What Happens-After the First Race? Enhancing the Predictive Power of Happens-Before Based Dynamic Race Detection

UMANG MATHUR, University of Illinois, Urbana Champaign, USA
DILEEP KINI, Akuna Capital LLC, USA
MAHESH VISWANATHAN, University of Illinois, Urbana Champaign, USA

Dynamic race detection is the problem of determining if an observed program execution reveals the presence of a data race in a program. The classical approach to solving this problem is to detect if there is a pair of conflicting memory accesses that are unordered by Lamport's happens-before (HB) relation. HB based race detection is known to not report false positives, i.e., it is sound. However, the soundness guarantee of HB only promises that the first pair of unordered, conflicting events is a *schedulable* data race. That is, there can be pairs of HB-unordered conflicting data accesses that are not schedulable races because there is no reordering of the events of the execution, where the events in race can be executed immediately after each other. We introduce a new partial order, called schedulable happens-before (SHB) that exactly characterizes the pairs of schedulable data races — every pair of conflicting data accesses that are identified by SHB can be scheduled, and every HB-race that can be scheduled is identified by SHB. Thus, the SHB partial order is truly sound. We present a linear time, vector clock algorithm to detect schedulable races using SHB. Our experiments demonstrate the value of our algorithm for dynamic race detection — SHB incurs only little performance overhead and can scale to executions from real-world software applications without compromising soundness.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; *Formal software verification*;

Additional Key Words and Phrases: Concurrency, Race Detection, Dynamic Program Analysis, Soundness, Happens-Before

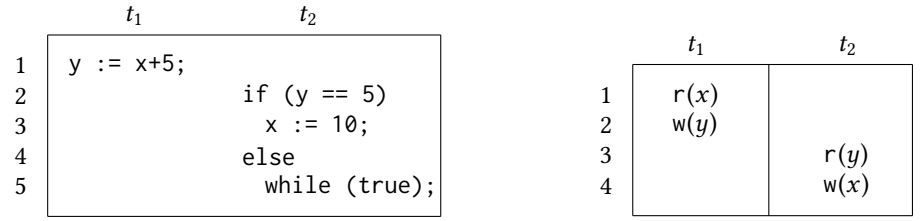
ACM Reference Format:
Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-After the First Race? Enhancing the Predictive Power of Happens-Before Based Dynamic Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 145 (November 2018), 29 pages. https://doi.org/10.1145/3276515

1 INTRODUCTION

The presence of data races in concurrent software is the most common indication of a programming error. Data races in programs can result in nondeterministic behavior that can have unintended consequences. Further, manual debugging of such errors is prohibitively difficult owing to nondeterminism. Therefore, automated detection and elimination of data races is an important problem that has received widespread attention from the research community. Dynamic race detection techniques examine a single execution of a concurrent program to discover a data race in the program. In this paper we focus on dynamic race detection.

Authors' addresses: Umang Mathur, Department of Computer Science, University of Illinois, Urbana Champaign, USA, umathur3@illinois.edu; Dileep Kini, Akuna Capital LLC, USA, dileeprkini@gmail.com; Mahesh Viswanathan, Department of Computer Science, University of Illinois, Urbana Champaign, USA, vmahesh@illinois.edu.

2475-1421/2018/11-ART145
https://doi.org/10.1145/3276515



(a) Example concurrent program P_1.

(b) Trace σ_1 generated from P_1.

Fig. 1. Concurrent program P_1 and its sample execution σ_1. Initially $x = y = 0$.

Dynamic race detection may either be sound or unsound. Unsound techniques, like lockset based methods [Savage et al. 1997], have low overhead but they report potential races that are spurious. Sound techniques [Lamport 1978; Mattern 1988; Said et al. 2011; Huang et al. 2014; Smaragdakis et al. 2012; Kini et al. 2017], on the other hand, never report the presence of a data race, if none exist. The most popular, sound technique is based on computing the *happens-before* (HB) partial order [Lamport 1978] on the events of the trace, and declares a data race when there is a pair of conflicting events (reads/writes to a common memory location performed by different threads, at least one of which is a write operation) that are unordered by the partial order. There are two reasons for the popularity of the HB technique. First, because it is sound, it does not report false positives. Low false positive rates are critical for the wide-spread use of debugging techniques [Serebryany and Iskhodzhanov 2009; Sadowski and Yi 2014]. Second, even though HB-based algorithms may miss races detected by other sound techniques [Said et al. 2011; Huang et al. 2014; Smaragdakis et al. 2012; Kini et al. 2017], they have the lowest overhead among sound techniques. Many improvements [Pozniansky and Schuster 2003; Flanagan and Freund 2009; Elmas et al. 2007] to the original vector clock algorithm [Mattern 1988] have helped reduce the overhead even further.

However, HB-based dynamic analysis tools suffer from some drawbacks. Recall that a program has a data race, if there is some execution of the program where a pair of conflicting data accesses are performed consecutively. Even though HB is a sound technique, its soundness guarantee is only limited to the *first* pair of unordered conflicting events; a formal definition of "first" unordered pair is given later in the paper. Thus, a trace may have many HB-unordered pairs of conflicting events (popularly called *HB-races*) that do not correspond to data races. To see this, consider the example program and trace shown in Fig. 1. The trace corresponds to first executing the statement of thread t_1, before executing the statements of thread t_2. The statement `y := x + 5` requires first reading the value of x (which is 0) and then writing to y. Recall that HB orders (i) two events performed by the same thread, and (ii) synchronization events performed by different threads, in the order in which they appear in the trace. Using e_i to denote the ith event of the trace, in this trace since there are no synchronization events, both (e_1, e_4) and (e_2, e_3) are in HB race. Observe that while e_2 and e_3 can appear consecutively in a trace (as in Fig. 1b), there is no trace of the program where e_1 and e_4 appear consecutively. Thus, even though the events e_1 and e_4 are unordered by HB, they do not constitute a data race.

As a consequence, developers typically fix the first race discovered, re-run the program and the dynamic race detection algorithm, and repeat the process until no races are discovered. This approach to bug fixing suffers from many disadvantages. First, running race detection algorithms can be expensive [Sadowski and Yi 2014], and so running them many times is a significant overhead. Second, even though only the first HB race is guaranteed to be a real race, it doesn't mean that it

(a) Example concurrent program P_2.

(b) Trace σ_2 generated from P_2.

Fig. 2. Concurrent program P_2 and its sample execution σ_2. Initially $x = y = 0$.

is the *only* HB race that is real. Consider the example shown in Fig. 2. In the trace σ_2 (shown in Fig. 2b), both pairs (e_1, e_4) and (e_2, e_3) are in HB-race. σ_2 demonstrates that (e_2, e_3) is a valid data race (because they are scheduled consecutively). But (e_1, e_4) is also a valid data race. This can be seen by first executing `y := 1;` in thread t_2, followed by `if (x == 0) skip;` in thread t_1, and then finally `x := 2;` in t_2. The approach of fixing the first race, and then re-executing and performing race detection, not only unnecessarily ignores the race (e_1, e_4), but it might miss it completely because (e_1, e_4) might not show up as a HB race in the next execution due to the inherent nondeterminism when executing multi-threaded programs. As a result, most practical race detection tools including ThreadSanitizer [Serebryany and Iskhodzhanov 2009], Helgrind [Müehlenfeld and Wotawa 2007] and FastTrack [Flanagan and Freund 2009] report more than one race, even if those races are likely to be false, to give software developers the opportunity to fix more than just the first race. In our companion technical report [Mathur et al. 2018], we illustrate this observation on four practical dynamic race detection tools based on the happens-before partial order. Each of these tools resort to naïvely reporting races beyond the first race and produce false positives as a result.

The central question we would like to explore in this paper is, can we detect multiple races in a given trace, soundly? One approach would be to mimic the software developer's strategy in using HB-race detectors — every time a race is discovered, force an order between the two events constituting the race and then analyze the subsequent events. This ensures that the HB soundness theorem then applies to the *next* race discovered, and so on. Such an algorithm can be proved to only discover valid data races. For example, in trace σ_1 (Fig. 1), after discovering the race (e_2, e_3) assume that the events e_2 and e_3 are ordered when analyzing events after e_3 in the trace. By this algorithm, when we process event e_4, we will conclude that (e_1, e_4) are not in race because e_1 comes before e_2, e_2 has been force ordered before e_3, and e_3 is before e_4, and so e_1 is ordered before e_4. However, force ordering will miss valid data races present in the trace. Consider the trace σ_2 from Fig. 2. Here the force ordering algorithm will only discover the race (e_2, e_3) and will miss (e_1, e_4) which is a valid data race. Another approach [Huang et al. 2014], is to search for a reordering of the events in the trace that respects the data dependencies amongst the read and write events, and the effect of synchronization events like lock acquires and releases. Here one encodes the event dependencies as logical constraints, where the correct reordering of events corresponds to a satisfying truth assignment. The downside of this approach is that the SAT formula encoding event dependencies can be huge even for a trace with a few thousand events. Typically, to avoid the prohibitive cost of determining the satisfiability of such a large formula, the trace is broken up into small "windows", and the formula only encodes the dependencies of events within a window. In addition, solver timeouts are added to give up the search for another reordering. As a consequence this approach can miss many data races in practice (see our experimental evaluation in Section 5).

In this paper, we present a new partial order on events in an execution that we call *schedulable happens-before* (SHB) to address these challenges. Unlike recent attempts [Smaragdakis et al. 2012; Kini et al. 2017] to *weaken* HB to discover more races, SHB is a *strengthening* of HB — some HB unordered events, will be ordered by SHB. However, the first HB race (which is guaranteed to be a real data race by the soundness theorem for HB) will also be SHB unordered. Further, every race detected using SHB is a valid, schedulable race. In addition, we prove that, not only does SHB discover every race found by the naïve force ordering algorithm and more (for example, SHB will discover both races in Fig. 2), it will detect *all HB-schedulable* races. The fact that SHB detects precisely the set of HB-schedulable races, we hope, will make it popular among software developers because of its enhanced predictive power per trace and the absence of false positives.

We then present a simple vector clock based algorithm for detecting all SHB races. Because the algorithm is very close to the usual HB vector clock algorithm, it has a low overhead. We also show how to adapt existing improvements to the HB algorithm, like the use of *epochs* [Flanagan and Freund 2009], into the SHB algorithm to lower overhead. We believe that existing HB-based detectors can be easily modified to leverage the greater power of SHB-based analysis. We have implemented our SHB algorithm and analyzed its performance on standard benchmarks. Our experiments demonstrate that (a) many HB unordered conflicting events may not be valid data races, (b) there are many valid races missed by the naïve force ordering algorithm, (c) SHB based analysis poses only a little overhead as compared to HB based vector clock algorithm, and (d) improvements like the use of epochs, are effective in enhancing the performance of SHB analysis.

The rest of the paper is organized as follows: Section 2 introduces notations and definitions relevant for the paper. In Section 3, we introduce the partial order SHB and present an exact characterization of schedulable races using this partial order. In Section 4, we describe a vector clock algorithm for detecting schedulable races based on SHB. We then show how to incorporate epoch-based optimizations to this vector clock algorithm. Section 5 describes our experimental evaluation. We discuss relevant related work in Section 6 and present concluding remarks in Section 7.

2 PRELIMINARIES

In this section, we will fix notation and present some definitions that will be used in this paper.

Traces. We consider concurrent programs under the sequential consistency model. Here, an execution, or trace, of a program is viewed as an interleaving of operations performed by different threads. We will use σ, σ' and σ'' to denote traces. For a trace σ, we will use $\mathsf{Threads}_\sigma$ to denote the set of threads in σ. A trace is a sequence of events of the form $e = \langle t, op \rangle$, where $t \in \mathsf{Threads}_\sigma$, and op can be one of $\mathtt{r}(x)$, $\mathtt{w}(x)$ (read or write to memory location x), $\mathtt{acq}(\ell)$, $\mathtt{rel}(\ell)$ (acquire or release of lock ℓ) and $\mathtt{fork}(u)$, $\mathtt{join}(u)$ (fork or join of some thread u) [1]. To keep the presentation simple, we assume that locks are not reentrant. However, all the results can be extended to the case when locks are assumed to be reentrant. The set of events in trace σ will be denoted by Events_σ. We will also use $\mathsf{Reads}_\sigma(x)$ (resp. $\mathsf{Writes}_\sigma(x)$) to denote the set of events that read from (resp. write to) memory location x. Further Reads_σ (resp. Writes_σ) denotes the union of the above sets over all memory locations. For an event $e \in \mathsf{Reads}_\sigma(x)$, the *last write before* e is the (unique) event $e' \in \mathsf{Writes}_\sigma(x)$ such that e' appears before e in the trace σ, and there is no event $e'' \in \mathsf{Writes}_\sigma(x)$ between e' and e in σ. The last write before event $e \in \mathsf{Reads}_\sigma(x)$ maybe undefined, if there is no $\mathtt{w}(x)$-event before e. We denote the last write before e by $\mathsf{lastWr}_\sigma(e)$. An event $e = \langle t_1, op \rangle$ is said

[1] Formally, each event in a trace is assumed to have a unique event id. Thus, two occurences of a thread performing the same operation will be considered *different* events. Even though we will implicitly assume the uniqueness of each event in a trace, to reduce notational overhead, we do not formally introduce event ids.

	t_1	t_2	t_3	t_4
1	$\texttt{acq}(\ell)$			
2	$\texttt{w}(x)$			
3	$\texttt{rel}(\ell)$			
4		$\texttt{acq}(\ell)$		
5		$\texttt{w}(x)$		
6		$\texttt{rel}(\ell)$		
7			$\texttt{r}(x)$	
8			$\texttt{fork}(t_4)$	
9				$\texttt{w}(x)$
10				$\texttt{w}(x)$
11			$\texttt{join}(t_4)$	
12			$\texttt{r}(x)$	

Fig. 3. Trace σ_3.

to be *an event of thread t* if either $t = t_1$ or $op \in \{\texttt{fork}(t), \texttt{join}(t)\}$. The *projection* of a trace σ to a thread $t \in \mathsf{Threads}_\sigma$ is the maximal subsequence of σ that contains only events of thread t, and will be denoted by $\sigma|_t$; thus an event $e = \langle t, \texttt{fork}(t')\rangle$ (or $e = \langle t, \texttt{join}(t')\rangle$) belongs to both $\sigma|_t$ and $\sigma|_{t'}$. For an event e of thread t, we denote by $\mathsf{pred}_\sigma(e)$ to be the last event e' before e in σ such that e and e' are events of the same thread. Again, $\mathsf{pred}_\sigma(e)$ may be undefined for an event e. The projection of σ to a lock ℓ, denoted by $\sigma|_\ell$, is the maximal subsequence of σ that contains only acquire and release events of lock ℓ. Traces are assumed to be well formed — for every lock ℓ, $\sigma|_\ell$ is a prefix of some string belonging to the regular language $(\cup_{t\in\mathsf{Threads}_\sigma}\langle t, \texttt{acq}(\ell)\rangle \cdot \langle t, \texttt{rel}(\ell)\rangle)^*$.

Example 2.1. Let us illustrate the definitions and notations about traces introduced in the previous paragraph. Consider the trace σ_3 shown in Fig. 3. As in the introduction, we will refer to the ith event in the trace by e_i. For trace σ_3 we have — $\mathsf{Events}_{\sigma_3} = \{e_1, e_2, \ldots e_{12}\}$; $\mathsf{Reads}_{\sigma_3} = \mathsf{Reads}_{\sigma_3}(x) = \{e_7, e_{12}\}$; $\mathsf{Writes}_{\sigma_3} = \mathsf{Writes}_{\sigma_3}(x) = \{e_2, e_5, e_9, e_{10}\}$. The last write of the read events is as follows: $\mathsf{lastWr}_{\sigma_3}(e_7) = e_5$ and $\mathsf{lastWr}_{\sigma_3}(e_{12}) = e_{10}$. The projection with respect to lock ℓ is $\sigma_3|_\ell = e_1e_3e_4e_6$. The definition of projection to a thread is subtle in the presence of forks and joins. This can be seen by observing that $\sigma_3|_{t_4} = e_8e_9e_{10}e_{11}$; this is because the fork event e_8 and the join event e_{11} are considered to be events of both threads t_3 and t_4 by our definition. Finally, we illustrate $\mathsf{pred}_{\sigma_3}(\cdot)$ through a few examples — $\mathsf{pred}_{\sigma_3}(e_2) = e_1$, $\mathsf{pred}_{\sigma_3}(e_7)$ is undefined, $\mathsf{pred}_{\sigma_3}(e_9) = e_8$, and $\mathsf{pred}_{\sigma_3}(e_{11}) = e_{10}$. The cases of e_9 and e_{11} are the most interesting, and they follow from the fact that both e_8 and e_{11} are also considered to be events of t_4.

Orders. A given trace σ induces several total and partial orders. The total order $\leq^\sigma_{\mathsf{tr}} \subseteq \mathsf{Events}_\sigma \times \mathsf{Events}_\sigma$, will be used to denote the *trace-order* — $e \leq^\sigma_{\mathsf{tr}} e'$ iff either $e = e'$ or e appears before e' in the sequence σ. Similarly, the *thread-order* is the smallest partial order $\leq^\sigma_{\mathsf{TO}} \subseteq \mathsf{Events}_\sigma \times \mathsf{Events}_\sigma$ such that for all pairs of events $e \leq^\sigma_{\mathsf{tr}} e'$ of the same thread, we have $e \leq^\sigma_{\mathsf{TO}} e'$.

Definition 2.2 (Happens-Before). Given trace σ, the happens-before order $\leq^\sigma_{\mathsf{HB}}$ is the smallest partial order on Events_σ such that

(a) $\leq^\sigma_{\mathsf{TO}} \subseteq \leq^\sigma_{\mathsf{HB}}$,
(b) for every pair of events $e = \langle t, \texttt{rel}(\ell)\rangle$, and, $e' = \langle t', \texttt{acq}(\ell)\rangle$ with $e \leq^\sigma_{\mathsf{tr}} e'$, we have $e \leq^\sigma_{\mathsf{HB}} e'$

Example 2.3. We illustrate the definitions of $\leq_{\mathsf{tr}}$, $\leq_{\mathsf{TO}}$, and $\leq_{\mathsf{HB}}$ using trace σ_3 from Fig. 3. Trace order is the simplest; $e_i \leq^{\sigma_3}_{\mathsf{tr}} e_j$ iff $i \leq j$. Thread order is also straightforward in most cases; the

interesting cases of $e_8 \leq^{\sigma_3}_{\mathsf{TO}} e_9$ and $e_{10} \leq^{\sigma_3}_{\mathsf{TO}} e_{11}$ follow from the fact that e_8 and e_{11} are events of both threads t_3 and t_4. Finally, let us consider $\leq^{\sigma_3}_{\mathsf{HB}}$. It is worth observing that $e_7 \leq^{\sigma_3}_{\mathsf{HB}} e_9 \leq^{\sigma_3}_{\mathsf{HB}} e_{10} \leq^{tr_3}_{\mathsf{HB}} e_{12}$ simply because these events are thread ordered due to the fact that e_8 and e_{11} are events of both thread t_3 and t_4. In addition, $e_2 \leq^{\sigma_3}_{\mathsf{HB}} e_5$ because $e_3 \leq^{\sigma_3}_{\mathsf{HB}} e_4$ by rule (b), $e_2 \leq^{\sigma_3}_{\mathsf{TO}} e_3$ and $e_4 \leq^{\sigma_3}_{\mathsf{TO}} e_5$, and $\leq^{\sigma_3}_{\mathsf{HB}}$ is transitive.

Trace Reorderings. Any trace of a concurrent program represents one possible interleaving of concurrent events. The notion of *correct reordering* [Smaragdakis et al. 2012; Kini et al. 2017] of trace σ identifies all these other possible interleavings of σ. In other words, if σ' is a correct reordering of σ then any program that produces σ may also produce σ'. The definition of correct reordering is given purely in terms of the trace σ and is agnostic of the program that produced it. We give the formal definition below.

Definition 2.4 (Correct reordering). A trace σ' is said to be a correct reordering of a trace σ if

(a) $\forall t \in \mathsf{Threads}_{\sigma'}, \sigma'|_t$ is a prefix of $\sigma|_t$, and
(b) for a read event $e = \langle t, \mathsf{r}(x)\rangle \in \mathsf{Events}_{\sigma'}$ such that e is not the last event in $\sigma'|_t$, $\mathsf{lastWr}_{\sigma'}(e)$ exists iff $\mathsf{lastWr}_{\sigma}(e)$ exists. Further, if it exists, then $\mathsf{lastWr}_{\sigma'}(e) = \mathsf{lastWr}_{\sigma}(e)$.

The intuition behind the above definition is the following. A correct reordering must preserve lock semantics (ensured by the fact that σ' is a trace) and the order of events inside a given thread (condition (a)). Condition (b) captures *local determinism* [Huang et al. 2014]. That is, the next event of a given thread can be completely determined by the earlier events in that thread. Now, the underlying program, that generated σ, can have conditional statements and the actual branches taken depend upon the data in shared memory locations. As a result, we demand that all reads in σ', with the exception of the last events in each thread, must see the same value as in σ. Since our traces don't record the value written, this can be ensured by conservatively requiring that every read event in σ' has the same last write event as in σ. We relax this requirement for read events that are last events in their corresponding threads. For example, consider the program and trace given in Fig. 1. The read event $\mathsf{r}(y)$ in the conditional in thread t_2 cannot be swapped with the preceding event $\mathtt{w}(y)$ in thread t_1, because such a swap would result in a different branch being taken in t_2, and the assignment `x := 10` in t_2 will never be executed. However, this is required only if the read event is not the last event of the thread in the reordering. If it is the last event, it does not matter what value is read, because it does not affect future behavior.

We note that the definition of correct reordering we have is more general than in [Kini et al. 2017; Smaragdakis et al. 2012] because of the relaxed assumption about the last-write events corresponding to read events which are not followed by any other events in their corresponding threads. In other words, every correct reordering σ' of a trace σ according to the definition in [Smaragdakis et al. 2012; Kini et al. 2017] is also a correct reordering of σ as per Definition 2.4, but the converse is not true. On the other hand, the related notion of *feasible* set of traces [Huang et al. 2014] is even more general and allows for an even larger set of alternate reorderings that can be inferred from an observed trace σ. We note that the read/write events in [Huang et al. 2014] also record the value read/written to the memory location. In this case, for a trace σ' to be in the feasible set of trace σ, [Huang et al. 2014] require that for every read event e, the value written by $\mathsf{lastWr}_{\sigma'}(e)$ equals the value written by $\mathsf{lastWr}_{\sigma}(e)$. In particular, $\mathsf{lastWr}_{\sigma'}(e)$ may not be the same as $\mathsf{lastWr}_{\sigma}(e)$, thus allowing for more reorderings.

In addition to correct reorderings, another useful collection of alternate interleavings of a trace is as follows. Under the assumption that $\leq^{\sigma}_{\mathsf{HB}}$ identifies certain causal dependencies between events of σ, we consider interleavings of σ that are consistent with $\leq^{\sigma}_{\mathsf{HB}}$.

Definition 2.5 ($\leq_{\mathsf{HB}}$-respecting trace). For trace σ, we say trace σ' respects $\leq^{\sigma}_{\mathsf{HB}}$ if for any $e, e' \in \mathsf{Events}_{\sigma}$ such that $e \leq^{\sigma}_{\mathsf{HB}} e'$ and $e' \in \mathsf{Events}_{\sigma'}$, we have $e \in \mathsf{Events}_{\sigma'}$ and $e \leq^{\sigma'}_{\mathsf{tr}} e'$.

Thus, a $\leq^{\sigma}_{\mathsf{HB}}$-respecting trace is one whose events are downward closed with respect to $\leq^{\sigma}_{\mathsf{HB}}$ and in which $\leq^{\sigma}_{\mathsf{HB}}$-ordered events are not flipped. We will be using the above notion only when the trace σ' is a reordering of σ, and hence $\mathsf{Events}_{\sigma'} \subseteq \mathsf{Events}_{\sigma}$.

Example 2.6. We give examples of correct reorderings of σ_3 shown in Fig. 3. The traces $\rho_1 = e_1e_2e_7$, $\rho_2 = e_4e_5e_6$, and $\rho_3 = e_1e_2e_3e_4e_5e_7$ are all examples of correct reorderings of σ_3. Among these, the trace ρ_2 is not $\leq_{\mathsf{HB}}$-respecting because it is not $\leq_{\mathsf{HB}}$-downward closed — events e_1, e_2, e_3 are all HB-before e_4 and none of them are in ρ_2.

Race. It is useful to recall the formal definition of a data race, and to state the soundness guarantees of happens-before. Two data access events $e = \langle t_1, \mathsf{a}_1(x)\rangle$ and $e' = \langle t_2, \mathsf{a}_2(x)\rangle$ are said to be *conflicting* if $t_1 \neq t_2$, and at least one among e and e' is a write event ($\mathsf{a}_1 = \mathtt{w}$ or $\mathsf{a}_2 = \mathtt{w}$). A trace σ is said to have a race if it is of the form $\sigma = \sigma' e e' \sigma''$ such that e and e' are conflicting; here (e, e') is either called a race pair or a race. A concurrent program is said to have a race if it has an execution that has a race.

The partial order $\leq^{\sigma}_{\mathsf{HB}}$ is often employed for the purpose of detecting races by analyzing program executions. In this context, it is useful to define what we call an HB-race. A pair of conflicting events (e, e') is said to be an *HB-race* if $e \leq^{\sigma}_{\mathsf{tr}} e'$ and e and e' are incomparable with respect to $\leq^{\sigma}_{\mathsf{HB}}$ (i.e., neither $e \leq^{\sigma}_{\mathsf{HB}} e'$ nor $e' \leq^{\sigma}_{\mathsf{HB}} e$). We say an HB-race (e, e') is the *first HB-race* if for any other HB-race $(f, f') \neq (e, e')$ in σ, either $e' <^{\sigma}_{\mathsf{tr}} f'$, or $e' = f'$ and $f <^{\sigma}_{\mathsf{tr}} e$. For example, the pair (e_2, e_3) in trace σ_1 from Fig. 1 is the first HB-race of σ_1. The soundness guarantee of HB says that if a trace σ has an HB-race, then the first HB-race is a valid data race.

Theorem 2.7 (Soundness of HB). *Let σ be a trace with an HB-race, and let (e, e') be the first HB-race. Then, there is a correct reordering σ' of σ, such that $\sigma' = \sigma'' e e'$.*

Instead of sketching the proof of Theorem 2.7, we will see that it follows from the main result of this paper, namely, Theorem 3.3.

Example 2.8. We conclude this section by giving examples of HB-races. Consider again σ_3 from Fig. 3. Among the different pairs of conflicting events in σ_3, the HB-races are (e_2, e_7), (e_5, e_7), (e_2, e_9), (e_5, e_9), (e_2, e_{10}), (e_5, e_{10}), (e_2, e_{12}), and (e_5, e_{12}).

Remark. Our model of executions and reorderings assume sequential consistency, which is a standard model used by most race detection tools. Executions in a more general memory model, such as Total Store Order (TSO), would also have events that indicate when a local write was committed to the global memory [Huang and Huang 2016]. In that scenario, the definition of correct reorderings would be similar, except that "last write" would be replaced by "last observed write", which would either be the last committed write or the last write by the same thread, whichever is later in the trace. The number of correct reorderings to be considered would increase — instead of just considering executions where every write is immediately committed, as we do here, we would also need to consider reorderings where the write commits are delayed. However, since our results here are about proving the existence of a reordered trace where a race is observed, they carry over to the more general setting. We might miss race pairs that could be shown to be in race in a weaker memory model, where more reoderings are permitted, but the races we identify would still be valid.

3 CHARACTERIZING SCHEDULABLE RACES

The example in Fig. 1 shows that not every HB-race corresponds to an actual data race in the program. The goal of this section is to characterize those HB-races which correspond to actual data races. We do this by introducing a new partial order, called schedulable happens-before, and using it to identify the actual data races amongst the HB-races of a trace. We begin by characterizing the HB-races that correspond to actual data races.

Definition 3.1 ($\leq^{\sigma}_{\mathsf{HB}}$-schedulable race). Let σ be a trace and let $e \leq^{\sigma}_{\mathsf{tr}} e'$ be conflicting events in σ. We say that (e, e') is a $\leq^{\sigma}_{\mathsf{HB}}$-schedulable race if there is a correct reordering σ' of σ that respects $\leq^{\sigma}_{\mathsf{HB}}$ and $\sigma' = \sigma'' e e'$ or $\sigma' = \sigma'' e' e$ for some trace σ''.

Note that any $\leq^{\sigma}_{\mathsf{HB}}$-schedulable race is a valid data race in σ. Our aim is to characterize $\leq^{\sigma}_{\mathsf{HB}}$-schedulable races by means of a new partial order. The new partial order, given below, is a strengthening of $\leq_{\mathsf{HB}}$.

Definition 3.2 (Schedulable Happens-Before). Let σ be a trace. Schedulable happens-before, denoted by $\leq^{\sigma}_{\mathsf{SHB}}$, is the smallest partial order on Events_{σ} such that

(a) $\leq^{\sigma}_{\mathsf{HB}} \subseteq \leq^{\sigma}_{\mathsf{SHB}}$
(b) $\forall e, e' \in \mathsf{Events}_{\sigma}, e' \in \mathsf{Reads}_{\sigma} \wedge e = \mathsf{lastWr}_{\sigma}(e') \implies e \leq^{\sigma}_{\mathsf{SHB}} e'$

The partial order $\leq^{\sigma}_{\mathsf{SHB}}$ can be used to characterize $\leq^{\sigma}_{\mathsf{HB}}$-schedulable races. We state this result, before giving examples illustrating the definition of $\leq^{\sigma}_{\mathsf{SHB}}$.

Theorem 3.3. *Let σ be a trace and $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$ be conflicting events in σ. (e_1, e_2) is an $\leq^{\sigma}_{\mathsf{HB}}$-schedulable race iff either $\mathsf{pred}_{\sigma}(e_2)$ is undefined, or $e_1 \not\leq^{\sigma}_{\mathsf{SHB}} \mathsf{pred}_{\sigma}(e_2)$.*

Proof. (Sketch) The full proof is presented in Appendix A; here we sketch the main ideas. We observe that if σ' is a correct reordering of σ that also respects $\leq^{\sigma}_{\mathsf{HB}}$, then σ' also respects $\leq^{\sigma}_{\mathsf{SHB}}$ except possibly for the last events of every thread in σ'. That is, for any e, e' such that $e \leq^{\sigma}_{\mathsf{SHB}} e'$, $e' \in \mathsf{Events}_{\sigma'}$, and e' is not the last event of some thread in σ', we have $e \in \mathsf{Events}_{\sigma'}$ and $e \leq^{\sigma'}_{\mathsf{tr}} e'$. Therefore, if $e \leq^{\sigma}_{\mathsf{SHB}} \mathsf{pred}_{\sigma}(e_2)$, then any correct reordering σ' respecting $\leq^{\sigma}_{\mathsf{HB}}$ that contains both e_1 and e_2 will also have $e = \mathsf{pred}_{\sigma}(e_2)$. Further since e is not the last event of its thread (since e_2 is present in σ') and $e_1 \leq^{\sigma}_{\mathsf{SHB}} e$, e must occur between e_1 and e_2 in σ'. Therefore (e_1, e_2) is not a $\leq^{\sigma}_{\mathsf{HB}}$-schedulable race. The other direction can be established as follows. Let σ'' be the trace consisting of events that are $\leq^{\sigma}_{\mathsf{SHB}}$-before e_1 or $\mathsf{pred}_{\sigma}(e_2)$ (if defined), ordered as in σ. Define $\sigma' = \sigma'' e_1 e_2$. We prove that when e_1 and e_2 satisfy the condition in the theorem, σ' as defined here, is a correct reordering and also respects $\leq^{\sigma}_{\mathsf{HB}}$. □

Remark. We remark that the proof of Theorem 3.3 can be easily lifted to construct a trace that witnesses a given $\leq_{\mathsf{HB}}$-schedulable race (e_1, e_2). Demonstrating an actual trace witnessing a bug is very useful for debugging purposes and enhances confidence of programmers using the race detector.

We now illustrate the use of $\leq_{\mathsf{SHB}}$ through some examples.

Example 3.4. In this example, we will look at different traces, and see how $\leq_{\mathsf{SHB}}$ reasons. Like in the introduction, we will use e_i to refer to the ith event of a given trace (which will be clear from context). Let us begin by considering the example program and trace σ_1 from Fig. 1. Notice that $\leq^{\sigma_1}_{\mathsf{HB}} = \leq^{\sigma_1}_{\mathsf{TO}}$, and so (e_1, e_4) and (e_2, e_3) are HB-races. Because $e_2 = \mathsf{lastWr}_{\sigma_1}(e_3)$, we have $e_1 \leq^{\sigma_1}_{\mathsf{SHB}} e_2 \leq^{\sigma_1}_{\mathsf{SHB}} e_3 \leq^{\sigma_1}_{\mathsf{SHB}} e_4$. Using Theorem 3.3, we can conclude correctly that (a) (e_2, e_3) is $\leq^{\sigma_1}_{\mathsf{HB}}$-schedulable as $\mathsf{pred}_{\sigma_1}(e_3)$ is undefined, but (b) (e_1, e_4) is not, as $e_1 \leq^{\sigma}_{\mathsf{SHB}} \mathsf{pred}_{\sigma}(e_4) = e_3$.

	t_1	t_2	t_3	t_4
1	$\mathtt{acq}(\ell)$			
2	$\mathtt{w}(x)$			
3		$\mathtt{r}(x)$		
4		$\mathtt{w}(y)$		
5		$\mathtt{w}(x)$		
6	$\mathtt{r}(x)$			
7	$\mathtt{rel}(\ell)$			
8				$\mathtt{acq}(\ell)$
9				$\mathtt{w}(z)$
10			$\mathtt{r}(z)$	
11			$\mathtt{w}(y)$	
12			$\mathtt{w}(z)$	
13				$\mathtt{r}(z)$
14				$\mathtt{rel}(\ell)$

Fig. 4. Trace σ_4.

Let us now consider trace σ_2 from Fig. 2. Observe that $\leq_{\mathsf{HB}}^{\sigma_2}=\leq_{\mathsf{SHB}}^{\sigma_2}=\leq_{\mathsf{TO}}^{\sigma_2}$, and so both (e_1, e_4) and (e_2, e_3) are $\leq_{\mathsf{HB}}^{\sigma_2}$-schedulable races by Theorem 3.3. Note that, unlike force ordering, $\leq_{\mathsf{SHB}}^{\sigma_2}$ correctly identifies all real data races.

Finally, let us consider two trace examples that highlight the kind of subtle reasoning $\leq_{\mathsf{SHB}}$ is capable of. Let us begin with σ_3 from Fig. 3. As observed in Example 2.8, the only HB-races in this trace are (e_2, e_7), (e_5, e_7), (e_2, e_9), (e_5, e_9), (e_2, e_{10}), (e_5, e_{10}), (e_2, e_{12}), and (e_5, e_{12}). Both (e_2, e_7) and (e_5, e_7) are $\leq_{\mathsf{HB}}^{\sigma_3}$-schedulable as demonstrated by the reorderings ρ_1 and ρ_3 from Example 2.6. However, the remaining are not real data races. Let us consider the pairs (e_2, e_9) and (e_5, e_9) for example. Theorem 3.3's justification for it is as follows: $e_2 \leq_{\mathsf{HB}}^{\sigma_3} e_5 = \mathsf{lastWr}_{\sigma_3}(e_7) \leq_{\mathsf{TO}}^{\sigma_3} e_8 = \mathsf{pred}_{\sigma_3}(e_9)$. But, let us unravel the reasoning behind why neither (e_2, e_9) nor (e_5, e_9) are data races. Consider an arbitary correct reordering σ' of σ_3 that respects $\leq_{\mathsf{HB}}^{\sigma_3}$ and contains e_9. Since e_8 is also an event of t_4, $e_8 \in \mathsf{Events}_{\sigma'}$. In addition, $e_7 \in \mathsf{Events}_{\sigma'}$ as $e_7 \leq_{\mathsf{TO}}^{\sigma_3} e_8$. Now, since $e_5 = \mathsf{lastWr}_{\sigma_3}(e_7)$, e_5 is before e_7 in σ' and since $e_2 \leq_{\mathsf{HB}}^{\sigma_3} e_5$, e_2 must also be before e_7. Therefore, e_7 and e_8 will be between e_2 and e_9 and between e_5 and e_9. Similar reasoning can be used to conclude that the other pairs are not $\leq_{\mathsf{HB}}^{\sigma_3}$-schedulable as well.

Lastly, consider trace σ_4 shown in Fig. 4. In this case, $\leq_{\mathsf{SHB}}^{\sigma_4}=\leq_{\mathsf{tr}}^{\sigma_4}$. All conflicting memory accesses are in HB-race. While HB correctly identifies the first race (e_2, e_3) as valid, there are 3 HB-races that are not real data races — (e_2, e_5), (e_9, e_{12}), and (e_4, e_{11}). (e_2, e_5) is not valid because any correct reordering of σ_4 must have e_2 before e_3 and e_3 before e_5. This is also captured by SHB reasoning because $e_2 \leq_{\mathsf{SHB}}^{\sigma_4} e_3 \leq_{\mathsf{TO}}^{\sigma_4} e_4 = \mathsf{pred}_{\sigma_4}(e_5)$. A similar reasoning shows that (e_9, e_{12}) is not valid. The interesting case is that of (e_4, e_{11}). Here, in any correct reordering σ' of σ_4, the following must be true: (a) if $e_4 \in \mathsf{Events}_{\sigma'}$ then $e_1 \in \mathsf{Events}_{\sigma'}$; (b) if $e_{11} \in \mathsf{Events}_{\sigma'}$ then $e_8 \in \mathsf{Events}_{\sigma'}$; (c) if $\{e_1, e_4, e_7\} \subseteq \mathsf{Events}_{\sigma'}$ then $e_1 \leq_{\mathsf{tr}}^{\sigma'} e_4 \leq_{\mathsf{tr}}^{\sigma'} e_7$; and (d) if $\{e_8, e_{11}, e_{14}\} \subseteq \mathsf{Events}_{\sigma'}$ then $e_8 \leq_{\mathsf{tr}}^{\sigma'} e_{11} \leq_{\mathsf{tr}}^{\sigma'} e_{14}$. Therefore, any correct reordering σ' of σ_4 containing both e_4 and e_{11} contains e_1 and e_8 (because of (a) and (b)) and must contain at least one of e_7 or e_{14} to ensure that critical sections of ℓ don't overlap. Then in σ', e_4 and e_{11} cannot be consecutive because either e_7 or e_{14} will appear between them (properties (c) and (d)). This is captured using SHB and Theorem 3.3 by the fact that $e_4 \leq_{\mathsf{SHB}}^{\sigma_3} e_7 \leq_{\mathsf{SHB}}^{\sigma_3} e_{10} = \mathsf{pred}_{\sigma_4}(e_{11})$.

We conclude this section by observing that the soundness guarantees of HB (Theorem 2.7) follows from Theorem 3.3. Consider a trace σ whose first HB-race is (e_1, e_2). We claim that (e_1, e_2) is a $\leq^{\sigma}_{\mathsf{HB}}$-schedulable race. Suppose (for contradiction) it is not. Then by Theorem 3.3, $e = \mathsf{pred}_{\sigma}(e_2)$ is defined and $e_1 \leq^{\sigma}_{\mathsf{SHB}} e$. Now observe that we must have $\neg(e_1 \leq^{\sigma}_{\mathsf{HB}} e)$ (or otherwise $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$, contradicting our assumption that (e_1, e_2) is an HB-race). Then, by the definition of $\leq^{\sigma}_{\mathsf{SHB}}$ (Definition 3.2), there are two events e_3 and e_4 (possibly same as e_1 and e) such that $e_1 \leq^{\sigma}_{\mathsf{SHB}} e_3$, $e_3 = \mathsf{lastWr}_{\sigma}(e_4)$, $e_4 \leq^{\sigma}_{\mathsf{SHB}} e$, and $\neg(e_3 \leq^{\sigma}_{\mathsf{HB}} e_4)$. Then (e_3, e_4) is an HB-race, and it contradicts the assumption that (e_1, e_2) is the first HB-race.

The above argument that Theorem 2.7 follows from Theorem 3.3, establishes that our SHB-based analysis using Theorem 3.3 does not miss the race detected by a *sound* HB-based race detection algorithm.

4 ALGORITHM FOR DETECTING $\leq_{\mathsf{HB}}$-SCHEDULABLE RACES

We will discuss our algorithm for detecting races identified by the $\leq_{\mathsf{SHB}}$ partial order. The algorithm is based on efficient, vector clock based computation of the $\leq_{\mathsf{SHB}}$-partial order. It is similar to the standard DJIT$^+$ algorithm [Pozniansky and Schuster 2003] to detect HB-races. We will first briefly discuss vector clocks and associated notations. Then, we will discuss a one-pass streaming vector clock algorithm to compute $\leq_{\mathsf{SHB}}$ for detecting $\leq_{\mathsf{HB}}$-schedulable races. Finally, we will discuss how epoch optimizations, similar to FASTTRACK [Flanagan and Freund 2009] can be readily applied in our setting to enhance performance of the proposed vector clock algorithm.

4.1 Vector Clocks and Times

A vector *time* or a vector *timestamp* $V : \mathsf{Threads}_{\sigma} \to \mathit{Nat}$ maps each thread in a trace σ to a natural number. Vector times support comparison operation $\sqsubseteq$ for point-wise comparison, join operation $\sqcup$ for point-wise maximum, and update operation $V[n/t]$ which assigns the time $n \in \mathit{Nat}$ to the component $t \in \mathsf{Threads}_{\sigma}$ in the vector time V. Vector time $\bot$ maps all threads to 0. Formally,

$$
\begin{array}{rcll}
V_1 \sqsubseteq V_2 & \text{iff} & \forall t : V_1(t) \leq V_2(t) & \text{(Point-wise Comparison)} \\
V_1 \sqcup V_2 & = & \lambda t : \mathit{max}(V_1(t), V_2(t)) & \text{(Join)} \\
V[n/u] & = & \lambda t : \texttt{if } (t = u) \texttt{ then } n \texttt{ else } V(t) & \text{(Update)} \\
\bot & = & \lambda t : 0 & \text{(Bottom)}
\end{array}
$$

Vector *clocks* are place holders for vector timestamps, or variables whose domain is the space of vector times. All the above operations, therefore, also apply to vector clocks. The algorithms described next maintain a state comprising of several vector clocks, whose values, at specific instants, will be used to assign timestamps to events. We will use double struck font ($\mathbb{C}$, $\mathbb{L}$, $\mathbb{R}$, etc.,) for vector clocks and normal font (C, R, etc.,) for vector times.

4.2 Vector Clock Algorithm for Detecting Schedulable Races

Algorithm 1 depicts the vector clock algorithm for detecting $\leq_{\mathsf{HB}}$-schedulable races using the $\leq_{\mathsf{SHB}}$ partial order. Similar to the vector clock algorithm for detecting HB races, Algorithm 1 maintains a state comprising of several vector clocks. The idea behind Algorithm 1 is to use these vector clocks to *assign a vector timestamp* to each event e (denoted by C_e) such that the ordering relation on the assigned timestamps ($\sqsubseteq$) enables determining the partial order $\leq_{\mathsf{SHB}}$ on events. This is formalized in Theorem 4.1. The algorithm runs in a streaming fashion and processes each event in the order in which it occurs in the trace. Depending upon the type of the observed event, an appropriate handler is invoked. The formal parameter t in each of the handlers refers to the thread performing the event, and the parameters ℓ, x and u represent the lock being acquired or released, the memory location being accessed and the thread being forked or joined, respectively. The procedure `Initialization`

assigns the initial values to the vector clocks in the state. We next present details of different parts of the algorithm.

Algorithm 1: *Vector Clock for Checking $\leq_{\text{SHB}}$-schedulable races*

```
1  procedure Initialization
2      foreach t do C_t := ⊥[1/t] ;
3      foreach ℓ do L_ℓ := ⊥ ;
4      for x ∈ Vars do
5          LW_x := ⊥;
6          R_x := ⊥;
7          W_x := ⊥;
8  procedure acquire(t, ℓ)
9      C_t := C_t ⊔ L_ℓ ;
10 procedure release(t, ℓ)
11     L_ℓ := C_t ;
12     C_t(t) := C_t(t) + 1 ; (* next event *)
13 procedure fork(t, u)
14     C_u := C_t[1/u] ;
15     C_t(t) := C_t(t) + 1 ; (* next event *)
16 procedure join(t, u)
17     C_t := C_t ⊔ C_u ;
18 procedure read(t, x)
19     if ¬(W_x ⊑ C_t) then
20         declare 'race with write';
21     C_t := C_t ⊔ LW_x;
22     R_x(t) := C_t(t);
23 procedure write(t, x)
24     if ¬(R_x ⊑ C_t) then
25         declare 'race with read';
26     if ¬(W_x ⊑ C_t) then
27         declare 'race with write';
28     LW_x := C_t ;
29     W_x(t) := C_t(t);
30     C_t(t) := C_t(t) + 1 ; (* next event *)
```

4.2.1 Vector clocks in the State. The description of each of the vector clocks that are maintained in the state of Algorithm 1 is as follows:

(1) **Clocks** $\mathbb{C}_t$: For every thread t in the trace being analyzed, the algorithm maintains a vector clock $\mathbb{C}_t$. At any point during the algorithm, let us denote by e_t the last event performed by thread t in the trace so far. Then, the *timestamp* C_{e_t} of the event e_t can be obtained from the value of the clock $\mathbb{C}_t$ as follows. If e_t is a read, acquire or a join event, then $C_{e_t} = \mathbb{C}_t$, otherwise $C_{e_t} = \mathbb{C}_t[(c-1)/t]$, where $c = \mathbb{C}_t(t)$.

(2) **Clocks** $\mathbb{L}_\ell$: The algorithm maintains a vector clock $\mathbb{L}_\ell$ for every lock ℓ in the trace. At any point during the algorithm, the clock $\mathbb{L}_\ell$ stores the timestamp C_{e_ℓ}, where e_ℓ is the last event of the form $e_\ell = \langle \cdot, \texttt{rel}(\ell) \rangle$, in the trace seen so far.

(3) **Clocks** $\mathbb{LW}_x$: For every memory location x accessed in the trace, the algorithm maintains a clock $\mathbb{LW}_x$ ($\mathbb{L}$ast $\mathbb{W}$rite to x) to store the timestamp C_{e_x}, of the last event e_x of the form $\langle \cdot, \texttt{w}(x) \rangle$.

(4) **Clocks** $\mathbb{R}_x$ and $\mathbb{W}_x$: The clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ store the read and write *access histories* of each memory location x. At any point in the algorithm, the vector time R_x stored in the the $\mathbb{R}$ead access history clock $\mathbb{R}_x$ is such that $\forall t, R_x(t) = C_{e_t^{\texttt{r}(x)}}(t)$ where $e_t^{\texttt{r}(x)}$ is the last event of thread t that reads x in the trace seen so far. Similarly, the vector time W_x stored in the $\mathbb{W}$rite access history clock $\mathbb{W}_x$ is such that $\forall t, W_x(t) = C_{e_t^{\texttt{w}(x)}}(t)$ where $e_t^{\texttt{w}(x)}$ is the last event of thread t that writes to x in the trace seen so far.

The clocks $\mathbb{C}_t, \mathbb{L}_\ell, \mathbb{LW}_x$ are used to correctly compute the timestamps of the events, while the access history clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ are used to detect races.

4.2.2 Initialization and Clock Updates. For every thread t, the clock $\mathbb{C}_t$ is initialized to the vector time $\bot[1/t]$. Each of the clocks $\mathbb{L}_\ell$, $\mathbb{LW}_x$, $\mathbb{R}_x$ and $\mathbb{W}_x$ are initialized to $\bot$. This is in accordance with the semantics of these clocks presented in Section 4.2.1.

When processing an acquire event $e = \langle t, \mathtt{acq}(\ell)\rangle$, the algorithm reads the clock $\mathbb{L}_\ell$ and updates the clock $\mathbb{C}_t$ with $\mathbb{C}_t \sqcup \mathbb{L}_\ell$ (see Line 9). This ensures that the timestamp C_e (which is the value of the clock $\mathbb{C}_t$ after executing Line 9) is such that $C_{e'} \sqsubseteq C_e$ for every ℓ-release event $e' = \langle t', \mathtt{rel}(\ell)\rangle$ observed in the trace so far.

At a release event $e = \langle t, \mathtt{rel}(\ell)\rangle$, the algorithm writes the timestamp C_e of the current event e to the clock $\mathbb{L}_\ell$ (see Line 11). Notice that e is also the last release event of lock ℓ in the trace seen so far, and thus, this update correctly maintains the invariant stated in Section 4.2.1. This update ensures that any future events that acquire the lock ℓ can update their timestamps correctly. The algorithm then increments the local clock $\mathbb{C}_t(t)$ (Line 12). This ensures that if the next event e' in the thread t and the next acquire event f of lock ℓ satisfy $e' \not\leq_{\mathsf{SHB}} f$, then the timestamps of these events satisfy $C_{e'} \not\sqsubseteq C_f$. This is crucial for the correctness of the algorithm (Theorem 4.1).

The updates performed by the algorithm at a fork (resp. join) event are similar to the updates performed when observing a release (resp. acquire) event. The update at Line 14 is equivalent to the update $\mathbb{C}_u := \mathbb{C}_t \sqcup \mathbb{C}_u$ and ensures that the timestamp of each event $e' = \langle u, \cdot\rangle$ performed by the forked thread u satisfy $C_e \sqsubseteq C_{e'}$, where e is the current event forking the new thread u. Similarly, the update performed at Line 17 when processing the join event $e = \langle t, \mathtt{join}(u)\rangle$ ensures that the timestamp of each event $e' = \langle u, \cdot\rangle$ of the joined thread u is such that $C_{e'} \sqsubseteq C_e$.

At a read event $e = \langle t, \mathtt{r}(x)\rangle$, the clock $\mathbb{C}_t$ is updated with the join $\mathbb{C}_t \sqcup \mathbb{LW}_x$ (Line 21). Recall that $\mathbb{LW}_x$ stores the timestamp of the last event that writes to x (or in other words, the event $\mathsf{lastWr}(e)$) in the trace seen so far. This ensures that the timestamps C_e and $C_{\mathsf{lastWr}(e)}$ satisfy $C_{\mathsf{lastWr}(e)} \sqsubseteq C_e$. In addition, the algorithm also updates the component $\mathbb{R}_x(t)$ with the local component of the clock $\mathbb{C}_x$ (Line 22) in order to maintain the invariant described in Section 4.2.1.

At a write event $e = \langle t, \mathtt{w}(x)\rangle$, the algorithm updates the value of the last-write clock $\mathbb{LW}_x$ (Line 28) with the timestamp C_e stored in $\mathbb{C}_t$. The component $\mathbb{W}_x(t)$ is updated with the value of the local component $\mathbb{C}_t(t)$ to ensure the invariant described in Section 4.2.1 is maintained correctly. Finally, similar to the increment after a release event, the local clock is incremented in Line 30.

4.2.3 Checking for Races. At a read/write event e, the algorithm determines if there is a conflicting event e' in the trace seen so far such that (e', e) is an $\leq_{\mathsf{HB}}$-schedulable race. From Theorem 3.3 and Theorem 4.1, it follows that it is sufficient to check if $C_{e'} \not\sqsubseteq C_{\mathsf{pred}(e)}$. However, since the algorithm does not explicitly store the timestamps of events, we use the access histories $\mathbb{R}_x$ and $\mathbb{W}_x$ to check for races. Below we briefly describe these checks. The formal statement of correctness is presented in Theorem 4.2 and its proof is presented in Appendix B. We briefly outline the ideas here.

Recall that, for an event $e = \langle t, \cdot\rangle$ if $\mathsf{pred}(e)$ is undefined, the `Initialization` procedure ensures that $C_e = \bot[1/t]$. In this case, we have $V \not\sqsubseteq C_e$, for any vector-timestamp V with non-negative entries such that $V(t) = 0$, $\bot \sqsubseteq V$ and $V \neq \bot$. Algorithm 1 correctly reports a race in this case (see Lines 19-20, 24-27).

On the other hand, if $\mathsf{pred}(e)$ is defined, then the clock $\mathbb{C}_t$, at Line 19, 24 or 26, is either the timestamp $C_{\mathsf{pred}(e)}$ (if $\mathsf{pred}(e)$ was a read, join or an acquire event) or the timestamp $C_{\mathsf{pred}(e)}[(c+1)/t]$, where $c = C_{\mathsf{pred}(e)}(t)$ (if $\mathsf{pred}(e)$ was a write, fork or a release event). In either case, if the check $\mathbb{W}_x \sqsubseteq \mathbb{C}_t$ at Line 19 fails, then the read event e being processed is correctly declared to be in race with an earlier conflicting write event. Similarly, Algorithm 1 reports that a write event e is in race with an earlier read (resp. write) event based on whether the check on Line 24 (resp. Line 26) fails or not.

4.2.4 Correctness and Complexity. Here, we fix a trace σ. Recall that, for an event e, we say that C_e is the timestamp assigned by Algorithm 1 to event e. Theorem 4.1 asserts that the time stamps computed by Algorithm 1 can be used to determine the partial order $\leq^{\sigma}_{\mathsf{SHB}}$.

Theorem 4.1. *For events $e, e' \in \mathsf{Events}_\sigma$ such that $e \leq^{\sigma}_{\mathsf{tr}} e'$, $C_e \sqsubseteq C_{e'}$ iff $e \leq^{\sigma}_{\mathsf{SHB}} e'$*

Next, we state the correctness of the algorithm. We say that Algorithm 1 reports a race at an event e, if it executes lines 20, 25 or 27 while processing the handler corresponding to e.

Theorem 4.2. *Let e be a read/write event $e \in \mathsf{Events}_\sigma$. Algorithm 1 reports a race at e iff there is an event $e' \in \mathsf{Events}_\sigma$ such that (e', e) is an $\leq^{\sigma}_{\mathsf{HB}}$-schedulable race.*

The following theorem states that the asymptotic time and space requirements for Algorithm 1 are the same as that of the standard HB algorithm.

Theorem 4.3. *For a trace σ with n events, T threads, V variables, and L locks, Algorithm 1 runs in time $O(nT \log n)$ and uses $O((V + L + T)T \log n)$ space.*

The proofs of Theorem 4.1, Theorem 4.2 and Theorem 4.3 are presented in Appendix B.

4.2.5 Differences from the HB algorithm. While the spirit of Algorithm 1 is similar to standard HB vector clock algorithms (such as Djit$^+$ [Pozniansky and Schuster 2003]), it differs from them in the following ways. First, we maintain an additional vector clock $\mathbb{LW}_x$ to track the timestamp of the last event that writes to memory location x (line 28), and use this clock to correctly update $\mathbb{C}_t$ (line 21). This difference is a direct consequence of the additional ordering edges in the $\leq_{\mathsf{SHB}}$ partial order—every read event e is ordered after the event $\mathsf{lastWr}(e)$, unlike $\leq_{\mathsf{HB}}$. Second, the 'local' component of the clock $\mathbb{C}_t$ is also incremented after every write event (line 19), in addition to after a release or a fork event (in contrast with Djit$^+$). This is to ensure correctness in the following scenario. Let e, e' and e'' be events such that $e = \langle t, \mathsf{r}(x)\rangle \in \mathsf{Reads}$, $e' = \langle t', \mathsf{w}(x)\rangle = \mathsf{lastWr}(e)$ (t' may be different from t), and e'' is the next event after e' in the thread t'. Incrementing the local component of the clock $\mathbb{C}_{t'}$ ensures that the vector timestamps of e and e'' are ordered only when $e'' \leq_{\mathsf{SHB}} e$. Third, our algorithm remains sound even beyond the first race, in contrast to Djit$^+$, which can lead to false positives beyond the first race.

4.3 Epoch Optimization

The epoch optimization, popularized by FastTrack [Flanagan and Freund 2009] exploits the insight that '*the full generality of vector clocks is unnecessary in most cases*', and can result in significant performance enhancement, especially when the traces are predominated by read and write events.

An epoch is a pair of an integer c and a thread t, denoted by $c@t$. Intuitively, epoch $c@t$ can be treated as the vector time $\bot[c/t]$. Thus, in order to compare an epoch $c@t$ with vector time V, it suffices to compare the t-th component of V with c. That is,

$$c@t \sqsubseteq V \quad \text{iff} \quad c \leq V(t).$$

Therefore, comparison between epochs is less expensive than that between vector times — $O(1)$ as opposed to $O((|\mathsf{Threads}_\sigma|)$ for full vector times. To exploit this speedup, some vector clocks in the new algorithm will adaptively store either epochs or vector times.

Algorithm 2 applies the epoch optimization to Algorithm 1. Here, similar to the FastTrack algorithm, we allow clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ to be adaptive, while other clocks ($\mathbb{C}_t$, $\mathbb{L}_\ell$ and $\mathbb{LW}_x$) always store vector times. The optimization only applies to the `read` and `write` handlers and thus we omit the other handlers from the description as they are same as those described in Algorithm 1. We also omit the `Initialization` procedure which only differs in that the $\mathbb{R}_x$ and $\mathbb{W}_x$ are initialized to the epoch 0@0.

Algorithm 2: *Epoch Optimization for Algorithm 1*

```
1  procedure read(t, x)
2      if ¬(W_x ⊑ C_t) then
3          declare 'race with write';
4      C_t := C_t ⊔ LW_x;
5      if R_x is an epoch c@u then
6          if c ≤ C_t(u) then
7              R_x := C_t(t)@t;
8          else
9              R_x := ⊥[C_t(t)/t][c/u];
10     else
11         R_x(t) := C_t(t);

12 procedure write(t, x)
13     if ¬(R_x ⊑ C_t) then
14         declare 'race with read';
15     if (W_x ⊑ C_t) then
16         W_x := C_t(t)@t
17     else
18         declare 'race with write';
19         if W_x is an epoch c@u then
20             W_x := ⊥[C_t(t)/t][c/u];
21         else
22             W_x(t) := C_t(t);
23     LW_x := C_t ;
24     W_x(t) := C_t(t);
25     C_t(t) := C_t(t) + 1 ; (* next event *)
```

Depending upon how these clocks compare with the thread's clock $\mathbb{C}_t$, the clocks switch back and forth between epoch and vector time values:

- Initially, both $\mathbb{R}_x$ and $\mathbb{W}_x$ are assigned the epoch 0@0. The element 0@0 can be thought of as the analogue of $\bot$.
- The clock $\mathbb{W}_x$ is *fully adaptive* — it can switch back and forth between vector and epoch times depending upon how it compares with $\mathbb{C}_t$. Notice that, in the FastTrack algorithm proposed in [Flanagan and Freund 2009], the clock $\mathbb{W}_x$ is always an epoch. The underlying assumption for such a simplification is that all the events that write to a given memory location are totally ordered with respect to $\leq_{\mathsf{HB}}$. This assumption, however, need not hold beyond the first HB race. After the first race is encountered, two $\mathtt{w}(x)$ events e and e' may be unordered by both $\leq_{\mathsf{HB}}$ and $\leq_{\mathsf{SHB}}$. In Algorithm 2, $\mathbb{W}_x$ has an epoch representation if and only if the last write event e on x is such that $e' \leq_{\mathsf{SHB}} e$ for every event e' of the form $e' = \langle \cdot, \mathtt{w}(x) \rangle$ in the trace seen so far. When performing a write event $e = \langle t, \mathtt{w}(x) \rangle$, if $\mathbb{W}_x$ satisfies $\mathbb{W}_x \sqsubseteq \mathbb{C}_t$ (Line 15), then the event e is ordered after all previous $\mathtt{w}(x)$ events, and thus, in this case, $\mathbb{W}_x$ is converted to an epoch representation independent of its original representation (see Line 16). Otherwise, there are at least two $\mathtt{w}(x)$ events that are not ordered by $\leq_{\mathsf{SHB}}$ and thus $\mathbb{W}_x$ becomes a full-fledged vector clock (Lines 20 and 22).
- The clock $\mathbb{R}_x$ is only *semi-adaptive* — we do not switch back to epoch representation once the clock $\mathbb{R}_x$ takes up a vector-time value. The clock $\mathbb{R}_x$ is initialized to be an epoch. When processing a read event $e = \langle t, \mathtt{r}(x) \rangle$, if the algorithm determines that there is a read event $e' = \langle t', \mathtt{r}(x) \rangle$ observed earlier such that $e' \not\leq_{\mathsf{SHB}} e$, then the clock $\mathbb{R}_x$ takes a vector-time representation. After this point, $\mathbb{R}_x$ stays in the vector clock representation forever. The $\mathbb{R}_x$ clock is an epoch only if all the reads of x observed are ordered totally by $\leq_{\mathsf{SHB}}$. Thus, in order to determine if $\mathbb{R}_x$ can be converted back to an epoch representation, one needs to check if $(\mathbb{R}_x \sqsubseteq \mathbb{C}_t)$ every time a read event is processed. Since this is an expensive additional comparison and because most traces from real-world examples are dominated by

read events, we avoid such a check and force $\mathbb{R}_x$ to be only semi-adaptive. This is similar to the FastTrack algorithm.

As with FastTrack, the epoch optimization for $\leq_{\mathsf{SHB}}$ is sound and does not lead to any loss of precision — the optimized algorithm (Algorithm 2) declares a race at an event e iff the corresponding unoptimized algorithm (Algorithm 1) declares a race at e.

One must however note that the new clock $\mathbb{LW}_x$ does not have an adaptive representation, and is always required to be a vector clock. One can think of $\mathbb{LW}_x$ to be similar to, say, the clocks $\mathbb{L}_\ell$. These clocks are used to maintain the partial order, unlike the clocks $\mathbb{R}_x$ or $\mathbb{W}_x$ which are only used to check for races. Thus, one needs the full generality of vector times for $\mathbb{LW}_x$.

5 EXPERIMENTS

We first describe our implementation to detect $\leq_{\mathsf{HB}}$-schedulable races. We then present a brief description of the chosen benchmarks and finally the results of evaluating our implementation on these benchmarks.

5.1 Implementation

We have implemented our SHB-based race detection algorithms (Algorithm 1 and Algorithm 2) in our tool Rapid, which is publicly available at [Mathur 2018]. Rapid is written in Java and supports analysis on traces generated by the instrumentation and logging functionality provided by RVPredict [Rosu 2018] to generate traces from Java programs. The traces generated by RVPredict contain read, write, fork, join, acquire and release events. We assume that the traces are sequentially consistent, similar to the assumption made by [Huang et al. 2014]. We compare the performance of five dynamic race detection algorithms to demonstrate the effectiveness of SHB-based sound reasoning:

HB We implemented the Djit$^+$ algorithm for computing the $\leq_{\mathsf{HB}}$-partial order and detecting HB-races, in our tool Rapid. As with popular implementations of Djit$^+$, our implementation of Djit$^+$ discovers all $\leq_{\mathsf{HB}}$-unordered pairs of conflicting events. This serves as a base line to demonstrate how many false positives would result, if one considered all HB-races (instead of $\leq_{\mathsf{HB}}$-schedulable races). This algorithm is same as Algorithm 1 except that the lines involving the clock $\mathbb{LW}_x$ (Lines 5, 21 and 28) are absent.

SHB This is the implementation of Algorithm 1 in our tool Rapid. The soundness guarantee of Algorithm 1 (Theorem 4.2) ensures that our implementation reports only (and all) $\leq_{\mathsf{HB}}$-schedulable races and thus reports no false alarms. As pointed out in Section 3, $\leq_{\mathsf{SHB}}$ timestamps can be used for constructing a witness trace for a given $\leq_{\mathsf{HB}}$-schedulable race. We defer this functionality to future work.

FHB This is the algorithm that mimics a software developer's strategy when using HB-race detectors. This algorithm is a slight variant of the Djit$^+$ algorithm and is implemented in our tool Rapid. Every time an HB-race is discovered, the algorithm force orders the events in race, before analyzing subsequent events in the trace. When processing a read event $e = \langle t, \mathsf{r}(x)\rangle$, if the algorithm discovers a race (that is, if the check $\neg(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$ passes), the algorithm reports a race and also updates the clock $\mathbb{C}_t$ as $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{W}_x$. Similarly, at a write event, the algorithm updates the clock $\mathbb{C}_t$ as $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{R}_x$ (resp. $\mathbb{C}_t := \mathbb{C}_t \sqcup \mathbb{W}_x$) if the check $\neg(\mathbb{R}_x \sqsubseteq \mathbb{C}_t)$ (resp. $\neg(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$) passes. This algorithm is sound — all races reported by this algorithm are schedulable, but it may fail to identify some races that are schedulable. The complete description of FHB is presented in Algorithm 3.

WCP WCP or Weak Causal Precedence [Kini et al. 2017] is another sound partial order that can be employed for predictive data race detection. WCP is weaker than both, its precursor

Algorithm 3: *Vector Clock for FHB race detection*

```
1  procedure Initialization
2      foreach t do C_t := ⊥[1/t] ;
3      foreach ℓ do L_ℓ := ⊥ ;
4      for x ∈ Vars do
5          R_x := ⊥;
6          W_x := ⊥;
7  procedure acquire(t, ℓ)
8      C_t := C_t ⊔ L_ℓ ;
9  procedure release(t, ℓ)
10     L_ℓ := C_t ;
11     C_t(t) := C_t(t) + 1 ; (* next event *)
12 procedure fork(t, u)
13     C_u := C_t[1/u] ;
14     C_t(t) := C_t(t) + 1 ; (* next event *)
15 procedure join(t, u)
16     C_t := C_t ⊔ C_u ;
17 procedure read(t, x)
18     if ¬(W_x ⊑ C_t) then
19         declare 'race with write';
20         C_t := C_t ⊔ W_x; (* force ordering *)
21     C_t := C_t ⊔ LW_x;
22     R_x(t) := C_t(t);
23 procedure write(t, x)
24     if ¬(R_x ⊑ C_t) then
25         declare 'race with read';
26         C_t := C_t ⊔ R_x; (* force ordering *)
27     if ¬(W_x ⊑ C_t) then
28         declare 'race with write';
29         C_t := C_t ⊔ W_x; (* force ordering *)
30     W_x(t) := C_t(t);
```

CP [Smaragdakis et al. 2012], and HB. That is, whenever HB or CP detect the presence of a race in a trace, WCP will also do so, and in addition, there are traces when WCP can correctly detect the presence of a race when neither HB or CP can. Nevertheless, WCP (and CP) also suffer from the same drawback as HB — the soundness guarantee applies only to the first race. As a result, races beyond the first one, detected by WCP (or CP) may not be real races. WCP admits a linear time vector clock algorithm and is also implemented in RAPID.

RVPredict RVPredict's race detection technology relies on maximal causal models [Huang et al. 2014]. RVPredict is sound and does not report any false alarms. Besides, RVPredict, at least in theory, guarantees to detect more races than any other sound race prediction tool, and thus more races than Algorithm 1 theoretically. RVPredict encodes the problem of race detection as a logical formula and uses an SMT solver to check for races. RVPredict can analyze the traces generated using its logging functionality, and thus is a natural choice for comparison.

Besides the vector clock algorithms (HB, SHB, FHB, WCP) described above, we also implemented the epoch optimizations for HB and SHB in RAPID.

5.2 Benchmarks

We measure the performance of our algorithms against traces drawn from a wide variety of benchmark programs (Column 1 in Table 1) that have previously been used to measure the performance of other race detection tools [Smaragdakis et al. 2012; Huang et al. 2014; Kini et al. 2017]. The set of benchmarks have been derived from different suites. The examples airlinetickets to pingpong are small-sized, and belong to the IBM Contest benchmark suite [Farchi et al. 2003], with lines of code roughly varying from 40 to 0.5M. The benchmarks moldyn and raytracer are drawn from the Java Grande Forum benchmark suite [Smith and Bull 2001] and are medium-sized with about 3K lines of code. The third set of benchmarks correspond to real-world software applications and include

Apache FTPServer, W3C Jigsaw web server, Apache Derby, and others (xalan to eclipse) derived from the DaCaPo benchmark suite (version 9.12) [Blackburn et al. 2006].

In Table 1, we also describe the characteristics of the generated traces that we use for analyzing our algorithms. The number of threads range from 3-12, the number of lock objects can be as high as 8K. The distinct memory locations accessed (Column 5) in the traces can go as high as 10M. The traces generated are dominated by access events, with the majority of events being read events (compare Columns 6, 7 and 8).

5.3 Setup

Our experiments were conducted on an 8-core 2.6GHz 46-bit Intel Xeon(R) Linux machine, with HotSpot 1.8.0 64-Bit Server as the JVM and 50 GB heap space. Using RVPredict's logging functionality, we generated one trace per benchmark and analyzed it with the various race detection engines: HB, SHB, FHB, WCP and RVPredict.

Our evaluation is broadly designed to evaluate our approach based on the following aspects:

(1) **Reducing false positives:** Dynamic race detection tools based on Eraser style lockset based analysis [Savage et al. 1997] are known to scale better than those based on happens-before despite careful optimizations like the use of epochs [Flanagan and Freund 2009]. One of the main reasons for the popularity of HB-based race detection tools such as FastTrack [Flanagan and Freund 2009] and ThreadSanitizer [Serebryany and Iskhodzhanov 2009] is the ability to produce reliable results (no false positives). However, as pointed out in Section 1, HB based analysis can report false races beyond the first race discovered. The purpose of detecting $\leq_{\mathsf{HB}}$-schedulable races, instead of all HB-races, is to ensure that only correct races are reported. However, since our algorithm for detecting $\leq_{\mathsf{HB}}$-schedulable races tracks additional vector clocks (namely $\mathbb{LW}_x$ for every memory location x), we would like to demonstrate the importance of such an additional book-keeping for ensuring soundness of happens-before based reasoning.

(2) **Prediction power:** As described in Section 1, a naïve fix to the standard HB race detection algorithm is to employ the FHB algorithm — after a race is discovered at an event, order the event with all conflicting events observed before it. We would like to examine if the use of $\leq_{\mathsf{SHB}}$-based reasoning enhances prediction power by detecting more races than this naïve strategy. Further, we would like to evaluate if more powerful approaches like the use of SMT solvers in RVPredict give significantly more benefit as compared to our linear time streaming algorithm.

(3) **Scalability:** While Algorithm 1 runs in linear time, it tracks additional clocks ($\mathbb{LW}_x$ for every memory location x accessed in the trace) over the standard HB vector clock algorithm. Since this can potentially slow down analysis, we would like to evaluate the performance overhead due to this additional book-keeping.

(4) **Epoch optimization:** The standard epoch optimization popularized by FastTrack [Flanagan and Freund 2009] is designed to work for the case when all the writes to a memory location are totally ordered. While this is true until the first race is discovered, this condition may not be guaranteed after the first race. We will evaluate the effectiveness of our adaptation of this optimization to work beyond the first race.

5.4 Evaluation

Our experimental results are summarized in Table 1, Table 2 and Table 3. Table 1 describes information about generated execution logs. Table 2 depicts the number of races and warnings raised

[2] a thread is active if there is an event $e = \langle t, op \rangle$ performed by the thread t in the trace generated

Table 1. Benchmarks and metadata of the traces generated. Columns 1 and 2 describe the name and the lines of code in the source code of the chosen benchmarks. Column 3, 4 and 5 describe respectively the number of *active*[2] threads, locks, and memory locations in the traces generated by the corresponding program in Column 1. Column 6 reports the total number of events in the trace. Columns 7, 8, 9, 10 and 11 respectively denote the number of read, write, acquire (or release), fork and join events.

1	2	3	4	5	6	7	8	9	10	11
Program	LOC	Thrds	Locks	Vars	Events					
					Total	Read	Write	Synch.	Fork	Join
airlinetickets	83	4	0	44	137	77	48	0	12	0
array	36	3	2	30	47	8	30	3	3	0
bufwriter	199	6	1	471	22.2K	15.8K	3.6K	1.4K	6	4
bubblesort	274	12	2	167	4.6K	4.0K	404	121	27	0
critical	63	4	0	30	55	18	31	0	4	2
mergesort	298	5	3	621	3.0K	2.0K	914	55	5	3
pingpong	124	6	0	51	147	57	71	0	19	0
moldyn	2.9K	3	2	1.2K	200.0K	182.8K	17.2K	31	3	1
raytracer	2.9K	3	8	3.9K	15.8K	10.4K	5.3K	60	3	1
derby	302K	4	1112	185.6K	1.3M	879.5K	404.5K	31.2K	4	2
ftpserver	32K	11	301	5.5K	49.0K	30.0K	7.8K	5.6K	11	4
jigsaw	101K	11	275	103.5K	3.1M	2.6M	413.5K	5.9K	13	4
xalan	180K	6	2491	4.4M	122.0M	101.7M	18.3M	1M	7	5
lusearch	410K	7	118	5.2M	216.4M	162.1M	53.9M	206.6K	7	0
eclipse	560K	14	8263	10.6M	87.1M	72.6M	12.9M	765.4K	16	3
Total					430.3M	340.2M	85.9M	2.1M	140	29

by the different race detection algorithms. Columns 3-8 in Table 2 report the number of distinct pairs (pc_1, pc_2) of program locations corresponding to an identified data race. That is, for every *event* race pair (e_1, e_2) identified by the different race detection algorithms, we identify the pair of *program locations* that give rise to this event pair and report the total number of such program location pairs (counting the pairs (pc_1, pc_2) and (pc_2, pc_1) only once). Since each of the vector clock algorithms (HB, SHB, FHB and WCP) only report whether the event being processed is in race with some earlier event, we need to perform a separate analysis step using the vector timestamps, to determine the actual pair of events (and thus the corresponding pair of program locations) in race. In Columns 9, 10, 11 and 12 in Table 2 we report the number of *warnings* raised by the four vector clock algorithms—HB, SHB, FHB and WCP respectively. A warning is raised when at a read/write event e, we determine if the event e is in race with an earlier event, counting multiple warnings for a single event only once. In Table 3, Columns 2, 5, 8, 9, 10 and 11 respectively report the time taken by different analyses engines — HB, SHB, FHB, WCP and RVPredict— on the trace generated. We also measure the time taken by the epoch optimizations for both HB and SHB vector clock algorithms (Columns 4 and 7 respectively) and report the speedup thus obtained over the naïve vector clock algorithms (Columns 4 and 7 respectively). When analyzing the generated traces using WCP, we filter out events that are thread local; this does not affect any races. The memory requirement of a naïve vector clock algorithm for WCP, as described in [Kini et al. 2017] can be a bottleneck and removing thread local events allowed us to analyze the larger traces (xalan, lusearch and eclipse) without any memory blowup. We next discuss our results in detail.

5.4.1 Reducing False Positives. First, observe that both the number of races reported (Columns 3, 4 and 5 in Table 2) and the number of warnings raised (Columns 8, 9 and 10) by HB, SHB and FHB are monotonically decreasing, as expected — HB detects all $\leq_{\mathsf{HB}}$-schedulable races but additional false races, SHB detects exactly the set of $\leq_{\mathsf{HB}}$-schedulable races and FHB detects a subset of

Table 2. Number of races detected and warnings raised. Column 1 and 2 denote the benchmarks and the size of the traces generated. Columns 3, 4, 5 and 6 respectively report the number of distinct program location pairs for which there are pair of events in a race, as identified by HB, SHB, FHB and WCP. Columns 7 and 8 denote the races reported by RVPredict when run with the parameters (`window-size=1K`, `solver-timeout=60s`) and (`window-size=10K`, `solver-timeout=240s`). Columns 9, 10, 11 and 12 respectively denote the number of warnings generated when running the vector clock algorithms for detecting races using HB (unsound), SHB (sound and complete for $\leq_{\text{HB}}$-schedulable races), FHB (naive algorithm that forces an order after every race discovered) and WCP analyses.

1	2	3	4	5	6	7	8	9	10	11	12
		Races						Warnings			
Program	#Events	HB	SHB	FHB	WCP	RVPredict		HB	SHB	FHB	WCP
						1K/60s	10K/240s				
airlinetickets	137	6	6	3	6	6	6	8	8	5	8
array	47	0	0	0	0	0	0	0	0	0	0
bufwriter	22.2K	2	2	2	2	2	0	8	8	8	8
bubblesort	4.6K	6	6	6	6	6	0	602	269	100	612
critical	55	8	8	1	8	8	8	3	3	1	3
mergesort	3.0K	3	1	1	3	1	2	52	1	1	52
pingpong	147	3	3	3	3	3	3	11	8	8	11
moldyn	200.0K	44	2	2	44	2	2	24657	103	103	24657
raytracer	15.8K	3	3	3	3	2	3	118	8	8	118
derby	1.3M	26	13	11	26	12	-	89	29	28	89
ftpserver	49.0K	35	23	22	35	10	12	143	69	69	144
jigsaw	3.1M	8	4	4	10	4	2	14	4	4	17
xalan	122.0M	16	12	10	18	8	8	86	31	21	98
lusearch	216.4M	160	52	28	160	0	0	751002	232	119	751002
eclipse	87.1M	64	61	31	66	5	0	173	164	103	201
Total	430.3M	384	196	127	390	69	46	776966	937	578	777020

$\leq_{\text{HB}}$-schedulable races. Next, the number of races reported by HB can be way higher than the actual number of $\leq_{\text{HB}}$-schedulable races (see moldyn and lusearch). Similarly, the number of warnings raised can be an order of magnitude larger than those raised by either SHB or FHB. Clearly, many of these warnings are potentially spurious. Thus, an incorrect use of the popular HB algorithm can severely hamper developer productivity, and completely defies the point of using a sound race detection analysis technique. Further, in each of the benchmarks, both the set of races as well as the set of warnings reported by WCP were a superset of those reported by HB. This follows from the fact that WCP is a strictly weaker relation than HB.

While Theorem 3.3 guarantees that each of the additional race pairs reported by HB (over those reported by SHB) cannot be scheduled in any correct reordering of the observed trace that respects the induced $\leq_{\text{HB}}$ partial order, it does not guarantee that these extra races cannot be scheduled in *any* correct reordering. In order to see if the extra races reported by HB (Column 3 in Table 2) can be scheduled in a correct reordering that does not respect the $\leq_{\text{HB}}$ order induced by the observed trace, we manually inspected the traces (annotated with their vector timestamps) of mergesort, moldyn, derby, ftpserver, and jigsaw. In each of these benchmarks, we found that all the extra race pairs reported by HB indeed *cannot* be scheduled in *any* correct reordering (whether or not the correct reordering respects the induced $\leq_{\text{HB}}$ partial order). A common pattern that helped us conclude this observation has been depicted in Fig. 5a. Here, the trace writes to a memory location x in a thread t_1 (event e_1). Then, sometime later, another event e_2 performed by a different thread t_2 reads the value written by e_1. This is then followed by other events of thread t_2, not pertaining to memory location x. Finally, thread t_2 reads the memory location x again in event e_3. This pattern is

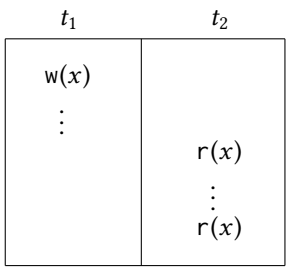
commonly observed when a thread reads a shared variable (here, this corresponds to the event e_2), takes a branch depending upon the value observed and then accesses the shared memory again within the branch. HB misses this dependency relation thus induced, and incorrectly reports that the pair (e_1, e_3) is in race. SHB, on the other hand, correctly orders $e_1 \leq_{\mathsf{SHB}} \mathsf{pred}(e_3)$, and does not report a race.

The two extra races reported by WCP but not by HB in jigsaw could not be confirmed to be false positives. Further, we did not inspect the extra races reported by HB or WCP (over SHB) in xalan, lusearch and eclipse owing to time constraints.

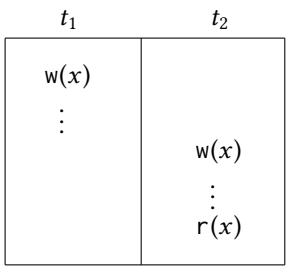
5.4.2 Prediction Power. The naïve algorithm FHB, while sound, can miss a lot of real races (Column 5 in Table 2) and has a poor prediction power as compared to the sound SHB algorithm. See for example, lusearch and eclipse where FHB misses almost half the races reported by SHB. Next, observe that while RVPredict, in theory, is *maximally sound*, it can miss a lot of races, sometimes even more than the naive FHB strategy (Columns 6 and 7 in Table 2). This is because RVPredict relies on SAT solving to determine data races. As a result, in order to scale to large traces obtained from real world software, RVPredict resorts to *windowing* — dividing the trace into smaller chunks and restricting its analysis to these smaller chunks. This strategy, while useful for scalability, can miss data races that are spread far across in the trace, yet can be captured using happens-before like analysis. Besides, since the underlying DPLL-based SAT solvers may not terminate within reasonable time, RVPredict sets a timeout for the solver — this means that even within a given window, RVPredict can miss races if the SAT solver does not return an answer within the set timeout. All these observations clearly indicate the power of $\leq_{\mathsf{SHB}}$-based reasoning.

Again, based on our manual inspection of program traces, we depict a common pattern found in Fig. 5b. Here, first, a thread t_1 writes to a shared variable x (event e_1). This is followed by another write to x in a different thread t_2 (event e_2). Finally, the next access to x is a read event e_3 performed by thread t_2. While FHB correctly reports the first write-write race (e_1, e_2), it fails to detect the write-read race (e_1, e_3) because of the artificial order imposed between e_1 and e_2. SHB, on the other hand, reports both (e_1, e_2) and (e_1, e_3) as $\leq_{\mathsf{HB}}$-schedulable races.

5.4.3 Scalability. First, the size of the traces, that SHB and the other three linear time vector clock algorithms can handle, can be really large, of the order of hundreds of millions (xalan, lusearch, etc.,). In contrast, RVPredict fails to scale for large traces, even after employing a windowing strategy. This is especially pronounced for the larger traces (bufwriter, derby-xalan). This suggests the power of using a linear time vector clock algorithm for dynamic race detection for real-world applications.



(a) Incorrect race reported by HB but not by SHB



(b) Correct race missed by FHB but detected by SHB

Fig. 5. Common race patterns found in the benchmarks

Table 3. Time taken by different race detection algorithms on traces generated by the corresponding programs in Column 1. Column 2 denotes the taken for analyzing the entire trace with the DJIT$^+$ vector-clock algorithm. Column 3 denotes the time taken by FASTTRACK-style optimization over the basic DJIT$^+$ and Column 4 denotes the speedup thus obtained. Column 5 denotes the times for the vector clock implementation of Algorithm 1. Column 6 and 7 denote the time and speedup due to the epoch optimization for SHB (Algorithm 2). A '-' in Columns 5 and 8 denote a downgraded performance due to epoch optimization. Column 8 denotes the time to analyze the traces using FHB (forcing an order in HB) analysis. Column 9 reports the time to analyze the traces using WCP partial order. The analysis in Column 9 is performed by filtering out thread local events and includes the time for this filtering. Column 10 and 11 respectively denote the time taken by RVPredict using the parameters (`window-size=1K`, `solver-timeout=60s`) and (`window-size=10K`, `solver-timeout=240s`). A '-' in Column 11 denotes that RVPredict did not finish within the set time limit of 4 hours.

1	2	3	4	5	6	7	8	9	10	11
	HB			SHB			FHB	WCP	RVPredict	
Program	VC (s)	Epoch (s)	Speed-up	VC (s)	Epoch (s)	Speed-up	(s)	(s)	1K/60s (s)	10K/240s (s)
airlinetickets	1.11	1.4	-	2.08	0.25	8.32x	2.33	0.35	1.34	1.33
array	1.13	1.66	-	0.55	0.25	2.2x	1.98	0.77	1.28	1.26
bufwriter	0.54	1.63	-	0.56	0.46	1.22x	0.96	1.3	3.61	1879
bubblesort	0.47	0.29	1.62x	0.78	0.34	2.3x	0.51	0.63	2.46	652
critical	0.68	0.24	2.8x	0.28	0.56	-	0.54	0.94	2.14	0.52
mergesort	0.4	1.58	-	0.43	0.35	1.23x	0.7	0.8	0.67	0.81
pingpong	0.31	0.31	1x	2.15	0.33	6.5x	0.41	0.99	2.3	0.57
moldyn	1.45	2.78	-	1.04	1.49	-	1.77	1.81	2.27	2.94
raytracer	0.95	2.25	-	1.82	0.41	4.44x	2.33	0.77	0.61	8.61
derby	4.27	2.91	1.46x	4.39	3.05	1.44x	5.56	9.24	72	-
ftpserver	0.84	0.84	1x	0.87	1.33	-	2.94	1.64	1.23	164
jigsaw	23	8.1	2.84x	10.04	7.35	1.37x	9.31	11.11	1.66	245
xalan	217	152	1.42x	222	184	1.21x	291	290	44	420
lusearch	362	383	-	444	337	1.32x	325	341	48	47
eclipse	525	200	2.63x	238	188	1.27x	168	512	25	951

The small and medium sized examples almost always finish with a few seconds for each of HB, SHB, FHB and WCP. The larger examples xalan, lusearch and eclipse can take as much as 4-10 minutes for the vector clock algorithms and 6-15 minutes for RVPredict's analysis with a window size of 10K and a solver timeout of 4 minutes.

5.4.4 Epoch Optimization. The epoch optimization is indeed effective in improving the performance of vector-clock algorithms even when all the write events to a memory location may not be totally ordered. The speedups vary from 1.2x to 8.3x on small and medium sized benchmarks and from 1.2x to 2.9x on larger traces. The speedup obtained for HB race detection is, in general, less than in the case of SHB algorithm. This is expected since SHB is strictly stronger than HB — every pair of events ordered by $\leq_{\mathsf{HB}}$ is also ordered by $\leq_{\mathsf{SHB}}$, and not every pair of events ordered by $\leq_{\mathsf{SHB}}$ may be ordered by $\leq_{\mathsf{HB}}$. As a result, a write event can get ordered after all previous write events more frequently when using $\leq_{\mathsf{SHB}}$ than when using $\leq_{\mathsf{HB}}$. This means that in the epoch optimization for SHB, the $\mathbb{W}_x$ clocks take up epoch representation more frequently than in the HB algorithm, and this difference is reflected in Columns 5 and 8 of Table 3.

6 RELATED WORK

The notion of correct reorderings to characterize causality in executions has been derived from [Smaragdakis et al. 2012; Kini et al. 2017]. In [Huang et al. 2014] a similar notion, called *feasible traces*

encompasses a more general causality model based on control flow information. Weak happens-before [Sen et al. 2005], Mazurkiewicz equivalence [Mazurkiewicz 1987; Abdulla et al. 2014] and observation equivalence [Chalupa et al. 2017] are other models that attempt to characterize causality. Many of these models also incorporate the notion of last-write causality, similar to SHB. However, these algorithms use expensive search algorithms like SAT solving to explore the space of correct reorderings, unlike a linear time vector clock algorithm like that for SHB. Our experimental evaluation concurs with this observation. Similar dependency relation called *reads-from* is also used to characterize weak memory consistency semantics [Alglave et al. 2014; Huang and Huang 2016].

Race detection techniques can be broadly classified as either being static or dynamic. Static race detection [Naik et al. 2006; Pratikakis et al. 2011; Radoi and Dig 2013; Engler and Ashcraft 2003; Voung et al. 2007; Musuvathi et al. 2008; Yahav 2001; Zhan and Huang 2016] is the problem of detecting if a program has an execution that exhibits a data race, by analyzing its source code. This problem, in its full generality, is undecidable and practical tools employing static analysis techniques often face a trade-off between scalability and precision. Further, the use of such techniques often require the programmer to add annotations to help guide static race detectors.

Dynamic race detection techniques, on the other hand, examine a single execution of the program to discover a data race in the program. A large number of tools employing dynamic analysis are based on lockset-like analysis proposed by Eraser [Savage et al. 1997]. Here, one tracks, for each memory location accessed, the set of locks that protect the memory location on each access. If this lockset becomes empty during the program execution, a warning is issued. Lockset-based analysis suffers from false positives. Other dynamic race detectors employ happens-before [Lamport 1978] based analysis. These include the use of vector clock [Mattern 1988; Fidge 1991] algorithms such as Djit$^+$ [Pozniansky and Schuster 2003] and FastTrack [Flanagan and Freund 2009] and the use of sets of threads and locks, as in, GoldiLocks [Elmas et al. 2007]. As demonstrated in this paper, happens-before based analysis is sound only if limited to detecting the first race. Other techniques can be categorized as *predictive* and can detect races missed by HB by exploring more correct reorderings of an observed trace. These include use of SMT solvers [Said et al. 2011; Huang et al. 2014; Liu et al. 2016; Huang and Rajagopalan 2016] and other techniques based on weakening the HB partial order including CP [Smaragdakis et al. 2012] and WCP [Kini et al. 2017]. Amongst these, WCP is the only technique that has a linear running time and is known to scale to large traces. The soundness guarantee of partial order based techniques, like WCP and CP, is again, limited to the first race. Nevertheless, they do detect subtle races that HB can miss. Our approach complements this line of research. Other dynamic techniques such as random testing [Sen 2008], sampling [Marino et al. 2009; Erickson et al. 2010], and hybrid race detection [O'Callahan and Choi 2003] are based on both locksets and happens-before relation.

7 CONCLUSION

Happens-before is a powerful technique that can be used to effectively detect for races. However, the detection power of HB is limited only until the first race is identified. We characterize when an HB-race, beyond just the first race, can be scheduled in an alternate reordering, by introducing a new partial order called SHB which identifies all HB-schedulable races. SHB can be implemented in a vector clock algorithm, which is only slightly different from HB vector clock algorithm, and thus, existing race detection tools can easily incorporate it to enhance their race detection capability. Also, standard epoch like optimizations can be employed to improve the performance of the basic algorithm. We show, through extensive experimental evaluation, the value our approach adds to sound race detection tools.

In the future, we would like to extend the work for weaker partial orders like CP [Smaragdakis et al. 2012] and WCP [Kini et al. 2017]. Another promising direction is to further enhance the prediction power by incorporating data values, control flow and data flow information in the traces.

A PROOF OF THEOREM 3.3

In this section, we prove Theorem 3.3. We begin with a couple of technical lemmas.

Lemma A.1. *Let σ be a trace and σ' be a correct reordering of σ that respects $\leq^{\sigma}_{\mathsf{HB}}$. For any e, e' such that $e \leq^{\sigma}_{\mathsf{SHB}} e'$, if $e' \in \mathsf{Events}_{\sigma'}$ and e' is not the last read event of its thread in σ', then $e \in \mathsf{Events}_{\sigma'}$ and $e \leq^{\sigma'}_{\mathsf{tr}} e'$.*

Proof. Consider any e, e' such that $e \leq^{\sigma}_{\mathsf{SHB}} e'$, $e' \in \mathsf{Events}_{\sigma'}$ and $e' = \langle t, op\rangle$ is not the last read event of the thread t in the trace σ'. Then it follows from Definition 3.2 that there is a sequence $e = e_0, e_1, \ldots e_n = e'$ such that for every $i \leq n-1$, $e_i \leq^{\sigma}_{\mathsf{tr}} e_{i+1}$ and either (a) $e_i \leq^{\sigma}_{\mathsf{TO}} e_{i+1}$ or (b) $e_i = \langle t_i, \mathtt{rel}(\ell)\rangle$, $e_{i+1} = \langle t_{i+1}, \mathtt{acq}(\ell)\rangle$, or (c) $e_{i+1} \in \mathsf{Reads}_{\sigma}$ and $e_i = \mathsf{lastWr}_{\sigma}(e_{i+1})$.

We will prove by induction on i, starting from $i = n$, that $e_i \in \mathsf{Events}_{\sigma'}$ and e_i is not the last read event of its thread in σ'. Observe that these properties hold for $e' = e_n - e_n \in \mathsf{Events}_{\sigma'}$ and e_n is not the last read event of its thread in σ'. Assume we have established the claim for e_{i+1}. Now there are three cases to consider for e_i. If $e_i \leq^{\sigma'}_{\mathsf{TO}} e_{i+1}$ then clearly $e_i \in \mathsf{Events}_{\sigma'}$ because $e_{i+1} \in \mathsf{Events}_{\sigma'}$. Further, if e_i is a read event, then it is not the last event of its thread because e_{i+1} is after it. If $e_i = \langle t_i, \mathtt{rel}(\ell)\rangle$ and $e_{i+1} = \langle t_{i+1}, \mathtt{acq}(\ell)\rangle$ then $e_i \in \mathsf{Events}_{\sigma'}$ because σ' respects $\leq^{\sigma}_{\mathsf{HB}}$. Further e_i is not the last read event because it is not a read event! The last case to consider is where $e_i = \mathsf{lastWr}_{\sigma}(e_{i+1})$. In this case, by induction hypothesis, we know that e_{i+1} is not the last read event of its thread, and therefore by properties of a correct reordering, we have $e_i \in \mathsf{Events}_{\sigma'}$. Notice that in this case e_i is not a read event, and so the claim holds. Thus, we have established that $e = e_0 \in \mathsf{Events}_{\sigma'}$.

Next, we show that for every $i \leq n-1$, $e_i \leq^{\sigma'}_{\mathsf{tr}} e_{i+1}$. If $e_i \leq^{\sigma}_{\mathsf{TO}} e_{i+1}$ or $e_i = \langle t_i, \mathtt{rel}(\ell)\rangle$ and $e_{i+1} = \langle t_{i+1}, \mathtt{acq}(\ell)\rangle$ with $e_i \leq^{\sigma}_{\mathsf{tr}} e_{i+1}$ then $e_i \leq^{\sigma'}_{\mathsf{tr}} e_{i+1}$ because σ' respects $\leq^{\sigma}_{\mathsf{HB}}$. On the other hand, if $e_i = \mathsf{lastWr}_{\sigma}(e_{i+1})$ then because σ' is a correct reordering of σ and e_{i+1} is not the last read event of its thread (established in the previous paragraph), we have $e_i = \mathsf{lastWr}_{\sigma}(e_{i+1}) = \mathsf{lastWr}_{\sigma'}(e_{i+1})$. This establishes the fact that $e = e_0 \leq^{\sigma'}_{\mathsf{tr}} e_n = e'$, which completes the proof of the lemma. □

A slightly weaker form of the converse of Lemma A.1 also holds.

Lemma A.2. *For a trace σ, let σ' be a trace with $\mathsf{Events}_{\sigma'} \subseteq \mathsf{Events}_{\sigma}$ such that (a) σ' is $\leq^{\sigma}_{\mathsf{SHB}}$ downward closed, i.e., for any $e, e' \in \mathsf{Events}_{\sigma}$ if $e \leq^{\sigma}_{\mathsf{SHB}} e'$ and $e' \in \mathsf{Events}_{\sigma'}$ then $e \in \mathsf{Events}_{\sigma'}$, and (b) $\leq^{\sigma'}_{\mathsf{tr}} = \leq^{\sigma}_{\mathsf{tr}} \cap (\mathsf{Events}_{\sigma'} \times \mathsf{Events}_{\sigma'})$. Then σ' is a correct reordering of σ that respects $\leq^{\sigma}_{\mathsf{HB}}$. Further, for **every** read event $e \in \mathsf{Reads}_{\sigma'}$, we have $\mathsf{lastWr}_{\sigma'}(e) \simeq \mathsf{lastWr}_{\sigma}(e)$, i.e., either both $\mathsf{lastWr}_{\sigma'}(e)$ and $\mathsf{lastWr}_{\sigma}(e)$ are undefined, or they are both defined and equal.*

Proof. The trace σ' in the lemma is such that the events in σ' are downward closed with respect to $\leq^{\sigma}_{\mathsf{SHB}}$ and in σ' they are ordered in exactly the same way as in σ. The fact that σ' respects $\leq^{\sigma}_{\mathsf{HB}}$ simply follows from the fact that $\leq^{\sigma}_{\mathsf{HB}} \subseteq \leq^{\sigma}_{\mathsf{SHB}}$ and $\leq^{\sigma}_{\mathsf{HB}} \subseteq \leq^{\sigma}_{\mathsf{tr}}$. So the main goal is to establish that σ' is a correct reordering of σ that preserves the last writes of *all* read events.

First we show that σ' respects lock semantics. Suppose $e_1 = \langle t_1, \mathtt{acq}(\ell)\rangle$ and $e_2 = \langle t_2, \mathtt{acq}(\ell)\rangle$ are two lock acquire events for some lock ℓ such that $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$ and $\{e_1, e_2\} \subseteq \mathsf{Events}_{\sigma'}$. Let e_1' be the matching release event for e_1 in σ; such an e_1' exists because σ is a valid trace. Then we have $e_1 \leq^{\sigma}_{\mathsf{HB}} e_1' \leq^{\sigma}_{\mathsf{HB}} e_2$, and so $e_1' \in \mathsf{Events}_{\sigma'}$ and $e_1' \leq^{\sigma'}_{\mathsf{tr}} e_2$ because σ' respects $\leq^{\sigma}_{\mathsf{HB}}$.

Next observe that since $\leq^{\sigma}_{\mathsf{TO}} \subseteq \leq^{\sigma}_{\mathsf{HB}}$ and σ' respects $\leq^{\sigma}_{\mathsf{HB}}$, we can conclude that $\sigma'|_t$ is a prefix of $\sigma|_t$ for any thread t.

Finally, consider any $e' \in \mathsf{Reads}_{\sigma'}$. Suppose $\mathsf{lastWr}_\sigma(e')$ is defined. Let $e = \mathsf{lastWr}_\sigma(e')$. Since $e \leq^\sigma_{\mathsf{SHB}} e'$ and σ' is downward closed with respect to $\leq^\sigma_{\mathsf{SHB}}$, we have $e \in \mathsf{Events}_{\sigma'}$. Let $e_1 = \mathsf{lastWr}_{\sigma'}(e')$. We need to argue that $e_1 = e$. Suppose (for contradiction) it is not, i.e., $e \neq e_1$. Then either $e_1 \leq^\sigma_{\mathsf{tr}} e$ or $e' \leq^\sigma_{\mathsf{tr}} e_1$, because $e = \mathsf{lastWr}_\sigma(e')$. However, the fact that $e_1 = \mathsf{lastWr}_{\sigma'}(e')$ contradicts the fact that $\leq^{\sigma'}_{\mathsf{tr}} = \leq^\sigma_{\mathsf{tr}} \cap (\mathsf{Events}_{\sigma'} \times \mathsf{Events}_{\sigma'})$. Conversely, if $\mathsf{lastWr}_{\sigma'}(e')$ is defined then let $e = \mathsf{lastWr}_{\sigma'}(e')$. Since $\leq^{\sigma'}_{\mathsf{tr}} = \leq^\sigma_{\mathsf{tr}} \cap (\mathsf{Events}_{\sigma'} \times \mathsf{Events}_{\sigma'})$, we have $e \leq^\sigma_{\mathsf{tr}} e'$. Thus, $\mathsf{lastWr}_\sigma(e')$ is defined. Let $e_1 = \mathsf{lastWr}_\sigma(e')$. Once again, since $e_1 \leq^\sigma_{\mathsf{SHB}} e'$, and σ' is downward closed with respect to $\leq^\sigma_{\mathsf{SHB}}$, we have $e_1 \in \sigma'$. Just like in the previous direction, we can conclude that $e = e_1$ because otherwise we violate the fact that $\leq^{\sigma'}_{\mathsf{tr}}$ is identical to $\leq^\sigma_{\mathsf{tr}}$ over $\mathsf{Events}_{\sigma'}$. □

We now prove Theorem 3.3 below

Theorem 3.3. *Let σ be a trace and $e_1 \leq^\sigma_{\mathsf{tr}} e_2$ be conflicting events in σ. (e_1, e_2) is an $\leq^\sigma_{\mathsf{HB}}$-schedulable race iff either $\mathsf{pred}_\sigma(e_2)$ is undefined, or $e_1 \not\leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$.*

Proof. Let us first prove the forward direction. That is, let (e_1, e_2) be an HB-race such that the event $e = \mathsf{pred}_\sigma(e_2)$ is defined and $e_1 \leq^\sigma_{\mathsf{SHB}} e$. Consider any correct reordering σ' that contains both e_1 and e_2 and respects $\leq^\sigma_{\mathsf{HB}}$. First, since σ' is a correct reordering of σ, we must have $e \in \mathsf{Events}_{\sigma'}$ and $e \leq^{\sigma'}_{\mathsf{tr}} e_2$. Further, since $e_1 \leq^\sigma_{\mathsf{SHB}} e$, from Lemma A.1, $e_1 \leq^{\sigma'}_{\mathsf{tr}} e$. Thus, we have that $e_1 \leq^{\sigma'}_{\mathsf{tr}} e \leq^{\sigma'}_{\mathsf{tr}} e_1$ for any correct reordering σ' of σ that respects $\leq^\sigma_{\mathsf{HB}}$. This means, (e_1, e_2) cannot be a $\leq^\sigma_{\mathsf{HB}}$-schedulable race.

We now prove the backward direction. Consider an HB-race (e_1, e_2) such that either $\mathsf{pred}_\sigma(e_2)$ is undefined, or if it exists, then it satisfies $e_1 \not\leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$. Consider the set $S^\sigma_{(e_1,e_2)}$ defined as

$$S^\sigma_{(e_1,e_2)} = \{e \in \mathsf{Events}_\sigma \setminus \{e_1, e_2\} \mid e \leq^\sigma_{\mathsf{SHB}} e_1 \text{ or } e \leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)\}$$

where we assume that if $\mathsf{pred}_\sigma(e_2)$ is undefined then no event e satisfies the condition $e \leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$.

First we will show that $S^\sigma_{(e_1,e_2)}$ is downward closed with respect to $\leq^\sigma_{\mathsf{SHB}}$. Consider e, e' such that $e \leq^\sigma_{\mathsf{SHB}} e'$ and $e' \in S^\sigma_{(e_1,e_2)}$. By definition of $S^\sigma_{(e_1,e_2)}$, we have $e' \notin \{e_1, e_2\}$ and either $e' \leq^\sigma_{\mathsf{SHB}} e_1$ or $e' \leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$. Observe that if $e \notin \{e_1, e_2\}$, then it is clear that $e \in S^\sigma_{(e_1,e_2)}$ by definition since $\leq^\sigma_{\mathsf{SHB}}$ is transitive. It is easy to see that $e \neq e_2$ — this is because since $e' \neq e_2$, and $\leq^\sigma_{\mathsf{SHB}} \subseteq \leq^\sigma_{\mathsf{tr}}$, $e' <^\sigma_{\mathsf{tr}} e_2$ and so $e <^\sigma_{\mathsf{tr}} e_2$. So, all we have left to establish is that $e \neq e_1$. Suppose for contradiction $e = e_1$. Then it must be the case that $e' \leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$. Since $e_1 = e \leq^\sigma_{\mathsf{SHB}} e' \leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$, we have $e_1 \leq^\sigma_{\mathsf{SHB}} \mathsf{pred}_\sigma(e_2)$, which contradicts our assumption about (e_1, e_2).

Let us now consider a trace σ'' which consists of the events in $S^\sigma_{(e_1,e_2)}$ ordered according to $\leq^\sigma_{\mathsf{tr}}$. That is, $\leq^{\sigma''}_{\mathsf{tr}} = \leq^\sigma_{\mathsf{tr}} \cap (S^\sigma_{(e_1,e_2)} \times S^\sigma_{(e_1,e_2)})$. Since σ'' satisfies the conditions of Lemma A.2, we can conclude that σ'' is a correct reordering of σ that respects $\leq^\sigma_{\mathsf{HB}}$ and preserves the last-writes of *every* read event present.

Consider the trace $\sigma' = \sigma'' e_1 e_2$. First we prove that σ' respects $\leq^\sigma_{\mathsf{HB}}$. To do that, we first show that for any event $e \in \mathsf{Events}_\sigma$ such that $e \leq^\sigma_{\mathsf{HB}} e_1$ and $e \neq e_1$, or $e \leq^\sigma_{\mathsf{HB}} e_2$ and $e \neq e_2$, then $e \in S^\sigma_{(e_1,e_2)}$. If $e \leq^\sigma_{\mathsf{HB}} e_1$ then $e \leq^\sigma_{\mathsf{SHB}} e_1$ and so $e \in S^\sigma_{(e_1,e_2)}$. On the other hand, if $e \leq^\sigma_{\mathsf{HB}} e_2$ (and $e \neq e_2$), since (e_1, e_2) is an HB-race, we must have $e \neq e_1$ and $e \leq^\sigma_{\mathsf{HB}} \mathsf{pred}_\sigma(e_2)$. So $e \in S^\sigma_{(e_1,e_2)}$. Now the fact σ' respects $\leq^\sigma_{\mathsf{HB}}$ follows from the fact that σ'' respects $\leq^\sigma_{\mathsf{HB}}$ and the claim just proved.

We now prove that σ' is a correct reordering. Observe that since σ' respects $\leq^\sigma_{\mathsf{HB}}$, σ' is well formed (lock semantics is not violated) and preserves thread-wise prefixes ($\forall t, \sigma'|_t$ is a prefix of $\sigma|_t$). Further, σ'' is such that every read event in σ'' reads the same last write as in σ. Also, since e_1 and e_2 are the last events in their threads in σ', we conclude that σ' is a correct reordering of σ that respects $\leq^\sigma_{\mathsf{HB}}$. □

B PROOFS FOR ALGORITHM 1

We now prove Theorem 4.1, which states the correctness of Algorithm 1. Before establishing this claim we would like to introduce some notation and prove some auxiliary claims.

Let us fix a trace σ. Recall that for any event e, C_e is the (vector) timestamp assigned by Algorithm 1. Let us denote by L^e_ℓ the value of clock $\mathbb{L}_\ell$ just before the event e is processed. Similarly, let LW^e_x denote the value of clock $\mathbb{LW}_x$ just before e is processed. It is easy to see that the following invariant is maintained by Algorithm 1.

Proposition B.1. *Let e be an arbitrary event of trace σ. Let e_ℓ be the last $\texttt{rel}(\ell)$-event in σ before e, and let e_x be the last $\texttt{w}(x)$-event in σ before e (with respect to $\leq^\sigma_{\text{tr}}$). Note that e_ℓ and e_x maybe undefined. Then, $L^e_\ell = C_{e_\ell}$ and $LW^e_x = C_{e_x}$, where if an event f is undefined, we take $C_f = \bot$.*

Proof. The observation follows from the way vector clocks $\mathbb{L}_\ell$ and $\mathbb{LW}_x$ are updated. □

Another invariant that follows from the update rules of Algorithm 1 is the following.

Proposition B.2. *Let e_1 and e_2 be events of thread t such that $e_1 \leq^\sigma_{\text{tr}} e_2$, i.e., $e_1 \leq^\sigma_{\text{TO}} e_2$. Let t' be any thread such that $t \neq t'$. Then the following observations hold.*

(1) $C_{e_1} \sqsubseteq C_{e_2}$.
(2) $C_{e_1}(t') = C_{e_2}(t')$ unless there is an event e of thread t that is either an $\texttt{acq}$-event, or a $\texttt{r}$-event, or a $\texttt{join}$-event such that $e \neq e_1$ and $e_1 \leq^\sigma_{\text{tr}} e \leq^\sigma_{\text{tr}} e_2$.
(3) $C_{e_1}(t) = C_{e_2}(t)$ unless there is an event e of thread t that is either a $\texttt{rel}$-event, or a $\texttt{w}$-event, or a $\texttt{fork}$-event such that $e \neq e_2$ and $e_1 \leq^\sigma_{\text{tr}} e \leq^\sigma_{\text{tr}} e_2$; in this case $C_{e_1}(t) < C_{e_2}(t)$.

Proof. Follows from the way $\mathbb{C}_t$ is updated by Algorithm 1. □

We now prove the main lemma crucial to the correctness of Algorithm 1, that relates $\leq_{\text{SHB}}$ to the ordering on vector clocks.

Lemma B.3. *Let $e = \langle t, op\rangle$ be an event such that $C_e(t') = k$ for some $t' \neq t$. Let $e' = \langle t', op'\rangle$ be the last event such that $C_{e'}(t') = k$. Then $e' \leq^\sigma_{\text{SHB}} e$.*

Proof. The result will be proved by induction on the position of e in the trace σ. Observe that if e is the first event of σ, then $C_e(t') = 0$ for all $t' \neq t$, no matter what event e is. And there is no event $e' = \langle t', op'\rangle$ such that $C_{e'}(t') = 0$. Thus, the lemma holds vaccuously in the base case.

Let us now consider the inductive step. Define $e_1 = \langle t, op_1\rangle$ be the last event in σ before e (possibly same as e) such that op_1 is either $\texttt{acq}$, $\texttt{r}$, or $\texttt{join}$; if no such e_1 exists then take e_1 to be the first event performed by t. Notice, by our choice of e_1 and Proposition B.2(2), for every $t'' \neq t$, $C_{e_1}(t'') = C_e(t'')$. If $e_1 \neq e$, the result follows by induction hypothesis on e_1.

Let us assume $e_1 = e$. We need to consider different cases based on what e_1 is.

- **Case $e = e_1 = \langle t, \texttt{acq}(\ell)\rangle$:** Let f_1 be the event immediately before e in $\sigma|_t$ and f_2 be the event such that $C_{f_2} = L^e_\ell$ (given by Proposition B.1). Note that both f_1 and f_2 may be undefined. Also notice that, for any t', either $C_e(t') = 0$, or $C_e(t') = C_{f_1}(t') \neq 0$ (and f_1 is defined), or $C_e(t') = C_{f_2}(t') \neq 0$ (and f_2 is defined). If $C_e(t') = 0$ then the lemma follows vaccuously as in the base case because there is no event $e' = \langle t', op'\rangle$ with $C_{e'}(t') = 0$. Let us now consider the remaining cases. Let t_2 denote the thread performing f_2, if f_2 is defined. Consider the case when either $C_e(t') = C_{f_1}(t') \neq 0$ or $C_e(t') = C_{f_2}(t')$ with $t' \neq t_2$. In this situation, the lemma follows using the induction hypothesis on either f_1 or f_2 since both f_1 and f_2 (when defined) are $\leq^\sigma_{\text{SHB}} e$. The last case to consider is when $t' = t_2$ and $C_e(t') = C_{f_2}(t')$. By Proposition B.2(3), f_2 is the last event of $t' = t_2$ whose t'th component is k. Further, by definition $f_2 \leq^\sigma_{\text{SHB}} e$, and so the lemma holds.

- **Case $e = e_1 = \langle t, \mathtt{r}(x)\rangle$:** Let f_1 be the event immediately before e in $\sigma|_t$ and f_2 be the event such that $C_{f_2} = LW_x^e$ (given by Proposition B.1). Again, both f_1 and f_2 may be undefined. Also notice that, for any t', either $C_e(t') = 0$, or $C_e(t') = C_{f_1}(t') \neq 0$ (and f_1 is defined), or $C_e(t') = C_{f_2}(t') \neq 0$ (and f_2 is defined). If $C_e(t') = 0$ then the lemma follows vaccuously as in the base case because there is no event $e' = \langle t', op'\rangle$ with $C_{e'}(t') = 0$. Let us now consider the remaining cases. Let t_2 denote the thread performing f_2, if f_2 is defined. Consider the case when either $C_e(t') = C_{f_1}(t') \neq 0$ or $C_e(t') = C_{f_2}(t')$ with $t' \neq t_2$. In this situation, the lemma follows using the induction hypothesis on either f_1 or f_2 since both f_1 and f_2 (when defined) are $\leq^{\sigma}_{\mathsf{SHB}} e$. The last case to consider is when $t' = t_2$ and $C_e(t') = C_{f_2}(t')$. By Proposition B.2(3), f_2 is the last event of $t' = t_2$ whose t'th component is k. Further, by definition $f_2 \leq^{\sigma}_{\mathsf{SHB}} e$, and so the lemma holds.
- **Case $e = e_1 = \langle t, \mathtt{join}(t_1)\rangle$:** Let f_1 be the event immediately before e in $\sigma|_t$ and f_2 be the last event of the form $\langle t_1, op\rangle$. Again, both f_1 and f_2 may be undefined. Also notice that, for any t', either (a) $C_e(t') = 0$, or (b) $t_1 = t'$, $C_e(t') = 1$, and f_2 is undefined, or (c) $C_e(t') = C_{f_1}(t') \neq 0$ and f_1 is defined, or (d) $C_e(t') = C_{f_2}(t') \neq 0$ and f_2 is defined. In cases (a) or (b) above, the lemma follows vaccuously as in the base case because there is no event $e' = \langle t', op'\rangle$ with $C_{e'}(t') = k$ (where k is either 0 or 1 depending on which we case we consider). Let us now consider the remaining cases. Let t_2 denote the thread performing f_2, if f_2 is defined. Consider the case when either $C_e(t') = C_{f_1}(t') \neq 0$ or $C_e(t') = C_{f_2}(t')$ with $t' \neq t_2$ (and f_2 defined). In this situation, the lemma follows using the induction hypothesis on either f_1 or f_2 since both f_1 and f_2 (when defined) are $\leq^{\sigma}_{\mathsf{SHB}} e$. The last case to consider is when $t' = t_2$ and $C_e(t') = C_{f_2}(t')$. By definition, f_2 is the last event of $t' = t_2$ whose t'th component is k. Further, by definition $f_2 \leq^{\sigma}_{\mathsf{SHB}} e$, and so the lemma holds.
- **Case $e = e_1$ is the first event:** This is the case when the above 3 cases don't hold. So $e = e_1$ is not an $\mathtt{acq}$-event, nor a $\mathtt{r}$-event, nor a $\mathtt{join}$-event. Moreover, since e is the first event of thread t and is of the form $\langle t, op\rangle$, it must be the the thread t has not been forked by any thread in σ. Thus, for any $t' \neq t$, $C_e(t') = 0$. The lemma, therefore, follows vaccuously as in the base case. □

We are ready to present the proof of Theorem 4.1.

Theorem 4.1. *For events $e, e' \in \mathsf{Events}_\sigma$ such that $e \leq^{\sigma}_{\mathsf{tr}} e'$, $C_e \sqsubseteq C_{e'}$ iff $e \leq^{\sigma}_{\mathsf{SHB}} e'$*

Proof. Let us first prove the implication from left to right. Consider e, e' such that $e \leq^{\sigma}_{\mathsf{tr}} e'$. If $e \leq^{\sigma}_{\mathsf{TO}} e'$ then $e \leq^{\sigma}_{\mathsf{SHB}} e'$ since $\leq^{\sigma}_{\mathsf{TO}} \subseteq \leq^{\sigma}_{\mathsf{SHB}}$. On the other hand, if e and e' are not events of the same thread, then this direction of the theorem follows from Lemma B.3.

Let us now prove the implication from right to left. Consider events such that $e \leq^{\sigma}_{\mathsf{SHB}} e'$. Then, by definition, we have a sequence of events $e = f_1, f_2, \ldots f_k = e'$ such that for every i, $f_i \leq^{\sigma}_{\mathsf{tr}} f_{i+1}$ and either (i) f_i and f_{i+1} are both events of the form $\langle t, op\rangle$, or (ii) f_i is a $\mathtt{rel}(\ell)$-event and f_{i+1} is a $\mathtt{acq}(\ell)$-event, or (iii) f_i is a $\mathtt{fork}(t)$-event and f_{i+1} is an event of the form $\langle t, op\rangle$, or (iv) f_i is an event of the form $\langle t, op\rangle$ and f_{i+1} is a $\mathtt{join}(t)$-event, or (v) $f_i = \mathsf{lastWr}_\sigma(f_{i+1})$. In each of these cases, Algorithm 1 ensures that $C_{f_i} \sqsubseteq C_{f_{i+1}}$. Thus, we have $C_e \sqsubseteq C_{e'}$. □

We now prove Theorem 4.2. We first note some auxiliary propositions. Let us denote by R_x^e the value of clock $\mathbb{R}_x$ just before the event e is processed. Similarly, let W_x^e denote the value of clock $\mathbb{W}_x$ just before e is processed. It is easy to see that the following invariant is maintained by Algorithm 1.

Proposition B.4. *Let e be an arbitrary event of trace σ. Let $e_t^{\mathtt{r}(x)}$ be the last $\langle t, \mathtt{r}(x)\rangle$-event in σ before e, and let $e_t^{\mathtt{w}(x)}$ be the last $\langle t, \mathtt{w}(x)\rangle$-event in σ before e (with respect to $\leq^{\sigma}_{\mathsf{tr}}$). Note that $e_t^{\mathtt{r}(x)}$ and*

$e_t^{\mathsf{w}(x)}$ *maybe undefined. Then,* $\forall t, R_x(t) = C_{e_t^{\mathsf{r}(x)}}(t)$ *and* $\forall t, W_x(t) = C_{e_t^{\mathsf{w}(x)}}(t)$ *where if an event* f *is undefined, we take* $C_f = \bot$.

Proof. The observation follows from the way vector clocks $\mathbb{R}_x$ and $\mathbb{W}_x$ are updated. □

Lemma B.5. *Let* $e_1, e_2 \in \mathsf{Events}_\sigma$ *performed by threads* t_1 *and* t_2*, respectively, such that* $t_1 \neq t_2$*. Then,* $e_1 \leq^\sigma_{\mathsf{SHB}} e_2$ *iff* $C_{e_1} \sqsubseteq C_{e_2}[(C_{e_2}(t_2)+1)/t_2]$.

Proof. Let $c_2 = C_{e_2}(t_2)$. First suppose that $e_1 \leq^\sigma_{\mathsf{SHB}} e_2$. Then, from Theorem 4.1, we have $C_{e_1} \sqsubseteq C_{e_2}$ and thus $C_{e_1} \sqsubseteq C_{e_2}[(c_2+1)/t_2]$. Next, assume that $C_{e_1} \sqsubseteq C_{e_2}[(c_2+1)/t_2]$. In particular, $C_{e_1}(t_1) \leq C_{e_2}(t_1)$. Then by Lemma B.3, we have $e_1 \leq^\sigma_{\mathsf{SHB}} e_2$ □

Theorem 4.2. *Let* e *be a read/write event* $e \in \mathsf{Events}_\sigma$*. Algorithm 1 reports a race at* e *iff there is an event* $e' \in \mathsf{Events}_\sigma$ *such that* (e', e) *is an* $\leq^\sigma_{\mathsf{HB}}$*-schedulable race.*

Proof. Let us first consider the case when $\mathsf{pred}_\sigma(e)$ is not defined. Then, the value of the clock $\mathbb{C}_t = \bot[1/t]$ at line 19, 24 or 26 (depending upon whether e is a read or a write event). If the check $\neg(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$ passes, then there is a t' such that $\mathbb{W}_x(t') > \mathbb{C}_t$ and thus the there is an event e' (namely the last write event of x in thread t') that conflicts with e. Thus, (e', e) is a $\leq^\sigma_{\mathsf{HB}}$-schedulable race by Theorem 3.3. On the other hand, if the check fails, then $\mathbb{W}_x = \bot[1/t]$ or $\mathbb{W}_x = \bot$ and in either case there is no event that conflicts with e. One can similarly argue that the checks on Lines 24 and 26 are both necessary and sufficient for the case when e is a write event and $\mathsf{pred}_\sigma(e)$ is undefined.

Next we consider the case when $f = \mathsf{pred}_\sigma(e)$ is defined. Now let e be a read event. If the check $\neg(\mathbb{W}_x \sqsubseteq \mathbb{C}_t)$ passes, then there is a t' such that $\mathbb{W}_x(t') > \mathbb{C}_t(t')$ and thus there is an event $e' = \langle t', \mathsf{w}(x)\rangle$ such that $C_{e'}(t') > \mathbb{C}_t(t')$ and thus $C_{e'} \not\sqsubseteq \mathbb{C}_t$; note that it must be the case that $t' \neq t$. Depending upon whether f is a read/join/acquire event or a write/fork/release event, the value of the clock $\mathbb{C}_t$ at Line 19 is $\mathbb{C}_t = C_f$ or $\mathbb{C}_t = C_f[(C_f(t)+1)/t]$. In either case, by Lemma B.5, we have that $e' \not\leq^\sigma_{\mathsf{SHB}} f$. On the other hand if, $\mathbb{W}_x \sqsubseteq \mathbb{C}_t$, then $\forall t' \neq t, C_{e_{t'}^{\mathsf{w}(x)}}(t') \leq \mathbb{C}_t(t')$, where $e_u^{\mathsf{w}(x)}$ is the last write event of x performed by thread u. This means that for every event e' such that e' conflicts with, by Lemma B.5, we have $e' \leq^\sigma_{\mathsf{SHB}} f$ and thus (e', e) is not an $\leq^\sigma_{\mathsf{HB}}$-schedulable race. The argument for the case when e is a write event is similar. □

We now establish the asymptotic space and time bounds for Algorithm 1.

Theorem 4.3. *For a trace* σ *with* n *events,* T *threads,* V *variables, and* L *locks, Algorithm 1 runs in time* $O(nT \log n)$ *and uses* $O((V+L+T)T \log n)$ *space.*

Proof. Observe that for a trace of length n, every component of each of the vector clocks is bounded by n. Thus, each vector clock takes space $O(T \log n)$, where T is the number of threads. We have a vector clock for each thread, lock, and variable, which gives us a space bound of $O((V+T+L)T \log n)$. Notice that to process any event we need to update constantly many vector clocks. The time to update any single vector clock can be bounded by its size $O(T \log n)$. Thus, the total running time is $O(nT \log n)$. □

ACKNOWLEDGMENTS

We gratefully acknowledge National Science Foundation for supporting Umang Mathur (grant NSF CSR 1422798) and Mahesh Viswanathan (grant NSF CPS 1329991).

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

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 (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158119

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 (PLDI '07)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/1250734.1250762

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 237–252.

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 (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 151–162. http://dl.acm.org/citation.cfm?id=1924943.1924954

Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS '03)*. IEEE Computer Society, Washington, DC, USA, 286.2–.

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

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 (PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Jeff Huang and Arun K. Rajagopalan. 2016. Precise and Maximal Race Detection from Incomplete Traces. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 462–476. https://doi.org/10.1145/2983990.2984024

Shiyou Huang and Jeff Huang. 2016. Maximal Causality Reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 447–461. https://doi.org/10.1145/2983990.2984025

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 (PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

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

Peng Liu, Omer Tripp, and Xiangyu Zhang. 2016. IPA: Improving Predictive Analysis with Pointer Analysis. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 59–69.

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 (PLDI '09)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1542476.1542491

Umang Mathur. 2018. RAPID: Dynamic Analysis for Concurrent Programs. https://github.com/umangm/rapid. Accessed: July 30, 2018.

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. *CoRR* abs/1808.00185 (2018). http://arxiv.org/abs/1808.00185

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

Antoni Mazurkiewicz. 1987. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 278–324.

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 (PPoPP '07)*. ACM, New York, NY, USA, 142–143. https://doi.org/10.1145/1229428.1229457

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 (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 (PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

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 (PPoPP '03)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/781498.781528

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 (PPoPP '03)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical Static Race Detection for C. *ACM Trans. Program. Lang. Syst.* 33, 1, Article 3 (Jan. 2011), 55 pages.

Cosmin Radoi and Danny Dig. 2013. Practical Static Race Detection for Java Parallel Loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 178–190.

Grigore Rosu. 2018. RV-Predict, Runtime Verification. https://runtimeverification.com/predict/. Accessed: 2018-04-01.

Caitlin Sadowski and Jaeheon Yi. 2014. How Developers Use Data Race Detection Tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU '14)*. ACM, New York, NY, USA, 43–51. https://doi.org/10.1145/2688204.2688205

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 (NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 313–327.

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. *SIGOPS Oper. Syst. Rev.* 31, 5 (Oct. 1997), 27–37.

Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (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 (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 (WBIA '09)*. ACM, New York, NY, USA, 62–71.

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 (POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

Lorna A Smith and J Mark Bull. 2001. A multithreaded java grande benchmark suite. In *Proceedings of the third workshop on Java for high performance computing*.

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 (ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214.

Eran Yahav. 2001. Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 27–40. https://doi.org/10.1145/360204.360206

Sheng Zhan and Jeff Huang. 2016. ECHO: Instantaneous in Situ Race Detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 775–786.