.1K	209.13	199.43
OMPRacer-RSBench-1.2B-events-1	1.3B	66.0	828.8M	44.7M	22.00	0.78	0.28	22.00	0.78	0.28
OMPRacer-RSBench-1.2B-events-5	1.3B	226.0	882.0M	146.9M	95.00	5.95	12.23	95.00	5.95	12.23
DRB177-fib-taskdep-yes-1.6B-ev	1.6B	70.0	793.8M	47.6M	11.4K	374.89	258.20	2.4K	306.84	209.53
DRB176-fib-taskdep-no-1.6B-eve	1.6B	70.0	795.7M	47.6M	12.5K	396.09	319.93	2.9K	309.67	245.15
DRB176-fib-taskdep-no-1.6B-eve1	1.6B	230.0	795.4M	151.2M	71.6K	2.5K	5.4K	11.1K	1.9K	3.0K
moldyn	1.7B	18.0	650.1M	12.4M	17.6M	668.5K	747.57	18.4K	17.5K	3.24
OmpSCR-v2.0-c-fft-2.1B-events	2.1B	230.0	477.8M	152.9M	8.1M	233.4K	56.27	7.8M	233.4K	4.72
avrora	2.4B	32.0	956.3M	22.0M	3.1M	93.0K	20.0K	414.7K	17.5K	4.4K
raytracer	2.8B	18.0	1.5B	12.4M	7.00	1.42	0.00	4.00	1.19	0.00

REFERENCES

2021. *Intel®Inspector.* https://software.intel.com/content/www/us/en/develop/tools/inspector.html Accessed: 2021-11-01.

Martin Abadi, Cormac Flanagan, and Stephen N. Freund. 2006. Types for Safe Locking: Static Race Detection for Java. *ACM Trans. Program. Lang. Syst.* 28, 2 (March 2006), 207–255.

Advanced Simulation and Computing, LLNL. 2022a. *CORAL Benchmarks.* https://asc.llnl.gov/coral-benchmarks Accessed: 2022-04-11.

Advanced Simulation and Computing, LLNL. 2022b. *CORAL Benchmarks.* https://asc.llnl.gov/coral-2-benchmarks Accessed: 2022-04-11.

Sarita Adve. 2010. Data races are evil with no exceptions: Technical perspective. *Commun. ACM* 53, 11 (2010), 84–84.

Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. 2018. Race Detection and Reachability in Nearly Series-Parallel DAGs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana) *(SODA '18)*. Society for Industrial and Applied Mathematics, USA, 156–171.

Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. 2001. Regular languages are testable with a constant number of queries. *SIAM J. Comput.* 30, 6 (2001), 1842–1862.

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. 2017. Lightweight Data Race Detection for Production Runs. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC 2017)*. Association for Computing Machinery, New York, NY, USA, 11–21. https://doi.org/10.1145/3033019.3033020

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 144 (oct 2018), 28 pages. https://doi.org/10.1145/3276514

Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism* (Berkeley, CA) *(HotPar'11)*. USENIX Association, USA, 3.

Hans-J. Boehm and Sarita Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 68–78.

Michael Bond. 2021. *Pacer: Proportional Detection of Data Races*. https://sourceforge.net/p/jikesrvm/research-archive/28/ Accessed: 2021-11-01.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. OCTET: Capturing and Controlling Cross-thread Dependences Efficiently. *SIGPLAN Not.* 48, 10 (Oct. 2013), 693–712. https://doi.org/10.1145/2544173.2509519

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, Pennsylvania, USA) *(ASPLOS XV)*. Association for Computing Machinery, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2018. Testing Multithreaded Programs via Thread Speed Control. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 15–25. https://doi.org/10.1145/3236024.3236077

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting Data Races in Cilk Programs That Use Locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. ACM, New York, NY, USA, 298–309.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. ACM, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. 2012. RADISH: Always-on sound and complete race detection in software and hardware. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 201–212. https://doi.org/10.1109/ISCA.2012.6237018

Dimitar Dimitrov, Martin Vechev, and Vivek Sarkar. 2015. Race Detection in Two Dimensions. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 101–110. https://doi.org/10.1145/2755573.2755601

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal* 10 (2005), 405–435.

A.J. Dorta, C. Rodriguez, and F. de Sande. 2005. The OpenMP source code repository. In *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. 244–250. https://doi.org/10.1109/EMPDP.2005.41

Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162.

Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. 2009. Meta-analysis for Atomicity Violations Under Nested Locking. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 248–262. https://doi.org/10.1007/978-3-642-02658-4_21

Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures* (Newport, Rhode Island, USA) *(SPAA '97)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/258492.258493

Colin Fidge. 1991. Logical Time in Distributed Computing Systems. *Computer* 24, 8 (Aug. 1991), 28–33. https://doi.org/10.1109/2.84874

Cormac Flanagan and Stephen N. Freund. 2000. Type-based Race Detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. ACM, New York, NY, USA, 219–232. https://doi.org/10.1145/349299.349328

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. ACM, New York, NY, USA, 1–8.

Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 255–280.

Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for Atomicity: Static Checking and Inference for Java. *ACM Trans. Program. Lang. Syst.* 30, 4, Article 20 (Aug. 2008), 53 pages. https://doi.org/10.1145/1377492.1377495

Oded Goldreich. 2017. *Introduction to Property Testing*. Cambridge University Press.

Klaus Havelund. 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 245–264.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. 1999. Toward Integration of Data Race Detection in DSM Systems. *J. Parallel Distrib. Comput.* 59, 2 (Nov. 1999), 180–203. https://doi.org/10.1006/jpdc.1999.1574

Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. 2005. Reasoning About Threads Communicating via Locks. In *Proceedings of the 17th International Conference on Computer Aided Verification* (Edinburgh, Scotland, UK) *(CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 505–518. https://doi.org/10.1007/11513988_49

Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 406–422.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2018. Data Race Detection on Compressed Traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/3236024.3236025

Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *32nd International Conference on Concurrency Theory (CONCUR 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:23. https://doi.org/10.4230/LIPIcs.CONCUR.2021.16

Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: A Benchmark Suite for Systematic Evaluation of Data Race Detection Tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 14 pages. https://doi.org/10.1145/3126908.3126958

LLNL. 2022. *ECP Proxy Applications*. https://proxyapps.exascaleproject.org Accessed: 2021-04-01.

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1542476.1542491

Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 145 (Oct. 2018),

29 pages. https://doi.org/10.1145/3276515

Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 710–725. https://doi.org/10.1145/3503222.3507734

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (jan 2021), 29 pages. https://doi.org/10.1145/3434317

Umang Mathur and Mahesh Viswanathan. 2020. Atomicity Checking in Linear Time Using Vector Clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 183–199. https://doi.org/10.1145/3373376.3378475

Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

Arndt Müehlenfeld and Franz Wotawa. 2007. Fault Detection in Multi-threaded C++ Server Applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *(PPoPP '07)*. ACM, New York, NY, USA, 142–143. https://doi.org/10.1145/1229428.1229457

Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-Based Controlled Concurrency Testing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 230 (nov 2020), 31 pages. https://doi.org/10.1145/3428298

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 446–455. https://doi.org/10.1145/1250734.1250785

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280.

Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/1250734.1250738

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/781498.781528

Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California, USA) *(PPoPP '03)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. ACM, New York, NY, USA, 531–542. https://doi.org/10.1145/2254064.2254127

Dustin Rhodes, Cormac Flanagan, and Stephen N. Freund. 2017. BigFoot: Static Check Placement for Dynamic Race Detection. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 141–156. https://doi.org/10.1145/3062341.3062350

Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: Efficient Predictive Race Detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 747–762. https://doi.org/10.1145/3385412.3385993

Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the Third International Conference on NASA Formal Methods* (Pasadena, CA) *(NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 313–327.

Sandia National Laboratories. 2022. *Mantevo Project.* Accessed: 2022-04-11.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

Adrian Schmitz, Joachim Protze, Lechen Yu, Simon Schwitanski, and Matthias S. Müller. 2020. DataRaceOnAccelerator – A Micro-benchmark Suite for Evaluating Correctness Tools Targeting Accelerators. In *Euro-Par 2019: Parallel Processing Workshops*, Ulrich Schwardmann, Christian Boehme, Dora B. Heras, Valeria Cardellini, Emmanuel Jeannot, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Dieter Schwamborn, Laura Ricci, Oh Sangyoon, Thomas Gruber, Laura Antonelli, and Stephen L. Scott (Eds.). Springer International Publishing, Cham, 245–257. https://doi.org/10.1007/978-3-030-48340-1_19

Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

Koushik Sen, Grigore Roşu, and Gul Agha. 2005. Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In *Proceedings of the 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems* (Athens, Greece) *(FMOODS'05)*. Springer-Verlag, Berlin, Heidelberg, 211–226.

Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. ACM, New York, NY, USA, 62–71.

Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitry Vyukov. 2011. *Dynamic Race Detection with LLVM Compiler.* Technical Report. Google.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

L. A. Smith, J. M. Bull, and J. Obdrizalek. 2001. A Parallel Java Grande Benchmark Suite. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. 6–6. https://doi.org/10.1145/582034.582042

Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. ACM, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

Rishi Surendran and Vivek Sarkar. 2016. Dynamic determinacy race detection for task parallelism with futures. In *International Conference on Runtime Verification*. Springer, 368–385.

Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2016. Concurrency Testing Using Controlled Schedulers: An Empirical Study. *ACM Trans. Parallel Comput.* 2, 4, Article 23 (feb 2016), 37 pages. https://doi.org/10.1145/2858651

Christoph von Praun and Thomas R. Gross. 2003. Static Conflict Analysis for Multi-threaded Object-oriented Programs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. ACM, New York, NY, USA, 115–128. https://doi.org/10.1145/781131.781145

Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. 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* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214.

Jaroslav Ševčík. 2011. Safe optimizations for shared-memory concurrent programs. *SIGPLAN Notices* 46, 6 (2011), 306–316.

Jaroslav Ševčík and David Aspinall. 2008. On the validity of program transformations in the Java memory model. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 27–51.

Benjamin P. Wood, Man Cao, Michael D. Bond, and Dan Grossman. 2017. Instrumentation Bias for Dynamic Data Race Detection. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 69 (Oct. 2017), 31 pages. https://doi.org/10.1145/3133893

Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

M. Zhivich and R. K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security and Privacy* 7, 2 (March 2009), 87–90. https://doi.org/10.1109/MSP.2009.56

Received 2022-07-07; accepted 2022-11-07