A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

Umang Mathur
National University of Singapore
Singapore
umathur@comp.nus.edu.sg

Andreas Pavlogiannis
Aarhus University
Denmark
pavlogiannis@cs.au.dk

Hünkar Can Tunç
Aarhus University
Denmark
tunc@cs.au.dk

Mahesh Viswanathan
University of Illinois at Urbana-Champaign
USA
vmahesh@illinois.edu

ABSTRACT

Dynamic techniques are a scalable and effective way to analyze concurrent programs. Instead of analyzing all behaviors of a program, these techniques detect errors by focusing on a single program execution. Often a crucial step in these techniques is to define a causal ordering between events in the execution, which is then computed using *vector clocks*, a simple data structure that stores logical times of threads. The two basic operations of vector clocks, namely join and copy, require $\Theta(k)$ time, where k is the number of threads. Thus they are a computational bottleneck when k is large.

In this work, we introduce *tree clocks*, a new data structure that replaces vector clocks for computing causal orderings in program executions. Joining and copying tree clocks takes time that is roughly proportional to the number of entries being modified, and hence the two operations do not suffer the a-priori $\Theta(k)$ cost per application. We show that when used to compute the classic happens-before (HB) partial order, tree clocks are *optimal*, in the sense that no other data structure can lead to smaller asymptotic running time. Moreover, we demonstrate that tree clocks can be used to compute other partial orders, such as schedulable-happens-before (SHB) and the standard Mazurkiewicz (MAZ) partial order, and thus are a versatile data structure. Our experiments show that just by replacing vector clocks with tree clocks, the computation becomes from 2.02× faster (MAZ) to 2.66× (SHB) and 2.97× (HB) on average per benchmark. These results illustrate that tree clocks have the potential to become a standard data structure with wide applications in concurrent analyses.

CCS CONCEPTS

• **Software and its engineering** → **Software verification and validation**; • **Theory of computation** → *Theory and algorithms for application domains*; *Program analysis.*

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507734

KEYWORDS

concurrency, happens-before, vector clocks, dynamic analyses

ACM Reference Format:
Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507734

1 INTRODUCTION

The analysis of concurrent programs is one of the major challenges in formal methods, due to the non-determinism of inter-thread communication. The large space of communication interleavings poses a significant challenge to the programmer, as intended invariants can be broken by unexpected communication patterns. The subtlety of these patterns also makes verification a demanding task, as exposing a bug requires searching an exponentially large space [45]. Consequently, significant efforts are made towards understanding and detecting concurrency bugs efficiently [11, 20, 35, 62, 67, 72].

Dynamic analyses and partial orders. One popular approach to the scalability problem of concurrent program verification is dynamic analysis [24, 40, 43, 49]. Such techniques have the more modest goal of discovering faults by analyzing program executions instead of whole programs. Although this approach cannot prove the absence of bugs, it is far more scalable than static analysis and typically makes sound reports of errors. These advantages have rendered dynamic analyses a very effective and widely used approach to error detection in concurrent programs.

The first step in virtually all techniques that analyze concurrent executions is to establish a causal ordering between the events of the execution. Although the notion of causality varies with the application, its transitive nature makes it naturally expressible as a *partial order* between these events. One prominent example is the Mazurkiewicz partial order (MAZ), which often serves as the canonical way to represent concurrent traces [8, 44] (aka Shasha-Snir traces [61]). Another vastly common partial order is Lamport's *happens-before* (HB) [33], initially proposed in the context of distributed systems [59]. In the context of testing multi-threaded programs, partial orders play a crucial role in dynamic race detection

techniques, and have been thoroughly exploited to explore trade-offs between soundness, completeness, and running time of the underlying analysis. Prominent examples include the widespread use of HB [19, 24, 30, 49, 60], schedulably-happens-before (SHB) [37], causally-precedes (CP) [63], weak-causally-precedes (WCP) [31], doesn't-commute (DC) [53], and strong/weak-dependently-precedes (SDP/WDP) [28], M2 [48] and SyncP [41]. Beyond race detection, partial orders are often employed to detect and reproduce other concurrency bugs such as atomicity violations [9, 26, 42], deadlocks [57, 65], and other concurrency vulnerabilities [70].

Vector clocks in dynamic analyses. Often, the computational task of determining the partial ordering between events of an execution is achieved using a simple data structure called *vector clock*. Informally, a vector clock $\mathbb{C}$ is an integer array indexed by the processes/threads in the execution, and succinctly encodes the knowledge of a process about the whole system. For vector clock $\mathbb{C}_{t_1}$ associated with thread t_1, if $\mathbb{C}_{t_1}(t_2) = i$ then it means that the latest event of t_1 is ordered after the first i events of thread t_2 in the partial order. Vector clocks, thus seamlessly capture a partial order, with the point-wise ordering of the vector timestamps of two events capturing the ordering between the events with respect to the partial order of interest. For this reason, vector clocks are instrumental in computing the HB parial order efficiently [22, 23, 43], and are ubiquitous in the efficient implementation of analyses based on partial orders even beyond HB [24, 31, 32, 37, 42, 53, 57, 65].

The fundamental operation on vector clocks is the pointwise *join* $\mathbb{C}_{t_1} \leftarrow \mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2}$. This occurs whenever there is a causal ordering from thread t_2 to t_1. Operationally, a join is performed by updating $\mathbb{C}_{t_1}(t) \leftarrow \max(\mathbb{C}_{t_1}(t), \mathbb{C}_{t_2}(t))$ for every thread t, and captures the transitivity of causal orderings: as t_1 learns about t_2, it also learns about other threads t that t_2 knows about. Note that if t_1 is aware of a later event of t, this operation is vacuous. With k threads, a vector clock join takes $\Theta(k)$ time, and can quickly become a bottleneck even in systems with moderate k. This motivates the following question: is it possible to speed up join operations by proactively avoiding vacuous updates? The challenge in such a task comes from the efficiency of the join operation itself—since it only requires linear time in the size of the vector, any improvement must operate in sub-linear time, i.e., not even touch certain entries of the vector clock. We illustrate this idea on a concrete example, and present the key insight in this work.

Motivating example. Consider the example in Figure 1. It shows a partial trace from a concurrent system with 6 threads, along with the vector timestmamps at each event. When event e_2 is ordered before event e_3 due to synchronization, the vector clock $\mathbb{C}_{t_2}$ of t_2 is joined with that of $\mathbb{C}_{t_1}$, i.e., the t_j-th entry of $\mathbb{C}_{t_1}$ is updated to the maximum of $\mathbb{C}_{t_1}(t_j)$ and $\mathbb{C}_{t_2}(t_j)$[1]. Now assume that thread t_2 has learned of the current times of threads t_3, t_4, t_5 and t_6 via thread t_3. Since the t_3-th component of the vector timestamp of event e_1 is larger than the corresponding component of event e_2, t_1 cannot possibly learn any *new* information about threads t_4, t_5, and t_6 through the join performed at event e_3. Hence the naive pointwise updates will be redundant for the indices $j = \{3, 4, 5, 6\}$. Unfortunately, the flat structure of vector clocks is not amenable to such reasoning and cannot avoid these redundant operations.

To alleviate this problem, we introduce a new hierarchical tree-like data structure for maintaining vector times called a *tree clock*. The nodes of the tree encode local clocks, just like entries in a vector clock. In addition, the structure of the tree naturally captures which clocks have been learned transitively via intermediate threads. Figure 1 (right) depicts a (simplified) tree clock encoding the vector times of $\mathbb{C}_{t_2}$. The subtree rooted at thread t_3 encodes the fact that t_2 has learned about the current times of t_4, t_5 and t_6 *transitively*, via t_3. To perform the join operation $\mathbb{C}_{t_1} \leftarrow \mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2}$, we start from the root of $\mathbb{C}_{t_2}$, and traverse the tree as follows. Given a current node u, we proceed to the children of u *if and only if* u represents the time of a thread that is not known to t_1. Hence, in the example, the join operation will now access only the light-gray area of the tree, and thus compute the join without accessing the whole tree, resulting in a *sublinear running time* of the join operation.

The above principle, which we call *direct monotonicity* is one of two key ideas exploited by tree clocks; the other being *indirect monotonicity*. The key technical challenge in developing the tree clock data structure lies in (i) using direct and indirect monotonicity to perform efficient updates, and (ii) perform these updates such that direct and indirect monotonicity are preserved for future operations. Section 3.1 illustrates the intuition behind these two principles in depth.

Contributions. Our contributions are as follows.

1. We introduce *tree clock*, a new data structure for maintaining logical times in concurrent executions. In contrast to the flat structure of the traditional vector clocks, the dynamic hierarchical structure of tree clocks naturally captures ad-hoc communication patterns between processes. In turn, this allows for join and copy operations that run in *sublinear time*. As a data structure, tree clocks offer high versatility as they can be used in computing many different ordering relations.
2. We prove that tree clocks are an *optimal data structure* for computing HB, in the sense that, *for every input trace*, the total computation time cannot be improved (asymptotically) by replacing tree clocks with any other data structure. On the other hand, vector clocks do not enjoy this property.
3. We illustrate the versatility of tree clocks by presenting tree clock-based algorithms for the MAZ and SHB partial orders.
4. We perform a large-scale experimental evaluation of the tree clock data structure for computing the MAZ, SHB and HB partial orders, and compare its performance against the standard vector clock data structure. Our results show that just by replacing vector clocks with tree clocks, the computation becomes up to 2.97× faster on average. Given our experimental results, we believe that replacing vector clocks by tree clocks in partial order-based algorithms can lead to significant improvements on many applications. We provide the proofs for the theorems and lemmas presented in the paper in our technical report [39].

2 PRELIMINARIES

In this section we develop relevant notation and present standard concepts regarding concurrent executions, partial orders and vector clocks.

[1] As with many presentations of dynamic analyses using vector clocks [30], we assume that the *local* entry of a thread's clock increments by 1 after each event it performs. Hence, in Figure 1, the t_1-th entry of $\mathbb{C}_{t_1}$ increases from 27 to 28 after e_1 is performed.

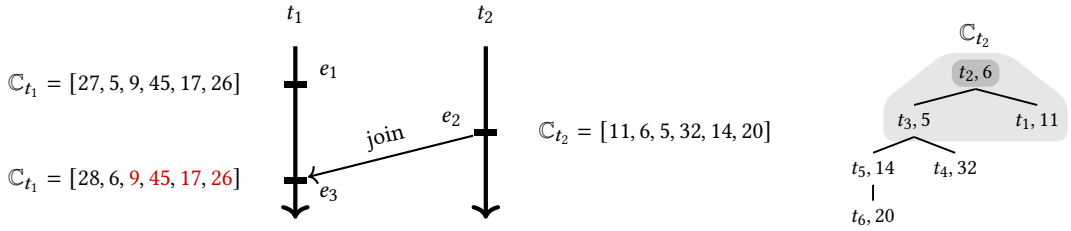


Figure 1: (Left) Illustration of the effect of a join operation $\mathbb{C}_{t_1} \leftarrow \mathbb{C}_{t_1} \sqcup \mathbb{C}_{t_2}$ on the clocks of the two threads. The j-th entry in timestamps correspond to thread t_j. Red entries remain unchanged, as t_1 already knows of a later time. (Right) A tree representation of the clocks $\mathbb{C}_{t_2}$ that encodes transitivity. Dark gray marks the threads whose clock has processed in $\mathbb{C}_{t_2}$ compared to $\mathbb{C}_{t_1}$ (i.e., just t_2). Light gray marks the nodes that we need to examine when performing the join operation.

2.1 Concurrent Model and Traces

We start with our main notation on traces. The exposition is standard and follows related work (e.g., [24, 31, 63]).

Events and traces. We consider execution traces of concurrent programs represented as a sequence of events performed by different threads. Each event is a tuple $e = \langle i, t, \text{op} \rangle$, where i is the unique event identifier of e, t is the identifier of the thread that performs e, and op is the operation performed by e, which can be one of the following types [2].

1. $\text{op} = \mathtt{r}(x)$, denoting that e reads global variable x.
2. $\text{op} = \mathtt{w}(x)$, denoting that e writes to global variable x.
3. $\text{op} = \mathtt{acq}(\ell)$, denoting that e acquires the lock ℓ.
4. $\text{op} = \mathtt{rel}(\ell)$, denoting that e releases the lock ℓ.

We write $\mathtt{tid}(e)$ and $\text{op}(e)$ to denote the thread identifier and the operation of e, respectively. For a read/write event e, we denote by $\text{Variable}(e)$ the (unique) variable that e accesses. We often ignore the identifier i and represent e as $\langle t, \text{op} \rangle$. In addition, we are often not interested in the thread of e, in which case we simply denote e by its operation, e.g., we refer to event $\mathtt{r}(x)$. When the variable of e is not relevant, it is also omitted (e.g., we may refer to a read event $\mathtt{r}$).

A (concrete) *trace* is a sequence of events $\sigma = e_1, \ldots, e_n$. The trace σ naturally defines a total order $\leq^{\sigma}_{\text{tr}}$ (pronounced *trace order*) over the set of events appearing in σ, i.e., we have $e \leq^{\sigma}_{\text{tr}} e'$ iff either $e = e'$ or e appears before e' in σ; when $e \neq e'$, then we say $e <^{\sigma}_{\text{tr}} e'$. We require that σ respects the semantics of locks. That is, for every lock ℓ and every two acquire events $\mathtt{acq}_1(\ell)$, $\mathtt{acq}_2(\ell)$ on the lock ℓ such that $\mathtt{acq}_1(\ell) <^{\sigma}_{\text{tr}} \mathtt{acq}_2(\ell)$, there exists a lock release event $\mathtt{rel}_1(\ell)$ in σ with $\mathtt{tid}(\mathtt{acq}_1(\ell)) = \mathtt{tid}(\mathtt{rel}_1(\ell))$ and $\mathtt{acq}_1(\ell) <^{\sigma}_{\text{tr}} \mathtt{rel}_1(\ell) <^{\sigma}_{\text{tr}} \mathtt{acq}_2(\ell)$. Finally, we denote by Thrds_{σ} the set of thread identifiers appearing in σ.

Thread order. Given a trace σ, the *thread order* $\leq^{\sigma}_{\text{TO}}$ is the smallest partial order such that $e_1 \leq^{\sigma}_{\text{TO}} e_2$ iff $\mathtt{tid}(e_1) = \mathtt{tid}(e_2)$ and $e_1 \leq^{\sigma}_{\text{tr}} e_2$. For an event e in a trace σ, the local time $\text{lTime}^{\sigma}(e)$ of e is the number of events that appear before e in the trace σ that are also performed by $\mathtt{tid}(e)$, i.e., $\text{lTime}^{\sigma}(e) = |\{e' \mid e' \leq^{\sigma}_{\text{TO}} e\}|$. We remark that the pair $(\mathtt{tid}(e), \text{lTime}^{\sigma}(e))$ uniquely identifies the event e in the trace σ.

[2] Fork and join events are ignored for ease of presentation. Handling such events is straightforward.

Conflicting events. Two events of e_1, e_2 of σ are called *conflicting*, denoted by $e_1 \asymp e_2$, if (i) $\text{Variable}(e_1) = \text{Variable}(e_2)$, (ii) $\mathtt{tid}(e_1) \neq \mathtt{tid}(e_2)$, and (iii) at least one of e_1, e_2 is a write event. The standard approach in concurrent analyses is to detect conflicting events that are causally independent, according to some pre-defined notion of causality, and can thus be executed concurrently.

2.2 Partial Orders, Vector Times and Vector Clocks

A partial order on a set S is a reflexive, transitive and anti-symmetric binary relation on the elements of S. Partial orders are the standard mathematical object for analyzing concurrent executions. The main idea behind such techniques is to define a partial order $\leq^{\sigma}_{\text{P}}$ on the set of events of the trace σ being analyzed. The intuition is that $\leq^{\sigma}_{\text{P}}$ captures *causality* — the relative order of two events of σ must be maintained if they are ordered by $\leq^{\sigma}_{\text{P}}$. More importantly, when two events e_1 and e_2 are unordered by $\leq^{\sigma}_{\text{P}}$ (denoted $e_1 \parallel^{\sigma}_{\text{P}} e_2$), then they can be deemed *concurrent*. This principle forms the backbone of all partial-order based concurrent analyses.

A naïve approach for constructing such a partial order is to explicitly represent it as an acyclic directed graph over the events of σ, and then perform a graph search whenever needed to determine whether two events are ordered. Vector clocks, on the other hand, provide a more efficient method to represent partial orders and therefore are the key data structure in most partial order-based algorithms. The use of vector clocks enables designing streaming algorithms, which are also suitable for monitoring the system. These algorithms associate *vector timestamps* [22, 23, 43] with events so that the point-wise ordering between timestamps reflects the underlying partial order. Let us formalize these notions now.

Vector Timestamps. Let us fix the set of threads Thrds in the trace. A *vector timestamp* (or simply vector time) is a mapping $\text{V}\colon \text{Thrds} \to \mathbb{N}$. It supports the following operations.

$$\text{V}_1 \sqsubseteq \text{V}_2 \quad \text{iff} \quad \forall t\colon \text{V}_1(t) \leq \text{V}_2(t) \qquad \text{(Comparison)}$$

$$\text{V}_1 \sqcup \text{V}_2 \quad = \quad \lambda t\colon \max(\text{V}_1(t), \text{V}_2(t)) \qquad \text{(Join)}$$

$$\text{V}[t' \to +i] \quad = \quad \lambda t\colon \begin{cases} \text{V}(t) + i, & \text{if } t = t' \\ \text{V}(t), & \text{otherwise} \end{cases} \qquad \text{(Increment)}$$

We write $\text{V}_1 = \text{V}_2$ to denote that $\text{V}_1 \sqsubseteq \text{V}_2$ and $\text{V}_2 \sqsubseteq \text{V}_1$. Let us see how vector timestamps provide an efficient implicit representation of partial orders.

Timestamping for a partial order. Consider a partial order $\leq_{\mathsf{P}}^{\sigma}$ defined on the set of events of σ such that $\leq_{\mathsf{TO}}^{\sigma} \subseteq \leq_{\mathsf{P}}^{\sigma}$. In this case, we can define the P-timestamp of an event e as the following vector timestamp:

$$C_e^{\leq_{\mathsf{P}}^{\sigma}} = \lambda u : \max\{\, \mathrm{lTime}^{\sigma}(f) \mid f \leq_{\mathsf{P}}^{\sigma} e,\ \mathrm{tid}(f) = u\}$$

In words, $C_e^{\leq_{\mathsf{P}}^{\sigma}}$ contains the timestamps of the events that appear the latest in their respective threads such that they are ordered before e in the partial order $\leq_{\mathsf{P}}^{\sigma}$. We remark that $C_e^{\leq_{\mathsf{P}}^{\sigma}}(\mathrm{tid}(e)) = \mathrm{lTime}^{\sigma}(e)$. The following observation then shows that the timestamps defined above precisely capture the order $\leq_{\mathsf{P}}^{\sigma}$.

Lemma 1. *Let $\leq_{\mathsf{P}}^{\sigma}$ be a partial order defined on the set of events of trace σ such that $\leq_{TO}^{\sigma} \subseteq \leq_{\mathsf{P}}^{\sigma}$. Then for any two events e_1, e_2 of σ, we have, $C_{e_1}^{\leq_{\mathsf{P}}^{\sigma}} \sqsubseteq C_{e_2}^{\leq_{\mathsf{P}}^{\sigma}} \iff e_1 \leq_{\mathsf{P}}^{\sigma} e_2$.*

In words, Lemma 1 implies that, in order to check whether two events are ordered according to $\leq_{\mathsf{P}}^{\sigma}$, it suffices to compare their vector timestamps.

The vector clock data structure. When establishing a causal order over the events of a trace, the timestamps of an event is computed using timestamps of other events in the trace. Instead of explicitly storing timestamps of each event, it is often sufficient to store only the timestamps of a few events, as the algorithms is running. Typically a data-structure called *vector clocks* is used to store vector times. Vector clocks are implemented as a simple integer array indexed by thread identifiers, and they support all the operations on vector timestamps. A useful feature of this data-structure is the ability to perform in-place operations. In particular, there are methods such as $\mathtt{Join}(\cdot)$, $\mathtt{Copy}(\cdot)$ or $\mathtt{Increment}(\cdot,\cdot)$ that store the result of the corresponding vector time operation in the original instance of the data-structure. For example, for a vector clock $\mathbb{C}$ and a vector time V, a function call $\mathbb{C}.\mathtt{Join}(V)$ stores the value $\mathbb{C} \sqcup V$ back in $\mathbb{C}$. Each of these operations iterates over all the thread identifiers (indices of the array representation) and compares the corresponding components in $\mathbb{C}$ and V. The running time of the join operation for the vector clock data structure is thus $\Theta(k)$, where k is the number of threads. Similarly, copy and comparison operations take $\Theta(k)$ time.

2.3 The Happens-Before Partial Order

Lamport's Happens-Before (HB) [33] is one of the most frequently used partial orders for the analysis of concurrent executions, with wide applications in domains such as dynamic race detection. Here we use HB to illustrate the disadvantages of vector clocks and form the basis for the tree clock data structure. In later sections we show how tree clocks also apply to other partial orders, such as Schedulably-Happens-Before and the Mazurkiewicz partial order.

Happens-before. Given a trace σ, the *happens-before* (HB) partial order $\leq_{\mathsf{HB}}^{\sigma}$ of σ is the smallest partial order over the events of σ that satisfies the following conditions.

1. $\leq_{\mathsf{TO}}^{\sigma} \subseteq \leq_{\mathsf{HB}}^{\sigma}$.
2. For every release event $\mathtt{rel}(\ell)$ and acquire event $\mathtt{acq}(\ell)$ on the same lock ℓ with $\mathtt{rel}(\ell) <_{\mathrm{tr}}^{\sigma} \mathtt{acq}(\ell)$, we have $\mathtt{rel}(\ell) \leq_{\mathsf{HB}}^{\sigma} \mathtt{acq}(\ell)$.

For two events e_1, e_2 in trace σ, we use $e_1 \parallel_{\mathsf{HB}}^{\sigma} e_2$ to denote that neither $e_1 \leq_{\mathsf{HB}}^{\sigma} e_2$, nor $e_2 \leq_{\mathsf{HB}}^{\sigma} e_1$. We say $e_1 <_{\mathsf{HB}}^{\sigma} e_2$ when $e_1 \neq e_2$ and $e_1 \leq_{\mathsf{HB}}^{\sigma} e_2$. Given a trace σ, two events e_1, e_2 of σ are said to be in a *happens-before (data) race* if (i) $e_1 \asymp e_2$ and (ii) $e_1 \parallel_{\mathsf{HB}}^{\sigma} e_2$.

Algorithm 1: Computing the HB partial order.

```
1 procedure acquire(t, ℓ)
2     C_t.Join(C_ℓ)
3 procedure release(t, ℓ)
4     C_ℓ.Copy(C_t)
```

The happens-before algorithm. In light of Lemma 1, race detection based on HB constructs the $\leq_{\mathsf{HB}}^{\sigma}$ partial order in terms of vector timestamps and detects races using these. The core algorithm for constructing $\leq_{\mathsf{HB}}$ is shown in Algorithm 1. The algorithm maintains a vector clock $\mathbb{C}_t$ for every thread $t \in \mathrm{Thrds}$, and a similar one $\mathbb{C}_\ell$ for every lock ℓ. When processing an event $e = \langle t, \mathrm{op}\rangle$, it performs an update $\mathbb{C}_t.\mathtt{Increment}(t, 1)$, which is implicit and not shown in Algorithm 1. Moreover, if $\mathrm{op} = \mathtt{acq}(\ell)$ or $\mathrm{op} = \mathtt{rel}(\ell)$, the algorithm executes the corresponding procedure. The HB-timestamp of e is then simply the value stored in $\mathbb{C}_{\mathrm{tid}(e)}$ right after e has been processed.

Running time using vector clocks. If a trace σ has n events and k threads, computing the HB partial order with Algorithm 1 and using vector clocks takes $O(n \cdot k)$ time. The quadratic bound occurs because every vector clock join and copy operation iterates over all k threads.

3 THE TREE CLOCK DATA STRUCTURE

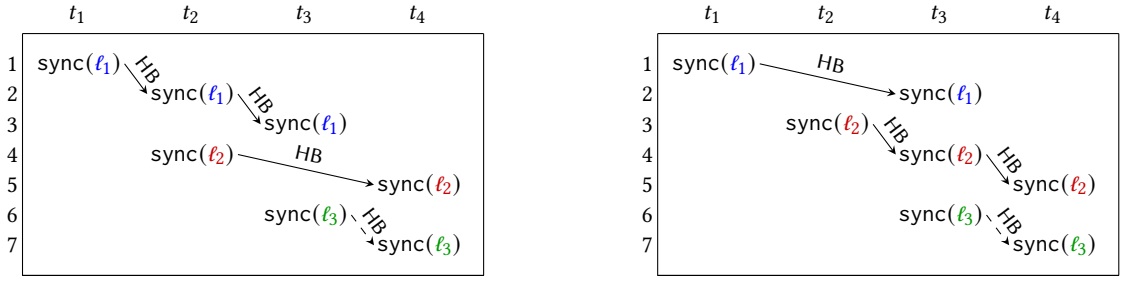
In this section we introduce tree clocks, a new data structure for representing logical times in concurrent and distributed systems. We first illustrate the intuition behind tree clocks, and then develop the data structure in detail.

3.1 Intuition

Like vector clocks, tree clocks represent vector timestamps that record a thread's knowledge of events in other threads. Thus, for each thread t, a tree clock records the last known local time of t. However, unlike a vector clock which is flat, a tree clock maintains this information hierarchically — nodes store local times of a thread, while the tree structure records how this information has been obtained transitively through intermediate threads. In the following examples we use the operation $\mathtt{sync}(\ell)$ to denote the sequence $\mathtt{acq}(\ell), \mathtt{rel}(\ell)$.

1. Direct monotonicity. Recall that a vector clock-based algorithm like Algorithm 1 maintains a vector clock $\mathbb{C}_t$ which intuitively captures thread t's knowledge about all threads. However, it does not maintain *how* this information was acquired. Knowledge of how such information was acquired can be exploited in join operations, as we show through an example. Consider a computation of the HB partial order for the trace σ shown in Figure 2a. At event e_7, thread t_4 transitively learns information about events in the trace through thread t_3 because $e_6 <_{\mathsf{HB}}^{\sigma} e_7$ (dashed edge in Figure 2a). This is accomplished by joining with clock $\mathbb{C}_{t_3}$ of thread t_3. Such a join using vector clocks will take 4 steps because we need to take the pointwise maximum of two vectors of length 4.

Suppose in addition to these timestamps, we maintain how these timestamps were updated in each clock. This would allow one to make the following observations.



(a) Direct monotonicity. **(b) Indirect monotonicity.**

Figure 2: Illustration of the two insights behind tree clocks. An event $\texttt{sync}(\ell)$ represents two events $\texttt{acq}(\ell), \texttt{rel}(\ell)$.

1. Thread t_3 knows of event e_1 of t_1 transitively, through event e_2 of thread t_2.
2. Thread t_4 (before the join at e_7) knows of event e_1 through e_4 of thread t_2.

Before the join, since t_4 has a more recent view of t_2 when compared to t_3, it is aware of all the information that thread t_3 knows about the world via thread t_2. Thus, when performing the join, we need not examine the component corresponding to thread t_1 in the two clocks. Tree clocks, by maintaining such additional information, can avoid examining some components of a vector timestamp and yield sublinear updates.

2. Indirect monotonicity. We now illustrate that if in addition to information about "how a view of a thread was updated", we also maintained "when the view of a thread was updated", the cost of join operations can be further reduced. Consider the trace σ of Figure 2b. At each of the events of thread t_4, it learns about events in the trace transitively through thread t_3 by performing two join operations. At the first join (event e_5), thread t_4 learns about events e_1, e_2, e_3 transitively through event e_4. At event e_7, thread t_4 finds out about new events in thread t_3 (namely, e_6). However, it does not need to update its knowledge about threads t_1 and t_2 — thread t_3's information about threads t_1 and t_2 were acquired by the time of event e_4 about which thread t_4 is aware. Thus, if information about when knowledge was acquired is also kept, this form of "indirect monotonicity" can be exploited to avoid examining all components of a vector timestamp.

The flat structure of vector clocks misses the transitivity of information sharing, and thus arguments based on monotonicity are lost, resulting in vacuous operations. On the other hand, tree clocks maintain transitivity in their hierarchical structure. This enables reasoning about direct and indirect monotonicity, and thus avoid redundant operations.

3.2 Tree Clocks

We now present the tree clock data structure in detail.

Tree clocks. A tree clock TC consists of the following.

1. $\mathrm{T} = (\mathcal{V}, \mathcal{E})$ is a *rooted tree* of nodes of the form $(\mathrm{tid}, \mathrm{clk}, \mathrm{aclk}) \in \mathrm{Thrds} \times \mathbb{N}^2$. Every node u stores its children in an ordered list $\mathrm{Chld}(u)$ of descending aclk order. We also store a pointer $\mathrm{Prnt}(u)$ of u to its parent in T.
2. $\mathrm{ThrMap}\colon \mathrm{Thrds} \to \mathcal{V}$ is a *thread map*, with the property that if $\mathrm{ThrMap}(t) = (\mathrm{tid}, \mathrm{clk}, \mathrm{aclk})$, then $t = \mathrm{tid}$.

Figure 3: The tree clock of t_4 after processing the event e_7 in the traces of Figure 2a (left) and Figure 2b (right).

We denote by $\mathrm{T}.\mathrm{root}$ the root of T, and for a tree clock TC we refer by $\mathrm{TC}.\mathrm{T}$ and $\mathrm{TC}.\mathrm{ThrMap}$ to the rooted tree and thread map of TC, respectively. For a node $u = (\mathrm{tid}, \mathrm{clk}, \mathrm{aclk})$ of T, we let $u.\mathrm{tid} = \mathrm{tid}$, $u.\mathrm{clk} = \mathrm{clk}$ and $u.\mathrm{aclk} = \mathrm{aclk}$, and say that u *points to* the unique event e with $\mathrm{tid}(e) = \mathrm{tid}$ and $\mathrm{lTime}(e) = \mathrm{clk}$. Intuitively, if $v = \mathrm{Prnt}(u)$, then u represents the following information.

1. TC has the *local time* $u.\mathrm{clk}$ for thread $u.\mathrm{tid}$.
2. $u.\mathrm{aclk}$ is the *attachment time* of $v.\mathrm{tid}$, which is the local time of v when v learned about $u.\mathrm{clk}$ of $u.\mathrm{tid}$ (this will be the time that v had when u was attached to v).

Naturally, if $u = \mathrm{T}.\mathrm{root}$ then $u.\mathrm{aclk} = \bot$. See Figure 3.

Tree clock operations. Just like vector clocks, tree clocks provide functions for initialization, update and comparison. There are two main operations worth noting. The first is $\mathrm{TC}_1.\texttt{Join}(\mathrm{TC}_2)$ joins the tree clock TC_2 to TC_1. In contrast to vector clocks, this operation takes advantage of the direct and indirect monotonicity outlined in Section 3.1 to perform the join in sublinear time in the size of TC_1 and TC_2 (when possible). The second is $\texttt{MonotoneCopy}$. We use $\mathrm{TC}_1.\texttt{MonotoneCopy}(\mathrm{TC}_2)$ to copy TC_2 to TC_1 when we know that $\mathrm{TC}_1 \sqsubseteq \mathrm{TC}_2$. The idea is that when this holds, the copy operation has the same semantics as the join, and hence the principles that make $\texttt{Join}$ run in sublinear time also apply to $\texttt{MonotoneCopy}$.

Algorithm 2 gives a pseudocode description of this functionality. The functions on the left column present operations that can be performed on tree clocks, while the right column lists helper routines for the more involved functions $\texttt{Join}$ and $\texttt{MonotoneCopy}$. In the following we give an intuitive description of each function.

1. $\texttt{Init}(t)$. This function initializes a tree clock TC_t that belongs to thread t, by creating a node $u = (t, 0, \bot)$. Node u will always be the root of TC_t. This initialization function is only used for tree clocks that represent the clocks of threads. Auxiliary tree clocks for storing vector times of release events do not execute this initialization.

Algorithm 2: The tree clock data structure.

```
// Initialize a tree clock for thread t
function Init(t)
    Let u ← (t, 0, ⊥)
    Make u the root of T
    Let ThrMap(t) ← u

// Get the clock for thread t
function Get(t)
    if TC.ThrMap(t) ≠ ⊥ then
        Let u ← ThrMap(t)
        return u.clk
    return 0

// Increment the clock of the root thread
function Increment(i)
    Let z ← T.root
    z.clk ← z.clk + i

// True iff ⊑ TC'
function LessThan(TC')
    Let z ← T.root
    return z.clk ≤ TC'.Get(z.tid)

// Update with ⊔ TC'
function Join(TC')
    Let z' ← TC'.T.root
    if z'.clk ≤ Get(z'.tid) then
        return
    Let S ← an empty stack
    getUpdatedNodesJoin (S, z')
    detachNodes (S)
    attachNodes (S)
    // Place the updated subtree under the root of T
    Let w ← ThrMap(z'.tid)
    Let z ← T.root
    Assign w.aclk ← z.clk
    pushChild (w, z)

// Monotone copy, assumes that this ⊑ TC'
function MonotoneCopy(TC')
    Let z' ← TC'.T.root
    Let z ← T.root
    Let S ← an empty stack
    getUpdatedNodesCopy (S, z', z)
    detachNodes (S)
    attachNodes (S)
    // New root has the same tid as the root of TC'.T
    Assign T.root ← ThrMap(z'.tid)

// Populate S with a pre-order traversal of the subtree rooted at u'
   with nodes whose clock has progressed
routine getUpdatedNodesJoin(S, u')
    foreach v' in Chld(u') do
        if Get(v'.tid) < v'.clk then getUpdatedNodesJoin (S, v');
        else if v'.aclk ≤ Get(u'.tid) then break;
    Push u' in S

// Detach from T the nodes with tid that appears in S
routine detachNodes(S)
    foreach v' in S do
        if ThrMap(v'.tid) ≠ ⊥ then
            Let v ← ThrMap(v'.tid)
            if v ≠ T.root then
                Let x ← Prnt(v)
                Remove v from Chld(x)

// Re-attach the nodes of T with tid that appears in S to obtain the
   shape corresponding to TC'.T
routine attachNodes(S)
    while S is not empty do
        Let u' ← pop S
        if ThrMap(u'.tid) ≠ ⊥ then
            Let u ← ThrMap(u'.tid)
        else
            Let u ← (u'.tid, 0, ⊥)
            Let ThrMap(u.tid) ← u
        Assign u.clk ← u'.clk
        Let y' ← Prnt(u')
        if y' ≠ ⊥ then
            Assign u.aclk ← u'.aclk
            Let y ← ThrMap(y'.tid)
            pushChild (u, y)

// Similar to getUpdatedNodesJoin
routine getUpdatedNodesCopy(S, u', z)
    foreach v' in Chld(u') do
        if Get(v'.tid) < v'.clk then
            getUpdatedNodesCopy (S, v', z)
        else
            if z ≠ ⊥ and v'.tid = z.tid then Push v' in S ;
            if v'.aclk ≤ Get(u'.tid) then break;
    Push u' in S

// Push u in the front of head of Chld(v)
routine pushChild(u, v)
    Assign Prnt(u) ← v
    Push u to the front of Chld(v)
```

2. `Get(`t`)`. This function simply returns the time of thread t stored in TC, while it returns 0 if t is not present in TC.
3. `Increment(`i`)`. This function increments the time of the root node of TC. It is only used on tree clocks that have been initialized using `Init`, i.e., the tree clock belongs to a thread that is always stored in the root of the tree.
4. `LessThan(`TC$'$`)`. This function compares the vector time of TC to the vector time of TC$'$, i.e., it returns True iff TC $\sqsubseteq$ TC$'$.

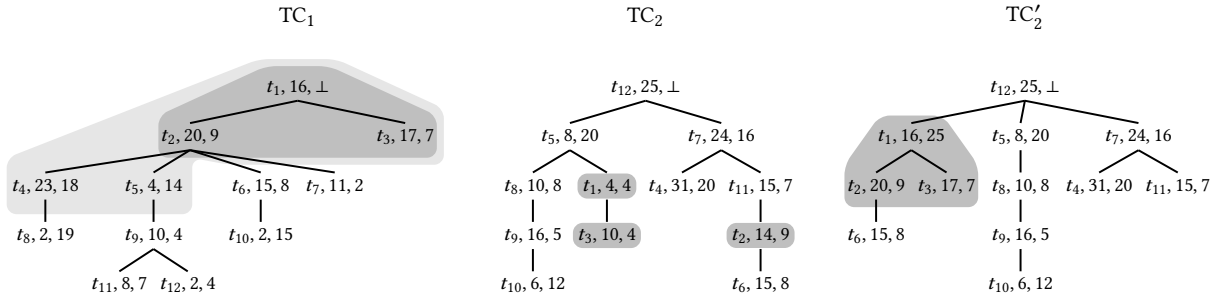


Figure 4: Illustration of TC_2 .Join(TC_1). Light gray marks the nodes of TC_1 whose time is compared to the time of the respective thread in TC_2 (i.e., the total iterations in Line 37). Dark gray marks the nodes that are updating/being updated (i.e., the size of $\mathcal{S}$). TC_2' is the result of the join, where dark gray marks the sub-tree updated by Join.

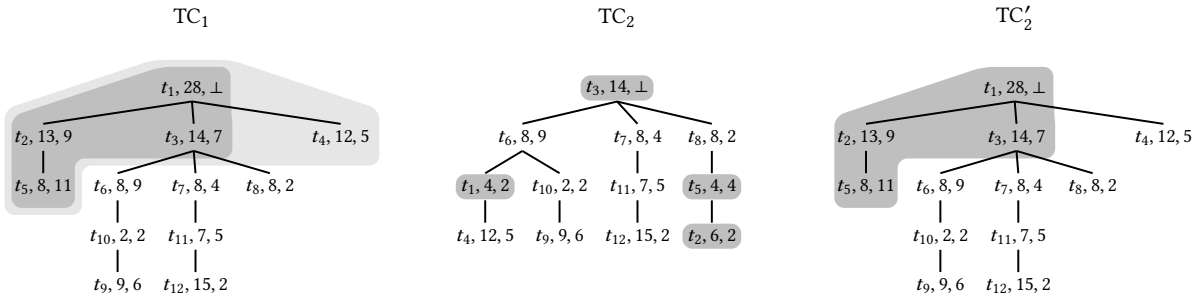


Figure 5: Illustration of TC_2 .MonotoneCopy(TC_1). Light gray marks the nodes of TC_1 whose time is compared to the time of the respective thread in TC_2 (i.e., the total iterations in Line 63). Dark gray marks the nodes that are updating/being updated (i.e., the size of $\mathcal{S}$). TC_2' is the result of the copy, where dark gray marks the sub-tree updated by MonotoneCopy. Node $(t_3, 14, \bot)$ (i.e., the root) of TC_2 is updated although t_3 has not progressed in TC_1, as it is placed under the new root $(t_1, 28, \bot)$ in TC_2'.

5. Join(TC'). This function implements the join operation with TC', i.e., updating $TC \leftarrow TC \sqcup TC'$. At a high level, the function performs the following steps.

1. Routine getUpdatedNodesJoin performs a pre-order traversal of TC', and gathers in a stack $\mathcal{S}$ the nodes of TC' that have progressed in TC' compared to TC. The traversal may stop early due to direct or indirect monotonicity, hence, this routine generally takes sub-linear time.
2. Routine detachNodes detaches from TC the nodes whose tid appears in $\mathcal{S}$, as these will be repositioned in the tree.
3. Routine attachNodes updates the nodes of TC that were detached in the previous step, and repositions them in the tree. This step effectively creates a subtree of nodes of TC that is identical to the subtree of TC' that contains the progressed nodes computed by getUpdatedNodesJoin.
4. Finally, the last 4 lines of Join attach the subtree constructed in the previous step under the root z of TC, at the front of the $\text{Chld}(z)$ list.

Figure 4 provides an illustration.

6. MonotoneCopy(TC'). This function implements the copy operation $TC \leftarrow TC'$ assuming that $TC \sqsubseteq TC'$. The function is very similar to Join. The key difference is that this time, the root of TC is always considered to have progressed in TC', even if the respective times are equal. This is required for changing the root of TC from the current node to one with tid equal to the root of TC'. Figure 5 provides an illustration.

The crucial parts of Join and MonotoneCopy that exploit the hierarchical structure of tree clocks are in getUpdatedNodesJoin and getUpdatedNodesCopy. In each case, we proceed from a parent u' to its children v' only if u' has progressed wrt its time in TC (recall Figure 2a), capturing *direct monotonicity*. Moreover, we proceed from a child v' of u' to the next child v'' (in order of appearance in $\text{Chld}(u')$) only if TC is not yet aware of the attachment time of v' on u' (recall Figure 2b), capturing *indirect monotonicity*.

Remark 1 (Constant time epoch accesses). *The function* TC .Get(t) *returns the time of thread t stored in* TC *in $O(1)$ time, just like vector clocks. This allows all epoch-related optimizations [24, 54] from vector clocks to apply to tree clocks.*

4 TREE CLOCKS FOR HAPPENS-BEFORE

Let us see how tree clocks are employed for computing the HB partial order. We start with the following observation.

Lemma 2 (Monotonicity of copies). *Right before Algorithm 1 processes a lock-release event $\langle t, \texttt{rel}(\ell)\rangle$, we have $\mathbb{C}_\ell \sqsubseteq \mathbb{C}_t$.*

Tree clocks for HB. Algorithm 3 shows the algorithm for computing HB using the tree clock data structure for implementing vector times. When processing a lock-acquire event, the vector-clock join

operation has been replaced by a tree-clock join. Moreover, in light of Lemma 2, when processing a lock-release event, the vector-clock copy operation has been replaced by a tree-clock monotone copy.

Algorithm 3: HB with tree clocks.

```
1 procedure acquire(t, ℓ)
2 |  C_t.Join(C_ℓ)
3 procedure release(t, ℓ)
4 |  C_ℓ.MonotoneCopy(C_t)
```

Correctness. We now state the correctness of Algorithm 3, i.e., we show that the algorithm indeed computes the HB partial order. We start with two monotonicity invariants of tree clocks.

Lemma 3. *Consider any tree clock $\mathbb{C}$ and node u of $\mathbb{C}$. T. For any tree clock $\mathbb{C}'$, the following assertions hold.*

1. Direct monotonicity: *If $u.\mathrm{clk} \leq \mathbb{C}'.\mathtt{Get}(u.\mathrm{tid})$ then for every descendant w of u we have that $w.\mathrm{clk} \leq \mathbb{C}'.\mathtt{Get}(w.\mathrm{tid})$.*
2. Indirect monotonicity: *If $v.\mathrm{aclk} \leq \mathbb{C}'.\mathtt{Get}(u.\mathrm{tid})$ where v is a child of u then for every descendant w of v we have that $w.\mathrm{clk} \leq \mathbb{C}'.\mathtt{Get}(w.\mathrm{tid})$.*

The following lemma follows from the above invariants and establishes that Algorithm 3 with tree clocks computes the correct timestamps on all events, i.e., the correctness of tree clocks for HB.

Lemma 4. *When Algorithm 3 processes an event e, the vector time stored in the tree clock $\mathbb{C}_{\mathrm{tid}(e)}$ is $C_e^{\leq_{HB}^{\sigma}}$.*

Data structure optimality. Just like vector clocks, computing HB with tree clocks takes $\Theta(n \cdot k)$ time in the worst case, and it is known that this quadratic bound is likely to be tight for common applications such as dynamic race prediction [32]. However, we have seen that tree clocks can take sublinear time on join and copy operations, whereas vector clocks always require time linear in the size of the vector (i.e., $\Theta(k)$). A natural question arises: is there a more efficient data structure than tree clocks? More generally, what is the most efficient data structure for the HB algorithm to represent vector times? To answer this question, we define *vector-time work*, which gives a lower bound on the number of data structure operations that HB has to perform regardless of the actual data structure used to store vector times. Then, we show that tree clocks match this lower bound, hence achieving optimality for HB.

Vector-time work. Consider the general HB algorithm (Algorithm 1) and let $\mathfrak{D} = \{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_m\}$ be the set of the vector-time data structures used. Consider the execution of the algorithm on a trace σ. Given some $1 \leq i \leq |\sigma|$, we let C_j^i denote the vector time represented by $\mathbb{C}_j$ after the algorithm has processed the i-th event of σ. We define the *vector-time work* (or *vt-work*, for short) on σ as

$$\mathrm{VTWork}(\sigma) = \sum_{1 \leq i \leq |\sigma|} \sum_{j} |t \in \mathrm{Thrds} : C_j^{i-1}(t) \neq C_j^i(t)|.$$

In words, for every processed event, we add the number of vector-time entries that change as a result of processing the event, and $\mathrm{VTWork}(\sigma)$ counts the total number of entry updates in the overall course of the algorithm. Note that vt-work is independent of the data structure used to represent each $\mathbb{C}_j$, and satisfies the inequality

$$n \leq \mathrm{VTWork}(\sigma) \leq n \cdot k.$$

as with every event of σ the algorithm updates one of $\mathbb{C}_j$.

Vector-time optimality. Given an input trace σ, we denote by $\mathcal{T}_{\mathrm{DS}}(\sigma)$ the time taken by the HB algorithm (Algorithm 1) to process σ using the data structure DS to store vector times. Intuitively, $\mathrm{VTWork}(\sigma)$ captures the number of times that instances of DS change state. For data structures that represent vector times explicitly, $\mathrm{VTWork}(\sigma)$ presents a natural lower bound for $\mathcal{T}_{\mathrm{DS}}(\sigma)$. Hence, we say that the data structure DS is *vt-optimal* if $\mathcal{T}_{\mathrm{DS}}(\sigma) = O(\mathrm{VTWork}(\sigma))$. It is not hard to see that vector clocks are not vt-optimal, i.e., taking DS = VC to be the vector clock data structure, one can construct simple traces σ where $\mathrm{VTWork}(\sigma) = O(n)$ but $\mathcal{T}_{\mathrm{DS}}(\sigma) = \Omega(n \cdot k)$, and thus the running time is k times more than the vt-work that must be performed on σ. In contrast, the following theorem states that tree clocks are vt-optimal.

Theorem 1 (Tree-clock Optimality). *For any input trace σ, we have $\mathcal{T}_{\mathrm{TC}}(\sigma) = O(\mathrm{VTWork}(\sigma))$.*

The key observation behind Theorem 1 is that, when HB uses tree clocks, the total number of tree-clock entries that are accessed over all join and monotone copy operations (i.e., the sum of the sizes of the light-gray areas in Figure 4 and Figure 5) is $\leq 3 \cdot \mathrm{VTWork}(\sigma)$.

Remark 2. *Theorem 1 establishes* strong optimality *for tree clocks, in the sense that they are vt-optimal* on every input. *This is in contrast to usual notions of optimality that is guaranteed on* only some *inputs.*

5 TREE CLOCKS IN OTHER PARTIAL ORDERS

5.1 Schedulable-Happens-Before

SHB is a strengthening of HB, introduced recently [37] in the context of race detection. Given a trace σ and a read event $\mathtt{r}$ let $\mathrm{lw}_\sigma(\mathtt{r})$ be the last write event of σ before $\mathtt{r}$ with $\mathrm{Variable}(\mathtt{w}) = \mathrm{Variable}(\mathtt{r})$. SHB is the smallest partial order that satisfies the following.

1. $\leq_{\mathrm{HB}}^{\sigma} \subseteq \leq_{\mathrm{SHB}}^{\sigma}$.
2. for every read event $\mathtt{r}$, we have $\mathrm{lw}_\sigma(\mathtt{r}) \leq_{\mathrm{SHB}}^{\sigma} \mathtt{r}$.

Algorithm for SHB. Similarly to HB, the SHB partial order is computed by a single pass of the input trace σ using vector-times [37]. The SHB algorithm processes synchronization events (i.e., $\mathtt{acq}(\ell)$ and $\mathtt{rel}(\ell)$) similarly to HB. In addition, for each variable x, the algorithm maintains a data structure LW_x that stores the vector time of the latest write event on x. When a write event $\mathtt{w}(x)$ is encountered, the vector time $\mathbb{C}_{\mathrm{tid}(\mathtt{w})}$ is copied to LW_x. In turn, when a read event $\mathtt{r}(x)$ is encountered the algorithm joins LW_x to $\mathbb{C}_{\mathrm{tid}(\mathtt{r})}$.

SHB with tree clocks. Tree clocks can directly be used as the data structure to store vector times in the SHB algorithm. We refer to Algorithm 4 for the pseudocode. The important new component is the function `CopyCheckMonotone` in Line 8 that copies the vector time of $\mathbb{C}_t$ to LW_x. In contrast to `MonotoneCopy`, this copy is not guaranteed to be monotone, i.e., we might have $\mathrm{LW}_x \not\sqsubseteq \mathbb{C}_t$. Note, however, that using tree clocks, this test requires only constant time. Internally, `CopyCheckMonotone` performs `MonotoneCopy` if $\mathrm{LW}_x \sqsubseteq \mathbb{C}_t$ (running in sublinear time), otherwise it performs a deep copy for the whole tree clock (running in linear time). In practice, we expect that most of the times `CopyCheckMonotone` results in `MonotoneCopy` and thus is very efficient. The key insight is that if `MonotoneCopy` is not used, then $\mathrm{LW}_x \not\sqsubseteq \mathbb{C}_t$ and thus we have a race $(\mathrm{lw}_\sigma(\mathtt{r}), \mathtt{r})$. Hence, the number of times a deep copy

Algorithm 4: SHB with tree clocks.

```
1 procedure acquire(t, ℓ)
2     C_t.Join(L_ℓ)
3 procedure read(t, x)
4     C_t.Join(LW_x)
5 procedure release(t, ℓ)
6     L_ℓ.MonotoneCopy(C_t)
7 procedure write(t, x)
8     LW_x.CopyCheckMonotone(C_t)
```

Algorithm 5: MAZ with tree clocks.

```
1  procedure acquire(t, ℓ)
2      C_t.Join(L_ℓ)
3  procedure read(t, x)
4      C_t.Join(LW_x)
5      R_{t,x}.MonotoneCopy(C_t)
6      LRDs_x ← LRDs_x ∪ {t}
7  procedure release(t, ℓ)
8      L_ℓ.MonotoneCopy(C_t)
9  procedure write(t, x)
10     C_t.Join(LW_x)
11     foreach t' ∈ LRDs_x do
12         C_t.Join(R_{t',x})
13     LW_x.MonotoneCopy(C_t)
14     LRDs_x ← ∅
```

is performed is bounded by the number of write-read races in σ between a read and its last write.

5.2 The Mazurkiewicz Partial Order

The Mazurkiewicz partial order (MAZ) [44] has been the canonical way to represent concurrent executions algebraically using an independence relation that defines which events can be reordered. This algebraic treatment allows to naturally lift language-inclusion problems from the verification of sequential programs to concurrent programs [8]. As such, it has been the most studied partial order in the context of concurrency, with deep applications in dynamic analyses [26, 42, 46], ensuring consistency [61] and stateless model checking [27]. In shared memory concurrency, the standard independence relation deems two events as dependent if they conflict, and independent otherwise [29]. In particular, MAZ is the smallest partial order that satisfies the following conditions.

1. $\leq^{\sigma}_{\mathsf{HB}} \subseteq \leq^{\sigma}_{\mathsf{MAZ}}$.
2. for every two events e_1, e_2 such that $e_1 \leq^{\sigma}_{\mathsf{tr}} e_2$ and $e_1 \asymp e_2$, we have $e_1 \leq^{\sigma}_{\mathsf{MAZ}} e_2$.

MAZ with tree clocks. The algorithm for computing MAZ is similar to that for SHB. The main difference is that MAZ includes read-to-write orderings, and thus we need to store additional vector times $\mathbb{R}_{t,x}$ of the last event $\mathsf{r}(x)$ of thread t. In addition, we use the set LRDs_x to store the threads that have executed a $\mathsf{r}(x)$ event after the latest $\mathsf{w}(x)$ event so far. This allows us to only spend computation time in the first read-to-write ordering, as orderings between the read event and later write events follow transitively via intermediate write-to-write orderings. Overall, this approach yields the efficient time complexity $O(n \cdot k)$ for MAZ, similarly to HB and SHB. We refer to Algorithm 5 for the pseudocode.

6 EXPERIMENTS

In this section we report on an implementation and experimental evaluation of the tree clock data structure. The primary goal of these experiments is to evaluate the practical advantage of tree clocks over the vector clocks for keeping track of logical times in a concurrent program executions.

Implementation. Our implementation is in Java and closely follows Algorithm 2. The tree clock data structure is represented as two arrays of length k (number of threads), the first one encoding the shape of the tree and the second one encoding the integer timestamps as in a standard vector clock. For efficiency reasons, recursive routines have been made iterative.

Benchmarks. Our benchmark set consists of standard benchmarks found in benchmark suites and recent literature. In particular, we used the Java benchmarks from the IBM Contest suite [20], Java Grande suite [64], DaCapo [10], and SIR [17]. In addition, we used OpenMP benchmark programs, whose execution lenghts and number of threads can be tuned, from DataRaceOnAccelerator [58], DataRaceBench [34], OmpSCR [18] and the NAS parallel benchmarks [7], as well as large OpenMP applications contained in the following benchmark suites: CORAL [1, 2], ECP proxy applications [3], and Mantevo project [4]. Each benchmark was instrumented and executed in order to log a single concurrent trace, using the tools RV-Predict [55] (for Java programs) and ThreadSanitizer [60] (for OpenMP programs). Overall, this process yielded a large set of 153 benchmark traces that were used in our evaluation. Table 1 presents aggregate information about the benchmark traces generated. Information on the individual traces is provided in our technical report [39].

Table 1: Trace Statistics

	Min	Max	Mean		Min	Max	Mean
Threads	3	222	31	Events	51	2.1B	227M
Locks	1	60.5k	688	Sync. Events (%)	0.0	44.4	9.5
Variables	18	37.8M	1.8M	R/W Events (%)	55.6	100	90.5

Setup. Each trace was processed for computing each of the MAZ, SHB and HB partial orders using both tree clocks and the standard vector clocks. This allows us to directly measure the speedup conferred by tree clocks in computing the respective partial order, which is the goal of this paper.

As the computation of these partial orders is usually the first component of any analysis, in general, we evaluated the impact of the conferred speedup in an overall analysis as follows. For each pair of conflicting events e_1, e_2, we computed whether these events are concurrent wrt the corresponding partial order (e.g., whether $e_1 \parallel^{\sigma}_{\mathsf{HB}} e_2$). This test is performed in dynamic race detection (in the cases of HB and SHB) where such pairs constitute data races, as well in stateless model checking (in the case of MAZ) where the model checker identifies such event pairs and attempts to reverse their order on its way to exhaustively enumerate all Mazurkiewicz traces of the concurrent program. For a fair comparison, in the case of HB we used common epoch optimizations [24] to speed up the analysis for both tree clocks and vector clocks (recall Remark 1). For consistency, every measurement was repeated 3 times and the average time was reported.

Running times. For each partial order, Table 2 shows the average speedup over all benchmarks, both with and without the analysis

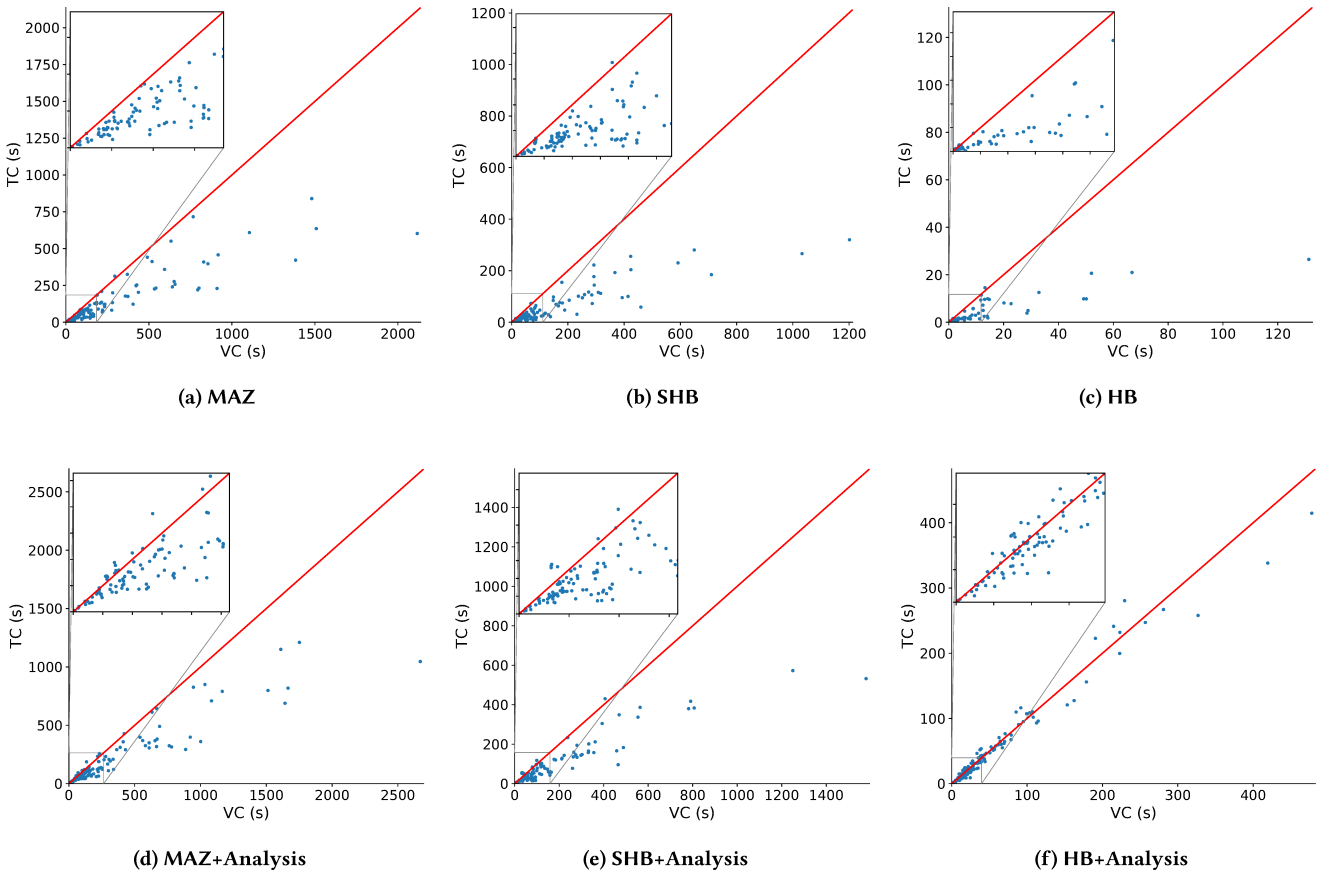


Figure 6: Times for processing each benchmark trace using tree clocks (TC) and vector clocks (VC). The top row shows the time for computing the partial order, while the bottom row shows the time including the analysis component.

component. We see that tree clocks are very effective in reducing the running time of the computation of all 3 partial orders, with the most significant impact being on SHB where the average speedup is 2.53 times. For the cases of MAZ and SHB, this speedup also lead to a significant speedup in the overall analysis time. On the other hand, although HB with tree clocks is about 2 times faster than with vector clocks, this speedup has a smaller effect on the overall analysis time. The reason behind this observation is straightforward: SHB and MAZ are much more computationally-heavy, as they are defined using all types of events; on the other hand, HB is defined only on synchronization events (`acq` and `rel`) and on average, only $\simeq 9.5\%$ of the events are synchronization events on our benchmark traces. Since our analysis considers all events, the HB-computation component occupies a smaller fraction of the overall analysis time. We remark, however, that for programs that are more synchronization-heavy, or for analyses that are more lightweight (e.g., when checking for data races on a specific variable as opposed to all variables), the speedup of tree clocks will be larger on the whole analysis. Indeed, Figure 7 shows the obtained speedup on the total analysis time for HB as a function of synchronization events. We observe a trend for the speedup to increase as the percentage of synchronization events increases in the trace. A further observation is that speedup is prominent when the number of threads are large.

Table 2: Average speedup for computing the partial order due to tree clocks.

	MAZ	SHB	HB
PO	2.02	2.66	2.97
PO + Analysis	1.49	1.80	1.11

Figure 6 gives a more detailed view of the tree clocks vs vector clocks times across all benchmarks. We see that tree clocks almost always outperform vector clocks on all partial orders, and in some cases by large margins. Interestingly, the speedup tends to be larger on more demanding benchmarks (i.e., on those that take more time). In the very few cases tree clocks are slower, this is only by a small factor. These are traces where the sub-linear updates of tree clocks only yield a small potential for improvement, which does not justify the overhead of maintaining the more complex tree data structure (as opposed to a vector). Nevertheless, overall tree clocks consistently deliver a generous speedup to each of MAZ, HB and

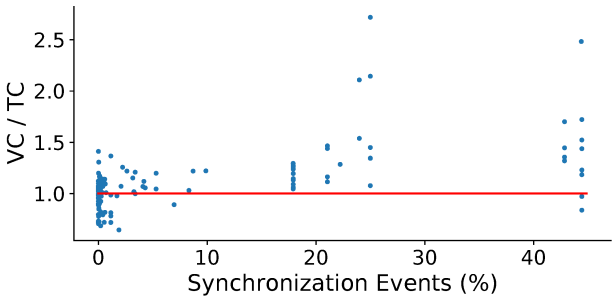


Figure 7: Speedup on HB+analysis computation as a function of the percentage of synchronization events, for the traces where the total time is not too small (≥ 100ms).

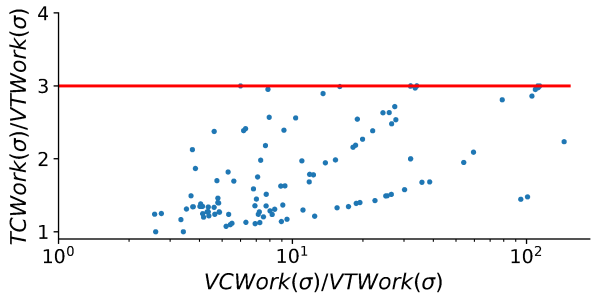


Figure 8: Comparison of the ratios TCWork(σ)/VTWork(σ) **and** VCWork(σ)/VTWork(σ) **across all benchmarks.**

SHB. Finally, we remark that all these speedups come directly from just replacing the underlying data structure, without any attempt to optimize the algorithm that computes the respective partial order, or its interaction with the data structure.

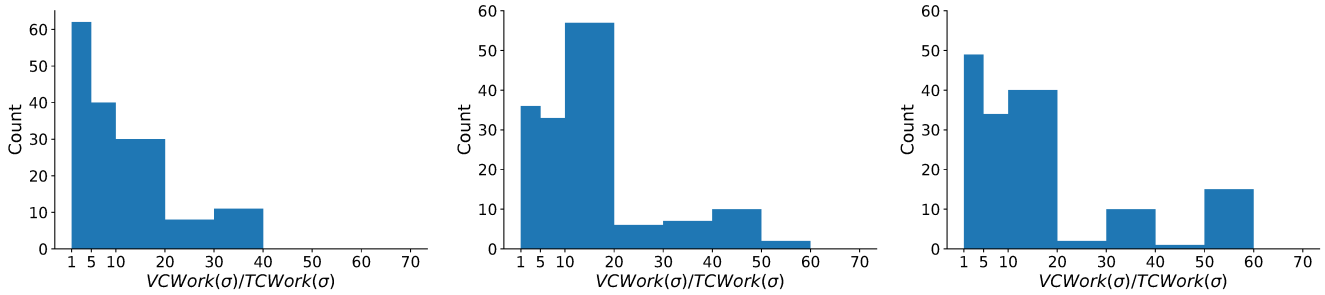
Comparison with vt-work. We also investigate the total number of entries updated using each of the data structures. Recall that the metric VTWork(σ) (Section 4) measures the minimum amount of updates that any implementation of the vector time must perform when computing the HB partial order. We can likewise define the metrics TCWork(σ) and VCWork(σ) corresponding to the number of entries updated when processing each event when using respectively the data structures tree clocks and vector clocks. These metrics are visualized in Figure 8 for computing the HB partial order in our benchmark suite. The figure shows that the VCWork(σ)/VTWork(σ) ratio is often considerably large. In contrast, the ratio TCWork(σ)/VTWork(σ) is typically significantly smaller. The differences in running times between vector and tree clocks reflect the discrepancies between TCWork($\cdot$) and VCWork($\cdot$). Next, recall the intuition behind the optimality of tree clocks (Theorem 1), namely that TCWork(σ) $\leq 3 \cdot$ VTWork(σ). Figure 8 confirms this theoretical bound, as the TCWork(σ)/VTWork (σ) ratio stays nicely upper-bounded by 3 while the VCWork(σ)/VTWork(σ) ratio grows to nearly 100. Interestingly, for some benchmarks we have TCWork(σ) $\simeq 2.99 \cdot$ VTWork(σ), i.e., these benchmarks push tree clocks to their vt-work upper-bound. Going one step further, Figure 9 shows the ratio VCWork(σ)/TCWork(σ) for each partial order in our dataset. The results indicate that vector clocks perform a lot of unnecessary work compared to tree clocks, and experimentally demonstrate the source of speedup on tree clocks. Although Figure 9 indicates that the potential for speedup can be large (reaching 55×), the actual speedup in our experiments (Figure 6) is smaller, as a single tree clock operation is more computationally heavy than a single vector clock operation.

Scalability. To get a better insight on the scalability of tree clocks, we performed a set of controlled experiments on custom benchmarks, by controlling the number of threads and the number of locks parameters while keeping the communication patterns constant. Each trace consists of 10M events, while the number of threads varies between 10 and 360. The traces are generated in a way such that a randomly chosen thread performs two consecutive operations, $acq(l)$ followed by a $rel(l)$, on a randomly (when applicable) chosen lock l. We have considered four cases: (a) all threads communicate over a single common lock (single lock); (b) similar to (a), but there are 50 locks, and 20% of the threads are 5 times more likely to perform an operation compared to the rest of the threads (50 locks, skewed); (c) $k-1$ client threads communicate with a single server thread via a dedicated lock per thread (star topology); (d) similar to (a), but every pair of threads communicates over a dedicated lock (pairwise communication). Figure 10 shows the obtained results. Scenario (a) shows how tree clocks have a constant-factor speedup over vector clocks in this setting. As we move to more locks in scenario (b), thread communication becomes more independent and the benefit of tree clocks may slightly diminish. As a subset of the threads is more active than the rest, timestamps are frequently exchanged through them, making tree clocks faster in this setting as well. Scenario (c) represents a case in which tree clocks thrive: while the time taken by vector clocks increases with the number of threads, it stays constant for tree clocks. This is because the star topology implies that, on average, every tree clock join and copy operation only affects a constant number of tree clock entries, despite the fact that every thread is aware of the state of every other thread. Intuitively, the star communication topology *naturally* affects the shape of the tree to (almost) a star, which leads to this effect. Finally, scenario (d) represents the worst case for tree clocks as all pairs of threads can communicate with each other and the communication is conducted via a unique lock per thread pair. This pattern nullifies the benefit of tree clocks, while their increased complexity results in a general slowdown. However, even in this worst-case scenario, the difference between tree clocks and vector clocks remains relatively small.

7 RELATED WORK

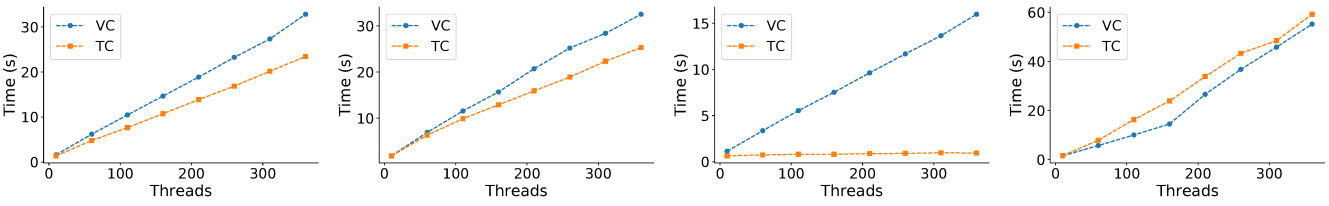
Other partial orders and tree clocks. As we have mentioned in the introduction, besides HB and SHB, many other partial orders perform dynamic analysis using vector clocks. In such cases, tree clocks can replace vector clocks either partially or completely, sometimes requiring small extensions to the data structure as presented here. In particular, we foresee interesting applications of tree clocks for the WCP [31], DC [53] and SDP [28] partial orders.

Speeding up dynamic analyses. Vector-clock based dynamic race detection is known to be slow [56], which many prior works



(a) MAZ (b) SHB (c) HB

Figure 9: Histogram of the ratios $\mathrm{VCWork}(\sigma)/\mathrm{TCWork}(\sigma)$ across all benchmarks in our dataset.



(a) Single lock (b) Fifty locks (c) Star topology (d) Pairwise communication

Figure 10: Comparison of tree clocks (TC) and vector clocks (VC) on four different benchmarks with increasing number of threads.

have aimed to mitigate. One of the most prominent performance bottlenecks is the linear dependence of the size of vector timestamps on the number of threads. Despite theoretical limits [13], prior research exploits special structures in traces [6, 14, 16, 21, 66] that enable succinct vector time representations. The Goldilocks [19] algorithm infers HB-orderings using locksets instead of vector timestamps but incurs severe slowdown [24]. The FastTrack [24] optimization uses epochs for maintaining succinct access histories and our work complements this optimization — tree clocks offer optimizations for other clocks (thread and lock clocks). Other optimizations in clock representations are catered towards dynamic thread creation [50, 51, 68]. Another major source of slowdown is program instrumentation and expensive metadata synchronization. Several approaches have attempted to minimize this slowdown, including hardware assistance [15, 73], hybrid race detection [47, 71], static analysis [25, 52], and sophisticated ownership protocols [12, 54, 69].

8 CONCLUSION

We have introduced tree clocks, a new data structure for maintaining logical times in concurrent executions. In contrast to standard vector clocks, tree clocks can dynamically capture communication patterns in their structure and perform join and copy operations in sublinear time, thereby avoiding the traditional overhead of these operations when possible. Moreover, we have shown that tree clocks are vector-time optimal for computing the HB partial order, performing at most a constant factor work compared to what is absolutely necessary, in contrast to vector clocks. Finally, our experiments show that tree clocks effectively reduce the running time for computing the MAZ, SHB and HB partial orders significantly, and thus offer a promising alternative over vector clocks.

Interesting future work includes incorporating tree clocks in an online analysis such as ThreadSanitizer [60]. Any use of additional synchronization to maintain analysis atomicity in this online setting is identical and of the same granularity to both vector clocks and tree clocks. However, the faster joins performed by tree clocks may lead to less congestion compared to vector clocks, especially for partial orders such as SHB and MAZ where synchronization occurs on all events (i.e., synchronization, as well as access events). We leave this evaluation for future work. Finally, since tree clocks are a drop-in replacement of vector clocks, most of the existing techniques that minimize the slowdown due to metadata synchronization (Section 7) are directly applicable to tree clocks.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive feedback on an earlier draft of this manuscript. Umang Mathur was partially supported by the Simons Institute for the Theory of Computing. Mahesh Viswanathan is partially supported by grants NSF SHF 1901069 and NSF CCF 2007428.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains all the source codes and experimental data for replicating our evaluation in Section 6. We implemented the analyses programs as part of the tool RAPID [36]. The provided experimental data contains all the 153 trace logs used in our evaluation. In our artifact we also provide Python scripts that fully automate the process of replicating our evaluation.

A.2 Artifact Check-List (Meta-Information)

- **Algorithm:** Tree Clock
- **Data set:** Trace logs obtained from the benchmarks described in Section 6.
- **Metrics:** Execution time.
- **Output:** CSV files and graphs (optional).
- **How much disk space required (approximately)?:** 150 GB.
- **How much time is needed to prepare workflow (approximately)?:** We provide all the scripts that automate our workflow.
- **How much time is needed to complete experiments (approximately)?:** Replicating all the results: 15 days (without parallelization). Replicating a small set of results: 1 day (without parallelization). We also provide instructions for parallelizing the computation (see Section A.6).
- **Publicly available?:** Yes [38].
- **Code licenses (if publicly available)?:** MIT License.
- **Data licenses (if publicly available)?:** None.
- **Archived (provide DOI)?:** doi.org/10.5281/zenodo.5749092

A.3 Description

A.3.1 How to Access. Obtain the artifact from [38]. The total size is expected to be approximately 50 MB.

A.3.2 Hardware Dependencies. Replicating the results of large benchmarks requires up to 60 GB RAM. Otherwise, there are no special hardware requirements for using our artifact.

A.3.3 Software Dependencies. Java 11, Ant 1.10 or higher, Python 3.7 or higher, including the packages pandas and matplotlib.

A.3.4 Data Sets. The trace logs are available for download at [5].

A.4 Installation

Obtain the artifact (see Section A.3.1), extract the archive files and set the `$AE_HOME` environment variable:

```
> export AE_HOME=/path/to/AE
```

Next, install RAPID:

```
> cd $AE_HOME/rapid/
> ant jar
```

Then, download the benchmark traces (see Section A.3.4) into the folder `$AE_HOME/benchmarks/`.

A.5 Experiment Workflow

In Figure 11 we display the directory structure of our artifact. The directory `rapid` contains the RAPID tool which includes our implementation of the tree clock and vector clock data structures and the analyses programs based on HB, SHB and MAZ partial orders. The directory `benchmarks` is designated for the trace logs. The directory `scripts` contains a collection of helper scripts that automate our workflow. In particular, the script `$AE_HOME/scripts/run.py` can be utilized to automate the process of replicating the results of Section 6. In Section A.6 we describe how the script can be used to replicate all or part of our experimental evaluation. In addition, Section A.7 contains instructions on how the script can be used to evaluate a new trace log that is not part of the original benchmark set. The `README.md` file provides more comprehensive information on certain aspects of our artifact.

```
AE_HOME/
|--- rapid/
|--- benchmarks/
|--- scripts/
|--- results/
|--- LICENSE.txt
|--- README.md
```

Figure 11: Directory structure of the artifact

A.6 Evaluation and Expected Results

Executing the following command will run all the analyses on all the trace logs:

```
> python $AE_HOME/scripts/run.py -b all
```

The outputs of the executions will be extracted as CSV files under the folder `$AE_HOME/results/`. Note that this command expects to locate the benchmarks used in our evaluation (see Section A.3.4) under the folder `$AE_HOME/benchmarks/`.

The main goal of this evaluation is to measure the performance benefits of tree clocks over vector clocks for keeping track of logical times in concurrent programs. We expect that the overall speedup would remain similar to the speedups reported in Table 2 for each category. After the CSV output files have been generated, the script `$AE_HOME/scripts/compute_averages.py` may be utilized to compute the average speedup for each category and replicate the Table 2:

```
> python $AE_HOME/scripts/compute_averages.py
$AE_HOME/results/
```

This script expects the path to the results folder as argument and outputs a file named `table2.csv` under the same folder which corresponds to replication of Table 2. Similarly, the script `$AE_HOME/scripts/plot.py` can be utilized to visualize the obtained outputs and replicate the Figure 6:

```
> python $AE_HOME/scripts/plot.py $AE_HOME/results/
```

This script also expects the path to the results folder as argument and outputs the plot files under the folder `$AE_HOME/results/plots` which corresponds to replication of Figure 6.

We remark that, as also indicated in Section A.2, replicating the whole evaluation can take very long if executed serially. We refer the interested readers to the file `$AE_HOME/README.md` where we describe a procedure which may be utilized to parallelize the evaluation. Furthermore, the script `$AE_HOME/scripts/run.py` also provides an option to replicate only parts of our experimental evaluation. The following command runs the analyses on a small set of benchmarks which require moderate system resources and reduced computation time:

```
> python $AE_HOME/scripts/run.py -b small
```

We refer the readers to the `$AE_HOME/README.md` file for more detailed information on customizing the experiments.

A.7 Experiment Customization

Users might utilize the script `$AE_HOME/scripts/run.py` to evaluate a new trace log that is not part of our original benchmark set. This can be achieved with the following command:

```
> python $AE_HOME/scripts/run.py -p path/to/trace -n output-folder-name
```

The above command will run all the analyses on the input trace located in `path/to/trace` and extract the output CSV files into `$AE_HOME/results/output-folder-name`. Note that the given input trace must be in one of the formats admitted by the RAPID tool. Readers may refer to the `$AE_HOME/rapid/README.md` file for information regarding the formats admitted by RAPID.

A.8 Notes

We note that the reported execution times correspond to the time taken for performing the respective analyses and do not include the time taken for processing the input files. Hence, the actual execution times are expected to be longer than the reported times.

REFERENCES

[1] [n.d.]. CORAL-2 Benchmarks. https://asc.llnl.gov/coral-2-benchmarks. Accessed: 2021-08-01.

[2] [n.d.]. CORAL Benchmarks. https://asc.llnl.gov/coral-benchmarks. Accessed: 2021-08-01.

[3] [n.d.]. ECP Proxy Applications. https://proxyapps.exascaleproject.org. Accessed: 2021-08-01.

[4] [n.d.]. Mantevo Project. https://mantevo.org. Accessed: 2021-08-01.

[5] 2022. Trace logs used in Section 6. https://uillinoisedu-my.sharepoint.com/:f:/g/personal/umathur3_illinois_edu/EskC1fg2xhNHnim2ZYjDD9gBJqme8hBTgWShHUmOfYmF-Q.

[6] Kunal Agrawal, Joseph Devietti, Jeremy T. Fineman, I-Ting Angelina Lee, Robert Utterback, and Changming Xu. 2018. Race Detection and Reachability in Nearly Series-Parallel DAGs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (New Orleans, Louisiana) *(SODA '18)*. Society for Industrial and Applied Mathematics, USA, 156–171. https://doi.org/10.1137/1.9781611975031.11

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

[8] A. Bertoni, G. Mauri, and N. Sabadini. 1989. Membership problems for regular and context-free trace languages. *Information and Computation* 82, 2 (1989), 135–150. https://doi.org/10.1016/0890-5401(89)90051-5

[9] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 28–39. https://doi.org/10.1145/2594291.2594323

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

[11] Hans-J. Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism* (Berkeley, CA) *(HotPar'11)*. USENIX Association, USA, 3. http://dl.acm.org/citation.cfm?id=2001252.2001255

[12] Michael D. Bond, Milind Kulkarni, Man Cao, Minjia Zhang, Meisam Fathi Salmi, Swarnendu Biswas, Aritra Sengupta, and Jipeng Huang. 2013. OCTET: Capturing and Controlling Cross-Thread Dependences Efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 693–712. https://doi.org/10.1145/2509136.2509519

[13] Bernadette Charron-Bost. 1991. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* 39, 1 (1991), 11 – 16. https://doi.org/10.1016/0020-0190(91)90055-M

[14] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting Data Races in Cilk Programs That Use Locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. ACM, New York, NY, USA, 298–309. https://doi.org/10.1145/277651.277696

[15] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. 2012. RADISH: Always-on Sound and Complete Race Detection in Software and Hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 201–212. https://doi.org/10.1109/ISCA.2012.6237018

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

[17] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

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

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

[20] 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–. http://dl.acm.org/citation.cfm?id=838237.838485

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

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

[23] Colin J. Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proc. 11th Australian Comput. Science Conf.* 56–66.

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

[25] Cormac Flanagan and Stephen N. Freund. 2013. RedCard: Redundant Check Elimination for Dynamic Race Detectors. In *ECOOP 2013 – Object-Oriented Programming*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 255–280. https://doi.org/10.1007/978-3-642-39038-8_11

[26] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 293–303. https://doi.org/10.1145/1375581.1375618

[27] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

[28] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 179 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360605

[29] Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and Pierre Wolper. 1996. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg.

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

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

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

[33] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

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

[35] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. ACM, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[36] Umang Mathur. 2019. RAPID. https://github.com/umangm/rapid. Accessed: 2021-08-01.

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

[38] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2021. *Artifact for "A Tree Clock Data Structure for Causal Orderings in Concurrent Executions"*. https://doi.org/10.5281/zenodo.5749092

[39] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. arXiv:2201.06325 [cs.LO]

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

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

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

[43] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, M. Cosnard et. al. (Ed.). Elsevier Science Publishers B. V., 215–226.

[44] 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. https://doi.org/10.1007/3-540-17906-2_30

[45] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

[46] Robert H.B. Netzer and Barton P. Miller. 1990. On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions. In *In Proceedings of the 1990 International Conference on Parallel Processing*. 93–97.

[47] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. *SIGPLAN Not.* 38, 10 (June 2003), 167–178. https://doi.org/10.1145/966049.781528

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

[49] Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. *SIGPLAN Not.* 38, 10 (June 2003), 179–190. https://doi.org/10.1145/966049.781529

[50] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Efficient data race detection for async-finish parallelism. *Formal Methods in System Design* 41, 3 (01 Dec 2012), 321–347. https://doi.org/10.1007/s10703-012-0143-7

[51] Veselin Raychev, Martin Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 151–166. https://doi.org/10.1145/2509136.2509538

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

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

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

[55] Grigore Rosu. 2021. RV-Predict, Runtime Verification. https://runtimeverification.com/predict/. Accessed: 2021-08-01.

[56] 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* (Portland, Oregon, USA) *(PLATEAU '14)*. Association for Computing Machinery, New York, NY, USA, 43–51. https://doi.org/10.1145/2688204.2688205

[57] Malavika Samak and Murali Krishna Ramanathan. 2014. Trace Driven Dynamic Deadlock Detection and Reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 29–42. https://doi.org/10.1145/2555243.2555262

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

[59] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed computing* 7, 3 (1994), 149–174. https://doi.org/10.1007/BF02277859

[60] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[61] Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

[62] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do I Use the Wrong Definition?: DeFuse: Definition-use Invariants for Detecting Concurrency and Sequential Bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 160–174. https://doi.org/10.1145/1869459.1869474

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

[64] L. A. Smith, J. M. Bull, and J. Obdrzálek. 2001. A Parallel Java Grande Benchmark Suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (Denver, Colorado) *(SC '01)*. ACM, New York, NY, USA, 8–8. https://doi.org/10.1145/582034.582042

[65] Martin Sulzmann and Kai Stadtmüller. 2018. Two-Phase Dynamic Analysis of Message-Passing Go Programs Based on Vector Clocks. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming* (Frankfurt am Main, Germany) *(PPDP '18)*. Association for Computing Machinery, New York, NY, USA, Article 22, 13 pages. https://doi.org/10.1145/3236950.3236959

[66] Rishi Surendran and Vivek Sarkar. 2016. Dynamic Determinacy Race Detection for Task Parallelism with Futures. In *Runtime Verification*, Yliès Falcone and César Sánchez (Eds.). Springer International Publishing, Cham, 368–385. https://doi.org/10.1007/978-3-319-46982-9_23

[67] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International*

Conference on Architectural Support for Programming Languages and Operating Systems (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 865–878. https://doi.org/10.1145/3297858.3304069

[68] Xinli Wang, J. Mayo, W. Gao, and J. Slusser. 2006. An Efficient Implementation of Vector Clocks in Dynamic Systems. In *PDPTA*.

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

[70] Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. 2021. Detecting Concurrency Vulnerabilities Based on Partial Orders of Memory and Thread Events. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 280–291. https://doi.org/10.1145/3468264.3468572

[71] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct. 2005), 221–234. https://doi.org/10.1145/1095809.1095832

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

[73] P. Zhou, R. Teodorescu, and Y. Zhou. 2007. HARD: Hardware-Assisted Lockset-based Race Detection. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 121–132. https://doi.org/10.1109/HPCA.2007.346191