CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis

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

Ameya Prashant Deshmukh
IIT Bombay
India
ameyapd@cse.iitb.ac.in

Berk Çirisci
Amazon Web Services
Germany
cirisci@amazon.de

Constantin Enea
Université de Paris
France
cenea@irif.fr

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

Abstract

Dynamic analyses are a standard approach to analyzing and testing concurrent programs. Such techniques observe program traces σ and analyze them to infer the presence or absence of bugs. At its core, each analysis maintains a partial order P that represents order dependencies between the events of σ . Naturally, the scalability of the analysis largely depends on maintaining P efficiently. The standard data structure for this task has thus far been Vector Clocks. These, however, are slow for analyses that follow a non-streaming style, costing $O(n)$ time for inserting (and propagating) each new ordering in P, where n is the size of σ, while they cannot handle the deletion of existing orderings.

In this paper we develop *Collective Sparse Segment Trees* (*CSSTs*), a simple but elegant data structure for maintaining a partial order P. CSSTs thrive when the width k of P is much smaller than the size n of its domain, allowing inserting, deleting, and querying for orderings in P to run in $O(\log n)$ time. For a concurrent trace, k normally equals the number of its threads, and is orders of magnitude smaller than its size n, making CSSTs fitting for this setting. Our experiments confirm that CSSTs are the best data structure currently to handle a range of dynamic analyses from existing literature.

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

Keywords: concurrency, happens-before, vector clocks, dynamic concurrency analyses, dynamic reachability

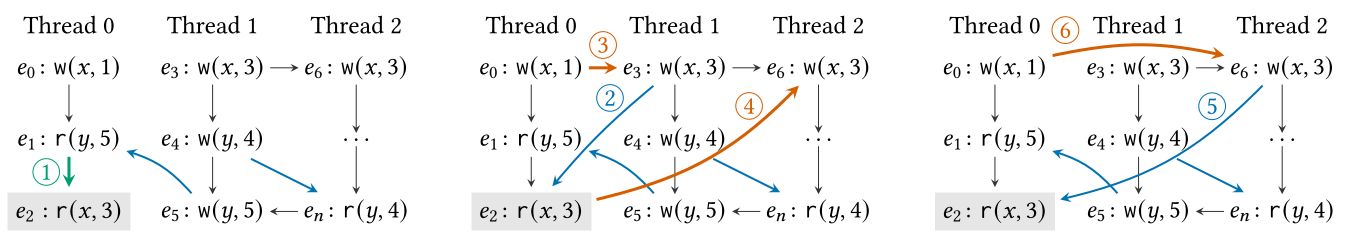
ACM Reference Format:
Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Çirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620666.3651358

1 Introduction

Concurrent programming is notoriously difficult due to the inherent nondeterminism in interprocess communication [29]. As such, considerable efforts go towards effective techniques for analyzing concurrent programs and testing them for correctness. Dynamic analyses are one of the most widespread types of such techniques. Their main working principle is to discover software faults by analyzing concrete program executions. Prominent examples include concurrency bugs such as data races [13, 15, 17, 23, 24, 26, 31–33], deadlocks [8, 20, 37], linearizability [12] and atomicity violations [6, 8, 16, 24], thread-safety [22], and memory violations [19, 40], to name a few.

Although each analysis employs reasoning that is specific to the problem at hand, one of its core components is nearly always the construction of a partial order P (often termed generically as "happens-before" [21]) that captures order dependencies between events of the analyzed trace σ. The analysis undergoes a sequence of interleaved operations of (i) *inserting* new orderings $e_1 \longrightarrow e_2$ in P, (ii) *querying* whether $e_1 \longrightarrow e_2$ in P, and even (iii) *deleting* existing orderings $e_1 \longrightarrow e_2$ from P, as it attempts to prove that σ has certain properties (e.g., that it is sequentially-consistent, or it contains a data race). To benefit scalability, each operation must take as little time as possible, as both the number of traces and the size of each trace normally span several orders of magnitude.

Consider analyzing a trace σ of size n. Representing P as a graph G, the above setting is colloquially known as *dynamic graph reachability*, with dynamically inserting/deleting edges and performing reachability queries. Such queries are handled in $O(1)$ time, at the cost of $O(n^2)$ for keeping G



(a) Initial partial order. **(b)** First update. **(c)** Delete and update.

Figure 1. A consistency analysis. Blue edges represent reads-from information. Numbered edges represent new orderings inserted by the analysis.

transitively closed after each update. This bound is prohibitively large for typical values of n in the range 10^4-10^{10}.

The scalability of dynamic analyses has been driven primarily by a simple data structure known as *Vector Clocks* [28], based on the realization that (i) σ consists of a small number of threads k, and (ii) normally, P contains k totally ordered chains (or a number proportional to k). Consequently, the whole backward set of an event e can be compactly summarized as an integer array, i.e., a Vector Clock. This allows to replace one factor n in the update complexity by the much smaller k, i.e., a single edge insertion now takes $O(nk)$ time.

In many analyses that have a *streaming nature*, the insertion cost of Vector Clocks is further reduced to $O(k)$. Here the events of σ are processed one by one, and new orderings $e_1 \rightarrow e_2$ are always inserted where e_2 is the current event being processed. Popular examples include analyses for detecting data-races [15, 24, 33], deadlocks [37], and atomicity violations [27], as well as consistency checking in some weak-memory models [36]. Thus in the streaming setting, Vector Clocks are arguably the best data structure to represent a partial order. On the other hand, many analyses are *non-streaming*, i.e., as the analysis progresses, it might insert new orderings between arbitrary events of σ. These new orderings might have to be propagated across n events, recovering the $O(nk)$ bound, which can lead to a significant slowdown. A natural question thus arises: *can the $O(nk)$ cost be reduced further?* In general, *what is the best data structure for non-streaming concurrent execution analysis?*

Here we develop a simple but elegant data structure for this general setting, called *Collective Sparse Segment Trees* (CSSTs). Like Vector Clocks, CSSTs exploit the fact that $k \ll n$, and offer update and querying operations that run roughly in $O(\log n)$ time, thereby reducing the prohibitive linear dependency on n to logarithmic. We argue that, as Vector Clocks are the most fitting data structure for handling *streaming* dynamic analyses, CSSTs are the most fitting data structure for handling non-streaming analyses.

1.1 Motivating Example

Consider the analysis of an abstract trace shown in Figure 1, consisting of three threads and write/read events w(x,v)/r(x,v), where x is a variable and v is a value. The consistency-testing problem asks whether there is an interleaving that is consistent with the values observed by each read [18], and has numerous applications in dynamic concurrency analyses [4, 7, 9, 10, 14, 19, 20, 23, 34, 37]. The analysis maintains (i) a reads-from map w$(x,v) \mapsto$ r(x,v), denoting that r(x,v) obtains its value from w(x,v), and (ii) a partial order P that is induced by this reads-from map and any additional indirect orderings this might impose.

Assume that the consistency analysis has, so far, established the partial order P in Figure 1a, with blue edges marking the current reads-from map. The analysis proceeds on the read event e_2 : r$(x,3)$. It first inserts in P the program order edge ①. As e_2 has no successors in P, this edge need not be propagated further. Now the analysis must decide which of the two writes e_3 and e_6 can be observed by e_2. In general, there is no efficient procedure for deciding this [18], and the consistency algorithm proceeds by trying each option.

In Figure 1b, the analysis tries to map $e_3 \mapsto e_2$, leading to inserting ② in P. Moreover, since $e_0 \rightarrow e_2$ in P, the analysis also inserts ③ and ④ in P. These are necessary orderings to respect the fact that e_2 reads from e_3 — the process of inferring such orderings is known as saturation, and is used widely in dynamic analyses (e.g., [2, 23, 31, 33, 34, 41, 42]). Crucially, both e_3 and e_6 have many successors in P, thus these edges must be propagated along many events, costing $O(n)$ time, even if P is represented using Vector Clocks. In contrast, CSSTs achieve a much smaller $O(\log n)$ bound.

The reads-from choice $e_3 \mapsto e_2$ is inconsistent due to the cycle $e_2 \rightarrow e_6 \rightarrow \cdots \rightarrow e_n \rightarrow e_5 \rightarrow e_1 \rightarrow e_2$. The analysis must delete the orderings ②, ③ and ④, before proceeding with an alternative writer for e_2. When P is represented via Vector Clocks, there is no efficient method for deletion, and essentially it must be recomputed from scratch, taking $O(n^2)$ time. In contrast, CSSTs only need $O(\log n)$ time per edge deletion, thereby completing this task efficiently.

Finally, in Figure 1c, the analysis tries to map $e_6 \mapsto e_2$, leading to inserting ⑤ in P. It also infers ordering ⑥, which again must be propagated further, a task that costs $O(n)$ time for Vector Clocks but only $O(\log n)$ time for CSSTs.

1.2 Contributions

We develop CSSTs, a new data structure for maintaining and querying partial orders of small width. Concretely, consider the maintenance of a DAG G, representing a partial order P over n events and k totally ordered chains (normally k equals the number of threads).

1. CSSTs support fully dynamic updates (inserting and deleting edges) in $O(\max(\log\delta, \min(\log n, d)))$ time, where δ is the maximum degree in G and d is the *cross-chain density* of G, a novel notion we introduce here that captures a type of sparsity prevalent in practice. Normally, δ is constant and the above expression results in $O(d)$. Reachability (ordering) queries take $O(k^3\min(\log n, d))$ time.
2. We further optimize CSSTs for the purely incremental setting, supporting the insertion, but not deletion, of edges. Incremental CSSTs run updates and queries in time $O(k^2\min(\log n, d))$ and $O(\min(\log n, d))$, respectively,
3. We make an extensive experimental evaluation of CSSTs in concurrency analyses from the literature, spanning data-race detection [31], deadlock detection [8], memory bugs [40], consistency checking for X86-TSO [34], use-after-free bugs [19], data-races in the C11 memory model [23], and root cause analysis of linearizability violations [12]. Our results show that CSSTs speed up the corresponding analysis in most cases, indicating that they are the most fitting data structure in this ubiquitous setting.

CSSTs are based on the insight that dynamic reachability on partial orders of small width can be reduced to a basic algorithmic problem known as *dynamic suffix minima*. This idea was partly explored in a data structure for incremental reachability underpinning the M2 data-race detector [31]. However, CSSTs are more advanced, as (i) they employ techniques making them faster in all cases, (ii) they achieve tighter time and space bounds on sparse instances, and (iii) they support fully dynamic updates (i.e., both incremental and decremental). Our experimental results confirm that CSSTs are indeed more efficient in both time and memory, besides being broader (as they handle edge deletions). In our technical report [39], we provide the proofs for the theorems and lemmas presented in the paper.

2 Preliminaries

In this section, we develop some general notation and introduce standard concepts on concurrent executions and partial orders that will be used throughout the paper.

2.1 Concurrent Execution Model

Events and traces. We broadly consider traces σ of concurrent programs, representing the history of operations in an execution. We write Events_σ to denote the set of events occurring in σ. Each event of σ is a tuple $e = \langle t, i, m\rangle$, where t is an identifier of the thread that performed e and i is the sequence id of e. The pair $\langle t, i\rangle$ serves as a unique identifier for e. The field m represents meta information for e that might be relevant to a dynamic analysis, but is immaterial for CSSTs. E.g., m may record the operation performed by e (read, write, etc), as well as the variable it accesses and the value it acquires. CSSTs operate only on the identifier $\langle t, i\rangle$, while the meta information m will be used only in our examples. We write $\mathsf{tid}(e)$, $\mathsf{id}(e)$ for the thread identifier and sequence id of e, respectively. Depending on the application, the events in σ may or may not be totally ordered.

Relations and functions on execution traces. For a binary relation X, we denote by X^* its reflexive transitive closure. A partial order is a reflexive, transitive, and anti-symmetric binary relation defined on the elements of a given set S. We denote by $\leq^\sigma_{\mathsf{P}}$ a partial order P over the set Events_σ. Given $e_i, e_j \in \mathsf{Events}_\sigma$, we write $e_i \leq^\sigma_{\mathsf{P}} e_j$ to represent $(e_i, e_j) \in \leq^\sigma_{\mathsf{P}}$. We write $e_i <^\sigma_{\mathsf{P}} e_j$ to represent $e_i \leq^\sigma_{\mathsf{P}} e_j$ and $e_i \neq e_j$. The *program order* $\leq^\sigma_{\mathsf{po}}$ represents a partial order where $e_i \leq^\sigma_{\mathsf{po}} e_j$ iff $\mathsf{id}(e_i) \leq \mathsf{id}(e_j)$ and $\mathsf{tid}(e_i) = \mathsf{tid}(e_j)$. We write $\mathtt{width}(P)$ to represent the largest size of a set $S \subseteq \mathsf{Events}_\sigma$ such that for all distinct pairs $e_i, e_j \in S$, we have that $e_i \not\leq^\sigma_{\mathsf{P}} e_j$ and $e_j \not\leq^\sigma_{\mathsf{P}} e_i$.

In concurrency analyses, $\mathtt{width}(P)$ is almost always bounded by some constant, a fact that is exploited by Vector Clocks, as well as the CSSTs we introduce in this work. E.g., in many analyses $\leq^\sigma_{\mathsf{po}} \subseteq \leq^\sigma_{\mathsf{P}}$, implying that $\mathtt{width}(P) \leq k$, where k is the number of threads. Although in other settings (e.g., those involving weak-memory concurrency) $\mathtt{width}(P)$ might be larger than k, it still remains bounded by some small factor of k. For example, analyses involving TSO normally satisfy $\mathtt{width}(P) \leq 2k$ (we touch on this later in Section 5).

2.2 Dynamic Reachability on DAGs

Chain directed acyclic graphs. We consider directed acyclic graphs (DAGs) $G = (V, E)$, where V is the set of nodes and E is the set of edges. Dynamic analyses use such DAGs to represent partial orders on n events of a concurrent execution, where k is normally the number of threads, or a value proportional to that. For this purpose, a node in V is a pair $u = \langle t, i\rangle \in [k] \times [n]$, where $[j] = \{0, 1, \dots, j-1\}$. The set of edges normally captures the program order, which is reflected in the fact that for any two nodes $\langle t, i\rangle, \langle t, i+1\rangle \in V$, we have $(\langle t, i\rangle, \langle t, i+1\rangle) \in E$. We can thus represent G as k totally-ordered chains with additional edges across chains, and call such graphs *chain DAGs*. A node $\langle t, i\rangle$ is said to be *earlier* than another node in the same chain $\langle t, j\rangle$ if $i \leq j$, and *later* otherwise. Then, G can be written as the composition of $k(k-1)$ subgraphs $\mathsf{Sub(G)}^{t_2}_{t_1}$ induced by the chains t_1 and t_2, as well as the cross-chain edges from t_1 to t_2 (see Figure 2). The *cross-chain density* d of G is the maximum, among all chains t_1, number of nodes in chain t_1 that have an outgoing edge to some other chain.

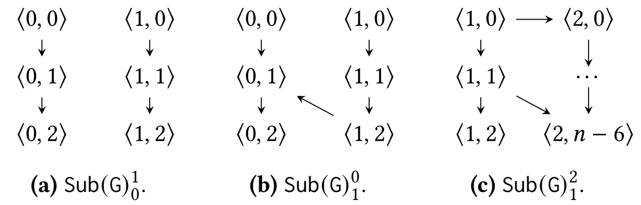


Figure 2. Three subgraphs of the chain DAG G formed from the execution in Figure 1a. The cross-chain density of G is 2, as chain 1 (i.e., thread 1 in Figure 1a) has two outgoing edges to chain 2 (from events e_3 and e_4).

Dynamic reachability. The dynamic reachability problem is defined with respect to a given chain DAG $G = (V, E)$ and online sequence of operations of the following types.

1. An `insertEdge`(u, v) operation, such that $(u, v) \notin E$, inserts the edge (u, v) in G.
2. An `deleteEdge`(u, v) operation, such that $(u, v) \in E$, deletes the edge (u, v) in G.
3. A `reachable`(u, v) operation returns True iff $u \longrightarrow^* v$.
4. A `successor`(u, t) operation returns the earliest successor of u in the t-th chain.
5. A `predecessor`(u, t) operation returns the latest predecessor of u in the t-th chain.

The operations `insertEdge` and `deleteEdge` are updates, while `reachable`, `successor` and `predecessor` are queries. We allow update operations only across nodes in different chains. In this dynamic setting, we define the cross-chain density of G to be the maximum cross-chain density of the graph across all update operations on G.

3 Collective Sparse Segment Trees

We now present *Collective Sparse Segment Trees* (CSSTs), a data structure for efficiently solving the dynamic reachability problem in the context of concurrent execution analysis. The properties of CSSTs are stated in the following theorem.

Theorem 1. *Consider a chain DAG G of n nodes, k chains, maximum out-degree δ and cross-chain density d. CSSTs maintain G under dynamic updates, costing $O(\max(\log \delta, \min(\log n, d)))$ time per update and $O(k^3 \min(\log n, d))$ time per query.*

CSSTs are based on the insight that dynamic reachability on DAGs with few chains can be efficiently reduced to repeated instances of another basic problem known as *dynamic suffix minima.* We begin by developing this concept when G consists of only $k = 2$ chains in Section 3.1. Dynamic suffix minima are solved using a data structure known as *Segment Trees.* A key novelty of CSSTs is based on the realization that most direct orderings in dynamic analyses make G have low cross-chain density (i.e., the cross-chain edges are sparse). We exploit this insight in Section 3.2, by developing *Sparse Segment Trees* that handle arbitrary graphs, but become more efficient the lower their cross-chain density becomes. Finally, in Section 3.3 we handle DAGs of arbitrarily many chains k, by appropriately maintaining *collections* of Sparse Segment Trees, which also explains the name of the data structure.

3.1 Dynamic Suffix Minima and Segment Trees

Here we state the algorithmic problem of dynamic suffix minima, and relate it to dynamic reachability on chain DAGs, focusing on the special case of $k = 2$ chains. In later sections we explain how to handle $k > 2$. Some insights are based on observations made in [31] for the purely incremental case.

Dynamic suffix minima. The dynamic suffix minima problem concerns maintaining an array A of n values in $\mathbb{N} \cup \{\infty\}$ under an online sequence of updates, and answering minima queries on ranges of A. We write $\mathtt{A}[i]$ and $\mathtt{A}[i : j]$ to denote the value of A at index i and the sequence of values $\mathtt{A}[i], \dots, \mathtt{A}[j]$, respectively, and write $\mathtt{A}[i :]$ as shorthand for $\mathtt{A}[i : |\mathtt{A}| - 1]$. We say that A is empty on index i if $\mathtt{A}[i] = \infty$. The density of A is the number of non empty entries of A.

For $i \in [n]$ and $a \in \mathbb{N} \cup \{\infty\}$, an operation on A is one of:

1. $\mathtt{min}(\mathtt{A}, i)$, returning $\mathtt{min}_{i \le j < n} \mathtt{A}[j]$, i.e., the minimum value in the suffix $\mathtt{A}[i :]$;
2. $\mathtt{argleq}(\mathtt{A}, a)$, returning $\max_{i : \mathtt{A}[i] \le a} i$, i.e., the largest index i of A such that $\mathtt{A}[i] \le a$; and
3. $\mathtt{update}(\mathtt{A}, i, a)$, setting $\mathtt{A}[i] = a$.

The task is to answer `min` and `argleq` queries, taking into consideration all preceding `update` operations.

Dynamic suffix minima (in fact, the more general problem of dynamic *range* minima) is solved efficiently using Segment Trees (STs), taking $O(\log n)$ time per operation (see, e.g., [5]). A ST T represents A as a binary tree of nodes $\mathtt{nd}_{i,j}$ each associated with a range $[i, j]$. Each $\mathtt{nd}_{i,j}$ maintains a value $\mathtt{nd}_{i,j}.\mathtt{min} = \min(\mathtt{A}[i : j])$. We write $\mathtt{depth}(\mathtt{nd}_{i,j})$ to denote the depth of $\mathtt{nd}_{i,j}$ in T. We let $T.\mathtt{root}$ be the root node of T. Each query is answered by traversing T from the root to a leaf, the latter containing the value to be returned.

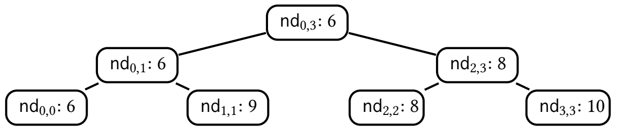
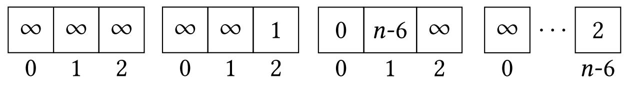


Figure 3. A Segment Tree representation of $A = [6, 9, 8, 10]$.

Example 1. *Consider the Segment Tree T in Figure 3 representing the array $A = [6, 9, 8, 10]$. We have the following.*

- $\mathtt{min}(A, 0) = 6$, $\mathtt{min}(A, 1) = \mathtt{min}(A, 2) = 8$, $\mathtt{min}(A, 3) = 10$
- $T.\mathtt{root.min} = 6$, $\mathtt{nd}_{2,3}.\,\mathtt{min} = 8$
- $\mathtt{argleq}(A, 7) = 0$, $\mathtt{argleq}(A, 9) = 2$, $\mathtt{argleq}(A, 11) = 3$
- $\mathtt{update}(A, 3, 7)$ *sets* $A[3] = 7$. *In T, this is captured by setting* $\mathtt{nd}_{2,3}.\mathtt{min} = \mathtt{nd}_{3,3}.\mathtt{min} = 7$.

Dynamic reachability on $k = 2$ chains. Recall that a chain DAG G of k chains is the composition of $k(k - 1)$ subgraphs $\mathtt{Sub(G)}^{t_2}_{t_1}$ (Figure 2). Now assume that $k = 2$. We represent each $\mathtt{Sub(G)}^{t_2}_{t_1}$ as an array $\mathtt{A}^{t_2}_{t_1} : [n] \to [n] \cup \infty$. $\mathtt{A}^{t_2}_{t_1}[j_1]$ stores



(a) Array A_0^1. **(b)** Array A_1^0. **(c)** Array A_1^2. **(d)** Array A_2^1.

Figure 4. Representation of the partial order in Figure 1a.

a unique neighbor of node $\langle t_1, j_1\rangle$ in chain t_2. What if $\langle t_1, j_1\rangle$ has multiple neighbors to t_2? Then, it suffices to store the *earliest neighbor* of $\langle t_1, j_1\rangle$ in t_2. Although this is not an exact representation of G, it captures exactly all reachability in G. In particular, we maintain the invariant

$$\mathsf{A}_{t_1}^{t_2} = \min(\{j_2 : \langle t_1, j_1\rangle \to \langle t_2, j_2\rangle\}) \qquad (1)$$

under the standard convention that $\min(\emptyset) = \infty$. A query $\mathtt{successor}(\langle t_1, j_1\rangle, t_2)$ simply returns the suffix minima $x = \mathtt{min}(\mathsf{A}_{t_1}^{t_2}, j_2)$, while a query $\mathtt{predecessor}(\langle t_2, j_2\rangle, t_1)$ returns the suffix arg-minima $\mathtt{argleq}(\mathsf{A}_{t_1}^{t_2}, j_2)$. For reachability queries, observe that we have $\langle t_1, j_1\rangle \to^* \langle t_2, j_2\rangle$ in G iff there is an edge $\langle t_1, i_1\rangle \to \langle t_2, i_2\rangle$ such that $i_1 \geq j_1$ and $i_2 \leq j_2$. Hence, a query $\mathtt{reachable}(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$ reduces to checking whether $\mathtt{successor}(\langle t_1, j_1\rangle, t_2) \leq j_2$.

How can we maintain the invariant Eq. (1) under edge insertions and deletions? We achieve this by storing all current successors of $\langle t_1, j_1\rangle$ to chain t_2 in a min heap $\mathtt{edgeHeap}_{t_1}^{t_2}[j_1]$. Inserting and deleting edges from $\langle t_1, j_1\rangle$ is handled by applying the same operation in $\mathtt{edgeHeap}_{t_1}^{t_2}[j_1]$. This ensures that the earliest neighbor of $\langle t_1, j_1\rangle$ in t_2 appears in the root of $\mathtt{edgeHeap}_{t_1}^{t_2}[j_1]$, which is also stored in $\mathsf{A}_{t_1}^{t_2}[j_1]$.

Example 2. *Consider again the partial order* $\leq_{\mathsf{P}}^{\sigma}$ *in Figure 1a. Some of the arrays* $\mathsf{A}_{t_1}^{t_2}$ *are shown in Figure 4. Observe that* $\mathtt{successor}(e_3, 2) = e_6$ *can be determined by the suffix minima query* $\mathtt{min}(\mathsf{A}_1^2, 0)$*. Similarly,* $\mathtt{predecessor}(e_2, 1) = e_5$ *can be determined by the suffix arg-minima query* $\mathtt{argleq}(\mathsf{A}_1^0, 1)$.

Handing $k > 2$ chains. Naturally, when $k > 2$, we can have a suffix minima array $\mathsf{A}_{t_1}^{t_2}$ for every pair of threads $t_1, t_2 \in [k]$ with $t_1 \neq t_2$. However, a path $\langle t_1, j_1\rangle \to^* \langle t_2, j_2\rangle$ might go through intermediate nodes $\langle t_3, j_1\rangle$, i.e., reachability is formed transitively across chains (e.g., the path $e_6 \to^* e_2$ in Figure 1a). We address this challenge depending on the setting (fully dynamic or purely incremental).

On a high level, in the fully dynamic setting (Section 3.3) in each $\mathsf{A}_{t_1}^{t_2}$ we store direct edges, i.e., each edge insertion is handled similarly to what was already described for $k = 2$. During queries, we perform a transitive closure operation across chains. Taking advantage of the properties of our data structure, this transitive closure takes $O(k^3)$ time, as opposed to $O(n^2)$ time, and thus remains efficient.

In the purely incremental setting (Section 4), on the other hand, $\mathsf{A}_{t_1}^{t_2}$ stores transitive reachability across chains, which is computed every time a new edge inserted. Again, taking advantage of the properties of our data structure, this transitive closure takes $O(k^2)$ time, as opposed to $O(n)$ time, and is thus very efficient. Now, queries can be performed by directly querying the respective suffix minima array, similarly to what was already described for $k = 2$.

3.2 Sparse Segment Trees

Using STs T to solve suffix minima, each operation completes in time proportional to the height of T, i.e., $O(\log n)$. In practice n can be very large and thus hinder performance. Here we develop Sparse Segment Trees (SSTs), that optimize STs for our setting. These are based on two main ideas, namely (i) *minima indexing*, and (ii) *sparse tree representation.*

Intuitively, STs solve the more general problem of *range minima queries*, where every query asks for the minima in a range $A[i : j]$, as opposed to the suffix $A[i :]$. Minima indexing exploits the fact that our queries always concern a suffix of A, and allows them to often complete without traversing a full path from the root to a leaf.

Sparse tree representation is an optimization of the tree representation, which becomes more condensed the lower the density of A is (i.e., the fewer non-∞ entries it has), stemming from the observation that ∞ entries do not contribute to the queries. In a dynamic analysis setting, G normally has low cross-chain density d, which means that the suffix-minima arrays storing reachability across chains (as described in Section 3.1) are sparse. This reduces the height of the tree, allowing both queries and updates to run faster.

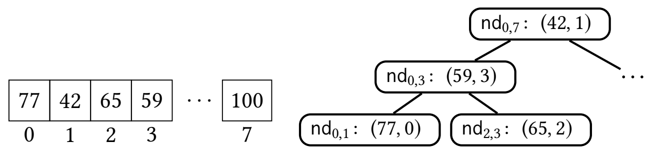
Minima indexing. Each node $\mathsf{nd}_{i,j}$ of a Sparse Segment Tree T maintains a field $\mathsf{pos} \in [i, j]$, that is defined recursively: it points to the largest index of A that contains the minimum element in the range $A[i : j]$, after excluding all indexes that are stored in the pos fields of all ancestors of $\mathsf{nd}_{i,j}$. Formally, let $\mathsf{nd}^1, \dots, \mathsf{nd}^m$ be the ancestors of $\mathsf{nd}_{i,j}$ in T. Then

$$\mathsf{nd}_{i,j}.\mathsf{pos} = \max\left(\underset{\ell \in [i,j] \setminus \{\mathsf{nd}^1.\mathsf{pos}, \dots, \mathsf{nd}^m.\mathsf{pos}\}}{\arg\min} A[\ell]\right) \qquad (2)$$

We also set $\mathsf{nd}_{i,j}.\mathtt{min} = A[\mathsf{nd}_{i,j}.\mathsf{pos}]$. Observe that now $\mathsf{nd}_{i,j}.\mathtt{min}$ might not be the minimum value in $A[i : j]$, in contrast to the standard Segment Trees.

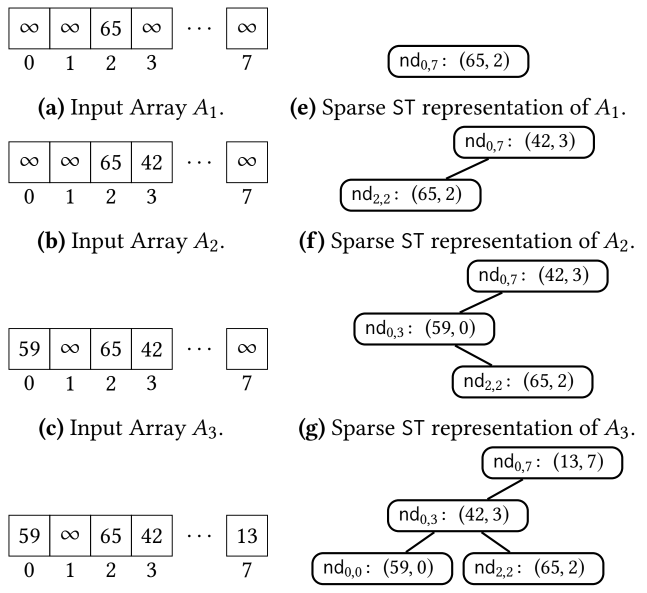
The intuition behind $\mathsf{nd}_{i,j}.\mathsf{pos}$ is as follows. Each query $\mathtt{min}(A, i)$ starts a traversal from $T.\mathtt{root}$ to a leaf. If we encounter a node $\mathsf{nd}_{i,j}$ with $\mathsf{nd}_{i,j}.\mathsf{pos} \geq i$, the traversal can stop early and return that $\mathtt{min}(A, i) = \mathsf{nd}_{i,j}.\mathtt{min}$. Since we always query on a suffix of A, storing the *largest* index pointing to a minima increases the chances of stopping early. Moreover, we exclude the pos fields of the ancestors of $\mathsf{nd}_{i,j}$ as these are irrelevant for $\mathsf{nd}_{i,j}$, in the sense that this index would have already stopped the traversal in an ancestor of $\mathsf{nd}_{i,j}$.

Example 3. *Consider the array A in Figure 5a, and part of the corresponding* ST *in Figure 5b. We have* $\mathsf{nd}_{0,7}.\mathsf{pos} = 1 = \arg\min_i A[i]$*. A query* $\mathtt{min}(A, 0)$ *is now directly answered at the root, as* $\mathsf{nd}_{0,7}.\mathsf{pos} \in [0 : 7]$*. On the other hand, the query* $\mathtt{min}(A, 2)$ *is not answered at the root as* $\mathsf{nd}_{0,7}.\mathsf{pos} \notin [2 : 7]$*. The procedure then traverses its child* $\mathsf{nd}_{0,3}$*. We have* $\mathsf{nd}_{0,3}.\mathsf{pos} = 3$,



(a) Array A. **(b)** ST representation of $A_{t_1}^{t_2}$.

Figure 5. ST representation with minima indexing. The entries $A[4{:}\ 6]$ are > 59. The notation $\mathsf{nd}_{i,j}\colon (x, y)$ denotes $\mathsf{nd}_{i,j}.\mathtt{min} = x$, $\mathsf{nd}_{i,j}.\mathsf{pos} = y$.



(a) Input Array A_1. **(e)** Sparse ST representation of A_1.

(b) Input Array A_2. **(f)** Sparse ST representation of A_2.

(c) Input Array A_3. **(g)** Sparse ST representation of A_3.

(d) Input Array A_4. **(h)** Sparse ST representation of A_4.

Figure 6. Sparse ST representation. The notation $\mathsf{nd}_{i,j}\colon (x, y)$ denotes $\mathsf{nd}_{i,j}.\mathtt{min} = x$, $\mathsf{nd}_{i,j}.\mathsf{pos} = y$.

which indexes the smallest element in $A[0 : 3]$ *after excluding* $\mathsf{nd}_{0,7}.\mathsf{pos}$. *The traversal stops here, as* $\mathsf{nd}_{0,3}.\mathsf{pos} \in [2 : 7]$.

Sparse representation. Sparse Segment Trees are tailored towards representing sparse arrays. This allows the height of the tree to shrink, which consequently results in more efficient updates and queries. A crucial aspect of SSTs is that they are self-adapting and do not require any a priori information about the sparsity of the input.

Intuitively, we achieve this as follows. Empty array indexes are never explicitly represented in the intermediate nodes of the tree. Every intermediate node nd is such that nd.pos points to a unique, non-empty entry of A. The opposite is also true: every non-empty entry of A is indexed by the pos field of a node in the earliest possible level satisfying Eq. (2).

Example 4. *Consider an array A of length* 8, *initially empty. Figure 6a–6d sequentially update A with new entries, and Figure 6e–6h show the evolution of the corresponding* SST T. *In Figure 6e, the unique node of T represents that A has only one non-empty entry, at index* 2 *with value* 65. *In contrast, a standard (non-sparse) Segment Tree would contain* 15 *nodes and have height* 3. *Figure 6f shows the update of T after setting* $A[3] = 42$. *Since* 42 *is now the smallest value of A, it is stored in the root, while a new node* $\mathsf{nd}_{2,2}$ *becomes the left child of the root, with its* pos *and* min *fields set according to Eq. (2). Again, in the standard* ST *representation,* $\mathsf{nd}_{2,2}$ *would have nodes* $\mathsf{nd}_{0,3}$ *and* $\mathsf{nd}_{2,3}$ *as ancestors, but these nodes are never created in our* SST. *Figure 6g, shows another update of T after setting* $A[0] = 59$. *As* $59 \le \mathsf{nd}_{0,7}.\mathtt{min}$, *the root remains intact. The index* 0 *belongs to the left subtree of the root and* $59 < \mathsf{nd}_{2,2}.\mathtt{min}$. *This causes the new entry to be inserted to a fresh node* $\mathsf{nd}_{0,3}$ *and assigned as the left child of the root, taking the existing node* $\mathsf{nd}_{2,2}$ *as its right child. Figure 6h shows the final update of T after setting* $A[7] = 13$. *Since* 13 *is now the smallest value of A, it is stored in the root, while the previous value stored in the root is recursively pushed downwards.*

One last optimization step is to flatten subtrees of small size at the leaves to array blocks. For a node $\mathsf{nd}_{i,j}$, if $\mathtt{depth}(\mathsf{nd}_{i,j})$ is larger than a value deeming that the node represents a small sub-array of A, then it becomes a leaf in T, storing the subarray $A[i : j]$. We call such a node $\mathsf{nd}_{i,j}$ a *block node*. This representation is valuable when the array contains segments which are densely populated but also far apart from each other.

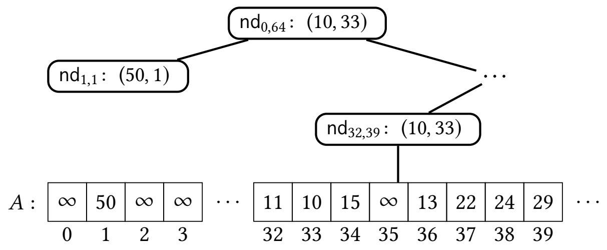


Figure 7. Flatting small subtrees to block nodes.

Example 5. *Consider the array A and its corresponding* SST *in Figure 7. A contains only a single entry in the range* $0, \ldots, 31$. *This entry is sparsely represented in the left subtree of the root. On the other hand, A is dense in the range* $32, \ldots, 39$, *which would require a full subtree. Instead, when the range of the represented subarray is smaller than a threshold b, the corresponding subtree is flattened to an array block.*

Operations on Sparse Segment Trees. Algorithm 1 implements the operations update, min, argleq on a SST.

Function update(nd, *pos*, v). This function inserts a node nd′ into the subtree of nd, with nd′.pos = *pos* and nd′.min = v. It first checks if nd is initialized; if not, it creates a new node (Line 2). If nd is the root, the function checks and deletes if a node nd″ exists in the subtree of nd where nd″.pos = *pos* (Line 3). Then, it checks if $2^{\mathtt{depth(nd)}} \ge n/b$, where b is the block-size threshold. If so, nd becomes a block node and the update is performed directly on nd.block (Line 4). Otherwise, it checks if $v \le \mathsf{nd}.\mathtt{min}$. If so, it swaps the values of nd.pos and nd.min with *pos* and v (Line 7). This captures that

Algorithm 1: The Sparse Segment Tree.

```
1  function update(nd, $pos$, $v$)
2      if nd = nil then nd ← createNode($pos$, $v$)
3      else if nd = $T$.root then nd ← deleteNode(nd, $pos$)
4      else if $2^{\text{depth(nd)}} \geq n/b$ then nd.block[$pos$] = $v$;
5      else
6          if $v$ < nd.min ∨ ($v$ = nd.min ∧ $pos$ > nd.pos) then
7              nd.min, nd.pos, $v$, $pos$ ← $v$, $pos$, nd.min, nd.pos
8          Let $mid$ ← nd.start + ⌊(nd.end − nd.start)/2⌋
9          if $pos \leq mid$ then
10             nd.left ← updateHelper(nd.left, $pos$, $v$)
11         else
12             nd.right ← updateHelper(nd.right, $pos$, $v$)
13 function updateHelper(nd, $pos$, $v$)
14     if nd = nil then nd ← createNode($pos$, $v$)
15     else if $pos$ ∈ [nd.start, nd.end] then update(nd, $pos$, $v$)
16     else nd ← createLowestCommonAncestor(nd, $pos$, $v$)
17     return nd

18 function min(nd, $i$)
19     if nd = nil ∨ $i$ > nd.end then return ∞
20     if nd.block ≠ nil then return min(nd.block, $i$);
21     if nd.pos ≥ $i$ then return nd.min;
22     else
23         $l$ ← min(nd.left, $i$)
24         $r$ ← min(nd.right, $i$)
25         if $l < r$ then return $l$ else return $r$

26 function argleq(nd, $v$)
27     if nd = nil ∨ nd.min > $v$ then return −∞
28     if nd.block ≠ nil then return argleq(nd.block, $v$);
29     if nd.pos ≥ nd.left.end ∧ nd.pos ≥ nd.right.end then
           return nd.pos
30     else if nd.right.min ≤ $v$ then return
           $max$(nd.pos, argleq(nd.right, $v$))
31     else return $max$(nd.pos, argleq(nd.left, $v$))
```

the new entry will be stored in nd and the previous entry in nd will be relocated to a child node. Next, it identifies which subtree the corresponding entry belongs to, and makes a call to `updateHelper` (Line 10, Line 12). If the subtree is not present, `updateHelper` creates a new node (Line 14). Otherwise, it checks if pos is within the range of the subtree and if so it recursively calls `update`. If the subtree is present but pos is not within the range, `createLowestCommonAncestor` creates a new node whose range corresponds to the lowest common ancestor of the existing subtree and pos.

Function `min(nd, i)`. This recursive function returns ∞ if nd is `nil` or $i >$ nd.end (Line 19), which captures the end of the recursion. Otherwise, it returns the smallest `min` value in the subtree of nd whose corresponding pos is greater than or equal to i, as follows. First, it checks if nd is a block node, in which case it returns the minimum of the block array nd.block (Line 20). Otherwise, if the minima indexing optimization succeeds, the traversal stops here, returning nd.min (Line 21). If both conditions fail, the function proceeds recursively on the left and right child of nd (Lines 22–25).

Function `argleq(nd, v)`. This function returns $-\infty$ if nd is `nil` or nd.min $> v$. Otherwise, it returns the largest pos in the subtree of nd whose corresponding `min` value is smaller than or equal to v, as follows. It first checks if nd is a block node, in which case it simply performs an array traversal on nd.block (Line 28). Otherwise, the function uses the minima indexing information in nd.pos to check if it can return early (Line 29). If the check fails, it makes a recursive call on the right subtree if nd.right. min $\leq v$ (Line 30). If all conditions fail, it makes a recursive call on the left subtree (Line 31).

Using SSTs. SSTs handle operations on the represented suffix-minima array A is as follows. Queries `min(A, i)` and `argleq(A, a)` are handled as `min(T.root, i)` and `argleq(T.root, a)`, respectively. On the other hand, `update(A, i, a)` is handled as `update(T.root, i, a)`.

Complexity and the height of SSTs. The theoretical advantage of SSTs over STs is the fact that their height is bounded not only by $\log n$, but also by the density of the represented array. This is captured in the following lemma.

Lemma 1. *Consider an array A of size n with d non-empty (i.e., non-∞) entries, and let T be its corresponding* SST. *Then the height of T is bounded by* $\min(\log n, d)$.

Thus, we can use Lemma 1 to bound the time taken to perform any update and query on the represented array A. In practice, however, the running time can be even smaller, as, in contrast to standard STs, the update and query operations can stop before traversing a full path from the root to a leaf.

3.3 Collective Sparse Segment Trees

We are now ready to describe CSSTs, leading to Theorem 1.

CSSTs. For every two distinct chains $t_1, t_2 \in [k]$, CSSTs maintain a suffix-minima array $\mathsf{A}_{t_1}^{t_2}$ storing direct edges from t_1 to t_2, together with a min-heap $\mathsf{edgeHeap}_{t_1}^{t_2}[j_1]$, for each $j_1 \in [n]$, as described in Section 3.1. Suffix-minima queries on $\mathsf{A}_{t_1}^{t_2}$ are handled using a SST, as described in Section 3.2. The pseudocode is shown in Algorithm 2. The functions `insertEdge` and `deleteEdge` follow directly the description in Section 3.1, i.e., they manipulate the arrays $\mathsf{A}_{t_1}^{t_2}$ and min-heaps $\mathsf{edgeHeap}_{t_1}^{t_2}[j_1]$.

The function `reachable` is implemented via a `successor` query, which is more involved. As we have $k > 2$ chains, `successor` runs a forward traversal from $\langle t_1, j_1 \rangle$ to discover transitive reachability across chains. This is done by repeated

Algorithm 2: Fully dynamic CSSTs.

```
1  function successor($(\langle t_1, j_1\rangle, t_2)$)
2    foreach $t'_1 \in [k] \setminus \{t_1\}$ do
3      closure[$t'_1$] $\leftarrow \min(\mathsf{A}^{t'_1}_{t_1}, j_1)$
4    do
5      changed $\leftarrow$ false
6      foreach $t'_1 \in [k] \setminus \{t_1\}$ do
7        foreach $t'_2 \in [k] \setminus \{t_1\}$ do
8          if $t'_1 \neq t'_2$ then
9            Let $v \leftarrow \min(\mathsf{A}^{t'_1}_{t'_2}$, closure[$t'_2$])
10           if $v <$ closure[$t'_1$] then
11             closure[$t'_1$] $\leftarrow v$
12             changed $\leftarrow$ true
13   while changed
14   return closure[$t_2$]

15 function predecessor($(\langle t_1, j_1\rangle, t_2)$)
     // Similar to successor

16 function reachable($(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$)
17   if successor($(\langle t_1, j_1\rangle, t_2) \leq j_2$) then return True
     else return False

18 function insertEdge($(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$)
19   if $j_2 <$ edgeHeap$^{t_2}_{t_1}[j_1]$.min then update($\mathsf{A}^{t_2}_{t_1}, j_1, j_2$)
20   edgeHeap$^{t_2}_{t_1}[j_1]$.insert($j_2$)

21 function deleteEdge($(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$)
22   if $j_2 =$ edgeHeap$^{t_2}_{t_1}[j_1]$.min then
23     edgeHeap$^{t_2}_{t_1}[j_1]$.delete($j_2$)
24     update($\mathsf{A}^{t_2}_{t_1}, j_1$, edgeHeap$^{t_2}_{t_1}[j_1]$.min)
25   else edgeHeap$^{t_2}_{t_1}[j_1]$.delete($j_2$)
```

$$\begin{tikzcd}
\text{Thread 0} & \text{Thread 1} & \text{Thread 2} & \text{Thread 3} \\
\langle 0,0\rangle \arrow[d] & \langle 1,0\rangle \arrow[d] & \langle 2,0\rangle \arrow[d] & \langle 3,0\rangle \arrow[d] \\
\langle 0,1\rangle \arrow[d] \arrow[ur, "\textcircled{1}"] & \langle 1,1\rangle \arrow[d] \arrow[r, "\textcircled{3}"] & \langle 2,1\rangle \arrow[d] & \langle 3,1\rangle \arrow[d] \\
\langle 0,2\rangle \arrow[rrr, bend right, "\textcircled{2}"] & \langle 1,2\rangle & \langle 2,2\rangle \arrow[ur, "\textcircled{4}"] & \langle 3,2\rangle
\end{tikzcd}$$

Figure 8. Query successor($\langle 0, 0\rangle, 3$).

min queries in the suffix minima arrays $\mathsf{A}^{t'_1}_{t'_2}$ (Line 9), which find the earliest node of t'_1 that has an edge from the currently earliest node of t'_2 found to be reachable from $\langle t_1, j_1\rangle$. Crucially, this computation completes in $O(k^3)$ steps, as opposed to $O(n^2)$ steps normally required for a graph traversal.

Example 6. *Consider the chain DAG in Figure 8 and the operation* successor($\langle 0, 0\rangle, 3$)*. The first step is identifying the earliest successors of $\langle 0, 0\rangle$ reachable via direct edges from $\langle 0, 0\rangle$ and its thread-local successors, in all other threads. This discovers $\langle 1, 0\rangle$ via $\textcircled{1}$ and $\langle 3, 2\rangle$ via $\textcircled{2}$. While $\langle 3, 2\rangle$ is indeed in chain 3, it is not necessarily the* earliest *successor of $\langle 0, 0\rangle$ (indeed, the earliest successor is $\langle 3, 1\rangle$). Hence the function proceeds iteratively for each newly discovered node until reaching a fixed point. This leads to the discovery of $\langle 2, 1\rangle$ via $\textcircled{3}$ and eventually the discovery of $\langle 3, 1\rangle$ via $\textcircled{4}$.*

Analysis. We now state the properties of fully dynamic CSSTs. First, since the arrays $\mathsf{A}^{t_2}_{t_1}$ only store direct edges, the following sparsity property is straightforward.

Lemma 2 (Sparsity). *Let d be the cross-chain density of G. Then the density of each array $\mathsf{A}^{t_2}_{t_1}$ is bounded by d.*

The use of min-heaps guarantees the following invariant.

Lemma 3 (Invariant). *For every distinct $t_1, t_2 \in [k]$, we have* $\mathsf{A}^{t_2}_{t_1}[j_1] = \min(\{j_2 \in [n].\ \langle t_1, j_1\rangle \to \langle t_2, j_2\rangle\})$.

A *crossing path* is a sequence of nodes $\pi\colon \langle t_0, j_0\rangle, \dots, \langle t_{m-1}, j_{m-1}\rangle$ such that for each $\ell \in [m-2]$ we have $t_\ell \neq t_{\ell+1}$ and there are $i_1, i_2 \in [n]$ such that (i) $i_1 \geq j_\ell$, (ii) $i_2 \leq j_{\ell+1}$, and (iii) $\langle t_\ell, i_1\rangle \to \langle t_{\ell+1}, i_2\rangle$. Clearly, for any two nodes $\langle t_1, j_1\rangle$ and $\langle t_2, j_2\rangle$, if $\langle t_1, j_1\rangle \to^* \langle t_2, j_2\rangle$, then there exists a crossing path between them of length at most k. The following lemma implies the correctness of queries.

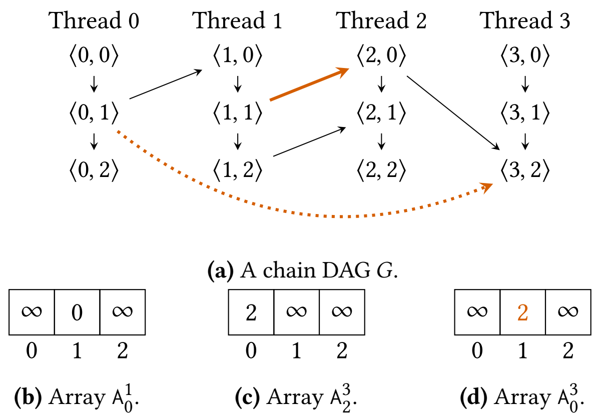
Lemma 4. *For every $i \in [k]$, after the i-th iteration of the do-while loop in Line 4,* closure[t'_1] *points to the earliest node in chain t'_1 that is reachable from $\langle t_1, j_1\rangle$ via a crossing path of length $\leq i + 1$.*

Regarding the time complexity, due to Lemma 2 and Lemma 1, each operation in a suffix minima array $\mathsf{A}^{t_2}_{t_1}$ takes $O(\min(\log n, d))$ time. Since the maximum out-degree is δ, each operation on a min heap edgeHeap$^{t_2}_{t_1}[j_1]$ runs in $O(\log \delta)$ time. Thus, updates take $O(\max(\log \delta, \min(\log n, d)))$ time, while, due to Lemma 4, queries take $O(k^3 \min(\log n, d))$ time, arriving at Theorem 1.

Space usage. We now discuss the space usage of CSSTs. Consider a chain DAG G of n nodes, k chains, and cross-chain density d. In this fully dynamic setting, CSSTs store all edges in the min heaps edgeHeap$^{t_2}_{t_1}[j_1]$, as edge deletions requires to remember all edges added so far. The suffix minima arrays $\mathsf{A}^{t_2}_{t_1}$ store in total $O(dk)$ entries, as (i) each chain has at most d entries that have at least one successor to another chain, and (ii) there are $< k$ other chains. Naturally, $d \leq n$, thus in the worst case the above expression becomes $O(nk)$. This is the same space complexity as Vector Clocks. However, Vector Clocks do not exploit the sparsity of G. As we will see in Section 5, in practice typically $d \ll n$, which makes CSSTs more space-efficient than Vector Clocks.

4 Incremental CSSTs

In practice, many dynamic analyses maintain their partial order incrementally, i.e., they only insert and never delete orderings. To optimize for this incremental setting, in this



(a) A chain DAG G.

(b) Array A_0^1. (c) Array A_2^3. (d) Array A_0^3.

Figure 9. Inserting an edge $\langle 1,1\rangle \longrightarrow \langle 2,0\rangle$ and inferring the transitive path $\langle 0,1\rangle \longrightarrow^* \langle 3,2\rangle$.

section we specialize `CSSTs` to only handle incremental updates, with guarantees stated in the following theorem.

Theorem 2. *Consider a chain DAG G of n nodes, k chains, and cross-chain density d. Incremental* `CSSTs` *maintain G under incremental updates, costing $O(k^2 \min(\log n, d))$ time per update and $O(\min(\log n, d))$ time per query.*

Compared to Theorem 1, Theorem 2 trades the cost dependency on the number of chains from queries to updates, while also decreasing it by a factor k. Moreover, now the cost of both queries and updates is bounded by the cross-chain density of G (as opposed to only queries in Theorem 1).

Incremental `CSSTs`. Incremental `CSSTs` maintain $k(k-1)$ suffix minima arrays $\mathsf{A}_{t_1}^{t_2}$, with $t_1 \neq t_2$, each implemented using an SST $T_{t_1}^{t_2}$. These arrays now store *transitive reachability*, as opposed to direct edges between chains. Algorithm 3 shows how each array $\mathsf{A}_{t_1}^{t_2}$ is queried and updated, following the reachability queries and edge insertions in `CSSTs`. The functions `successor`$(\langle t_1, j_1\rangle, t_2)$, `predecessor`$(\langle t_1, j_1\rangle\rangle, t_2)$ and `reachable`$(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$ handle successor, predecessor, and reachability queries in a straightforward manner: since each $\mathsf{A}_{t_1}^{t_2}$ is transitively closed, these operations reduce to suffix-minima queries on $\mathsf{A}_{t_1}^{t_2}$.

The function `insertEdge`$(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$ is somewhat more complex, to guarantee that the arrays $\mathsf{A}_{t_1}^{t_2}$ indeed store transitive reachability across chains. To achieve this, besides inserting the direct edge $\langle t_1, j_1\rangle \longrightarrow \langle t_2, j_2\rangle$, this function performs a transitive closure computation by finding the predecessor $\langle t_1', j_1'\rangle$ of $\langle t_1, j_1\rangle$ (Lines 10–11) and the successor $\langle t_2', j_2'\rangle$ of $\langle t_2, j_2\rangle$ (Lines 12–13) in all pairs of chains t_1', t_2'. Then, an edge $\langle t_1', j_1'\rangle \longrightarrow \langle t_2', j_2'\rangle$ is inserted unless a path between these nodes already exists.

Example 7. *Consider the chain DAG in Figure 9a and the operation* `insertEdge`$(\langle 1,1\rangle, \langle 2,0\rangle)$. *We obtain the transitive path $\langle 0,1\rangle \longrightarrow^* \langle 3,2\rangle$ as follows. First, the predecessor of $\langle 1,1\rangle$ in thread 0 is identified via the query* `predecessor`$(\langle 1,1\rangle, 0)$. *This query performs an* `argleq` *operation on A_0^1 (Figure 9b) with value 1, returning node $\langle 0,1\rangle$. Then, the successor of $\langle 2,0\rangle$ in thread 3 is identified via the query* `successor`$(\langle 2,0\rangle, 3)$. *This query performs a* `min` *operation on A_2^3 (Figure 9c) with index 0, and returns the node $\langle 3,2\rangle$. Next, the update operation on A_0^3 adds the transitive edge, resulting in A_0^3 in Figure 9d.*

Algorithm 3: Incremental `CSSTs`.

```
1  function successor($(\langle t_1, j_1\rangle, t_2)$)
2      return min($(\mathsf{A}_{t_1}^{t_2}, j_1)$)

3  function predecessor($(\langle t_1, j_1\rangle, t_2)$)
4      return argleq($(\mathsf{A}_{t_2}^{t_1}, j_1)$)

5  function reachable($(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$)
6      if successor($(\langle t_1, j_1\rangle, t_2) \leq j_2$) then return True
         else return False

7  function insertEdge($(\langle t_1, j_1\rangle, \langle t_2, j_2\rangle)$)
8      foreach $t_1' \in [k]$ do
9          foreach $t_2' \in [k] \setminus \{t_1'\}$ do
10             if $t_1' = t_1$ then Let $j_1' \leftarrow \langle t_1, j_1\rangle$
11             else Let $j_1' \leftarrow$ predecessor($(\langle t_1, j_1\rangle, t_1')$)
12             if $t_2' = t_2$ then Let $j_2' \leftarrow \langle t_2, j_2\rangle$
13             else Let $j_2' \leftarrow$ successor($(\langle t_2, j_2\rangle, t_2')$)
14             if successor($(\langle t_1', j_1'\rangle, t_2') > j_2'$) then
                   update($(\mathsf{A}_{t_1'}^{t_2'}, j_1', j_2')$)
```

Analysis. The correctness of incremental `CSSTs` is based on the following invariants. The first concerns soundness, i.e., every entry in each array $\mathsf{A}_{t_1}^{t_2}$ corresponds to a path between the respective nodes in chains t_1 and t_2. The second concerns completeness, i.e., if there is a path $\langle t_1, j_1\rangle \longrightarrow^* \langle t_2, j_2\rangle$ between two nodes, it is captured as a suffix minima in the array $\mathsf{A}_{t_1}^{t_2}$. In conjunction, they imply the correctness of `successor`, `predecessor`, and `reachable` queries implemented as `min` and `argleq` operations on the corresponding array $\mathsf{A}_{t_1}^{t_2}$.

Lemma 5 (Soundness). $\mathsf{A}_{t_1}^{t_2}[j_1] = j_2 \implies \langle t_1, j_1\rangle \longrightarrow^* \langle t_2, j_2\rangle$.

Lemma 6 (Completeness). *$\langle t_1, j_1\rangle \longrightarrow^* \langle t_2, j_2\rangle$ and $t_1 \neq t_2$ $\implies \exists j_1' \geq j_1.\ \mathsf{A}_{t_1}^{t_2}[j_1'] \leq j_2$.*

Recall that the key feature of SSTs is their ability to handle sparse arrays efficiently. As incremental `CSSTs` insert transitive edges in the arrays $\mathsf{A}_{t_1}^{t_2}$, this might increase their density, adversely affecting the underlying SST, as we saw in Section 3.2. At closer inspection, however, the density of each $\mathsf{A}_{t_1}^{t_2}$ cannot grow beyond the cross-chain density of G.

Lemma 7 (Sparsity). *Let d be the cross-chain density of G. Then the density of each array $\mathsf{A}_{t_1}^{t_2}$ is bounded by d.*

Combined with our bounds on the height of SSTs (Lemma 1 in Section 3.2), Lemma 7 implies that the height of the SST representing $\mathsf{A}_{t_1}^{t_2}$ is bounded both by $\log n$ and by the cross-chain density of G. We thus arrive at Theorem 2.

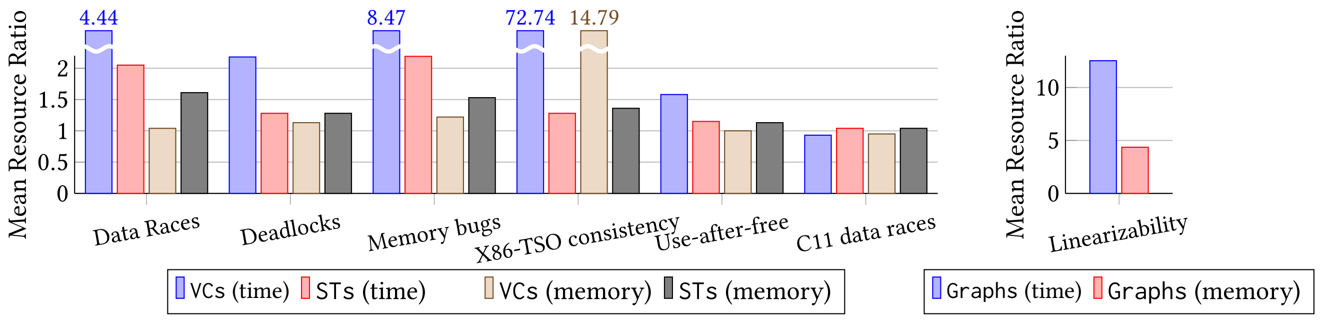


Figure 10. The geometric mean of the ratio of time and memory used in each benchmark by VCs and STs over incremental CSSTs (left), and Graphs over fully dynamic CSSTs (right), in the dataset of the corresponding analysis.

Space usage. Due to Lemma 7, the total space used by the suffix minima arrays $A_{t_1}^{t_2}$ is $O(dk)$, by an analysis similar to that of fully dynamic CSSTs in Section 3.3. This time, CSSTs does not use heaps to store all edges, thus the total space used is $O(dk)$. Vector Clocks and STs [31], on the other hand, use $O(nk)$ space, which is larger in practice.

5 Experimental Evaluation

Here we incorporate CSSTs in a number of dynamic analyses from recent literature and evaluate their performance.

5.1 Experimental Setup

Data structures. Each analysis dynamically maintains a partial order P using various representations. We explore different standard ways for this purpose, as follows.

- Graphs is a standard graph representation of P, that is not transitively closed.
- VCs is a standard Vector Clock representation of P, that is (by design) transitively closed [28].
- STs is the data structure in [31] based on Segment Trees.
- CSSTs, as introduced in this paper.

Most analyses make only incremental updates, in which case Vector Clocks are preferred over plain graphs. Thus in this setting we compare CSSTs to VCs and ST. On the other hand, VCs and ST cannot handle decremental updates, hence in the fully dynamic setting we compare CSSTs to Graphs. The data structure used in the respective paper is marked by †. As some tools are not public, we implemented their analysis following the corresponding paper.

Optimizations. To set the threshold b for the size of the block nodes in CSSTs (Section 3.2), we perform a randomized stress test with varying sizes of b, and measure their performance. Based on this test, we set $b = 32$.

We also implement two optimizations in VCs. First, each new edge $e_1 \longrightarrow e_2$ must be propagated to the successors of e_2 in its own chain. We stop this propagation as soon as we find some e_2' with already $e_1 \longrightarrow^* e_2'$. Second, for each chain, we do not create VCs for the suffix of its events that do not have any direct orderings from other chains, as ordering queries on such nodes can be inferred from their predecessors.

Setup. In each analysis, we use the dataset of the corresponding paper and relevant literature, and report on benchmarks on which some data structure runs in $\geq$ 1s. We use an Ubuntu 22.04 machine with 2.4GHz CPU and 64GB of memory.

5.2 Experimental Results

We begin with a macroscopic view in Figure 10, showing the (geometric) average improvement in time and memory that CSSTs offer over VCs, STs and Graphs in each analysis. We see that in nearly all cases, these ratios are above 1, meaning that CSSTs indeed outperform the other data structures. The only exception concerns data races in C11; we will analyze this case in more detail later. The improvement is even more pronounced on the last analysis (linearizability), which is fully dynamic and thus the only baseline is plain Graphs.

Although the worst-case memory usage of CSSTs is $O(nk)$, and thus the same as VCs and STs, in practice it is less. This confirms our insights that typical analyses give rise to sparse chain DAGs, which is then exploited by CSSTs. We will take a closer look into the sparsity of each benchmark later.

We remark that the resource (time, memory) reported in each benchmark concerns the whole analysis, not just the corresponding data structure[1]. The actual resource improvement achieved by CSSTs is significantly larger.

1. Data race prediction. We start with the M2 data race detector [31]. This analysis observes traces σ which might be data-race free, and attempts to permute them to new traces σ' that are valid (called correct reorderings) and expose a data race. Internally, M2 utilizes STs for maintaining its partial order P. The reachability queries are incremental. We measure the total time taken in the analysis.

Table 1 shows the performance of M2 with each data structure. VCs become unscalable early, and are outperformed by STs, which were indeed developed to address this scalability issue. We observe that CSSTs are the most performant data structure on each individual benchmark. Their advantage over STs is primarily due to the efficient way that CSSTs handle sparsity (Section 3.2). Indeed, column q reports the mean density among each suffix minima array inside CSSTs when

[1]The analysis on data races for C11 is an exception, as we explain later.

it obtained its densest form, and matches our expectations that is typically small. Our sparse treatment not only reduces the runtime of CSSTs but also their memory footprint, and thus support the analysis even when VCs and STs go out of memory (on xalan and batik).

As a side note, observe that the running time of each method is not always increasing when we move to larger traces (eg, all data structures take more time to process moldyn, with size $N = 200.3\text{K}$ than jigsaw, with size 3.1M). This is because moldyn has more potential data races that the analysis needs to check, despite its smaller size. Similar, non-monotonic, patterns appear in other analyses as well.

Table 1. Race prediction results. The timeout is 5h.

benchmark	T	N	q	VCs (s)	STs† (s)	CSSTs (s)
clean	12	1.3K	0.28	2.4	2.6	1.6
bubblesort	29	4.7K	0.15	69.4	56.2	27.5
lang	10	6.3K	0.22	93.3	39.2	27.3
readerswriters	8	11.3K	0.32	54.4	23.5	18.5
raytracer	6	15.8K	0.17	10.5	10.2	9.3
bufwriter	9	22.3K	0.2	130	24.6	12.9
ftpserver	14	49.6K	0.06	41.2	14.1	9.0
moldyn	6	200.3K	0.12	T.O.	T.O.	12636
linkedlist	15	1.0M	0.06	T.O.	T.O.	8893
derby	7	1.4M	0.04	1436	215	197
jigsaw	15	3.1M	0.06	154	45.6	32.0
sunflow	17	11.7M	0.01	T.O.	780	505
xalan	9	122.5M	0.01	T.O.	O.O.M.	979
batik	8	157.9M	0.01	O.O.M.	O.O.M.	1956
Total	-	**297.9M**	-	**>91992**	**>73211**	**25304**

2. Deadlock prediction. Here we incorporate CSSTs in SeqCheck, a dynamic deadlock prediction analysis [8]. This analysis identifies potential deadlocks by analyzing lock-acquisition orders between threads, and then try to witness each deadlock by a valid reordering of the observed trace. For this purpose, it incrementaly maintains a partial order which will eventually be linearized to this valid reordering.

Our results are shown in Table 2. VCs are again the slowest performer by far, though they are very competitive on a few benchmarks for which deadlocks can be detected fairly early in the analysis. Naturally, STs and CSSTs perform similarly on these benchmarks as well. On the more demanding benchmarks, however, CSSTs is clearly the best performer. We also observe that the average density q is small.

Table 2. Deadlock results.

benchmark	T	N	q	VCs (s)	STs† (s)	CSSTs (s)
jigsaw	21	143.0K	0.03	356	14.8	7.8
elevator	5	245.9K	0.02	3.7	2.8	2.6
hedc	7	409.8K	0.04	23.2	22.0	23.2
JDBCMySQL	3	442.9K	0.01	3.7	4.1	3.5
cache4j	2	775.5K	0.01	5.5	6.6	5.6
Swing	8	3.8M	0.01	14.8	13.4	13.1
sunflow	15	21.5M	0.01	369	111	110
eclipse	15	64.2M	0.01	535	711	282
Total	-	**91.4M**	-	**1311**	**886**	**448**

3. Memory bug prediction. Here we incorporate CSSTs in ConVulPOE, a dynamic analysis targeting memory bugs due to concurrency [40], by reordering observed traces. The reachability queries are incremental. We measured the total time taken in the analysis.

Our results are shown in Table 3. We again observe that VCs are the slowest data structure to support this analysis. While STs offer a generous speedup, CSSTs are again the most performant, and even manage to handle benchmarks that makes VCs to time out and STs to go out of memory.

Table 3. Memory bug prediction results. The timeout is 5h.

benchmark	T	N	q	VCs (s)	STs† (s)	CSSTs (s)
pbzip2	7	243.5K	0.06	298	6.7	3.8
pigz	6	2.3M	0.01	T.O.	2087	1265
xz	2	3.0M	0.01	15.1	13.9	12.9
lbzip2	11	9.3M	0.04	273	96.6	46.9
x264	7	10.4M	0.03	474	165	62.0
libvpx	2	19.0M	0.01	440	83.6	77.6
libwebp	2	29.5M	0.1	324	118	115
x265	15	37.1M	0.02	T.O.	O.O.M.	627
Total	-	**110.9M**	-	**>37824**	**>20570**	**2211**

4. Consistency checking in x86-TSO. Here we report our results for consistency checking under the x86-TSO memory model. Though the problem is NP-complete, polynomial-time heuristics are known to be remarkably accurate. We experiment with the consistency analysis developed in [34]. In contrast to the previous analyses where the underlying chain DAG had one chain per thread, here we have two chains per thread, one for its program order and one for its store buffer. The reachability queries are incremental. We measure the total time taken in the analysis.

The results are shown in Table 4. VCs are very slow compared to the other two data structures. Indeed, this analysis is quite demanding, as it performs repeated updates between events that are in the middle of the partial order, which leads to deep edge propagations. Although STs perform considerably better than VCs, CSSTs are decisively faster in this setting too. Finally, the average density q is larger here than in previous analyses, but still considerably below 1.

5. Use-after-free prediction. Here we incorporate CSSTs in the UFO analysis for use-after-free vulnerabilities due to concurrency [19]. This analysis is based on SMT, but relies on efficient partial-order reasoning to generate SMT queries. The reachability queries are incremental. We measure the time to generate the queries. Our results are shown in Table 5.

In alignment with our observations so far, CSSTs outperform VCs and STs on all benchmarks. However the speedup is not as large as in other analyses. Looking closer into the running times, we observe that the analysis itself spends considerable time in components other than maintaining reachability, which are common for all data structures. If we only focus on the time taken for maintaining the partial

Table 4. Consistency checking results. The timeout is 1h.

benchmark	T	N	q	VCs (s)	STs (s)	CSSTs (s)
dekker	3	16.3K	0.41	6.9	0.3	0.25
peterson	3	19.0K	0.37	9.4	0.4	0.45
lamport	3	28.1K	0.39	18.3	0.6	0.6
dq	4	31.5K	0.24	11.5	0.5	0.44
chase-lev	5	32.2K	0.24	23.3	0.6	0.51
szymanski	3	77.9K	0.33	17.4	0.9	0.8
buf-ring	9	115.3K	0.39	322	7.3	6.27
mcs-lock	11	196.4K	0.17	298	28.3	20.71
spsc	3	243.6K	0.53	2548	2.1	1.74
linuxrwlocks	6	276.4K	0.28	T.O.	12.6	11.18
fib-bench	3	300.0K	0.4	T.O.	4.7	4.21
seqlock	17	318.1K	0.15	1227	46.1	28.16
spinlock	11	482.5K	0.39	T.O.	103	75.11
ttaslock	11	491.1K	0.41	T.O.	111	81.19
exp-bug	4	498.5K	0.38	2591	3.6	2.76
mutex	11	519.7K	0.56	T.O.	122	86.09
ticketlock	6	569.6K	0.4	T.O.	20.8	17.17
gcd	3	750.1K	0.28	T.O.	6.0	4.52
indexer	17	800.0K	0.51	7.8	5.8	3.23
twalock	11	900.0K	0.43	T.O.	174	118.39
treiber	6	1.0M	0.22	T.O.	20.7	16.73
mpmc	10	2.0M	0.17	T.O.	41.6	22.59
barrier	5	2.8M	0.6	T.O.	139	112.61
Total	-	**12.5M**	-	**>46680**	**852**	**616**

order, CSSTs indeed offer a significant speedup, e.g., 3.5× and 1.7× in BoundedBuffer over VCs and STs, respectively.

Table 5. Use-after-free results. The timeout is 5h.

benchmark	T	N	q	VCs† (s)	STs (s)	CSSTs (s)
bbuf	3	27.7K	0.02	426	459	415
BoundedBuffer	11	325.7K	0.02	835	736	679
DiningPhil	21	1.4M	0.01	1559	1191	1110
fanger01-ok	5	93.4K	0.06	366	194	190
qtsort	6	41.7M	0.01	104	48.2	31.9
pbzip	5	269.3K	0.01	T.O.	15894	14471
Total	-	**43.8M**	-	**>21290**	**18521**	**16898**

6. Data-race detection in C11. Here we incorporate CSSTs in C11Tester, a data-race detector for the C11 memory model [23]. The analysis constructs incrementally a trace σ, by iteratively mapping a read to a write to obtain its value from. To make these choices consistent with the C11 memory model, it maintains a partial order P using VCs. The reachability queries are incremental. To ensure that all data structures process the same trace, we run them alongside each other on a single execution, and measure the time taken to maintain the partial order in each.

Our results are shown in Table 6. Interestingly, VCs are generally more performant than STs and CSSTs in this setting. Looking closer into each benchmark, we observe that new orderings lead to very little (typically none at all) propagation in P. Effectively, this makes VCs run in $O(1)$ time per update, and thus gain an advantage. In contrast, in readerswriters and atomicblocks C11Tester inserts non-trivial orderings in P (i.e., orderings that lead to propagation), and CSSTs are considerably more performant than VCs. Finally, the average density q is high in a few benchmarks, reaching even the value 1, which explains why CSSTs and STs have comparable performance, though CSSTs remain more performant.

Table 6. Race detection on C11.

benchmark	T	N	q	VCs† (s)	STs (s)	CSSTs (s)
dq	5	72.8K	0.48	2.5	2.8	3.1
mabain	7	101.6K	0.33	0.6	1.2	1.1
seqlock	18	693.9K	0.2	4.5	6.7	6.3
iris-1	13	1.1M	0.2	54.8	66.5	60.3
qu	11	1.3M	0.43	5.9	7.5	7.2
indexer	18	1.6M	1	1.0	1.1	1.1
exp-bug	5	2.5M	0.25	2.6	2.5	2.5
twalock	12	4.5M	0.34	29.6	53.3	49.0
gcd	4	4.5M	1	2.6	2.8	2.7
spinlock	12	4.8M	0.2	23.0	37.9	34.0
ttaslock	12	4.9M	0.25	26.7	48.0	42.7
silo	5	5.6M	0.01	5.9	6.2	6.1
fib-bench	4	6.0M	1	6.5	6.3	6.5
linuxrwlocks	7	6.9M	0.39	30.8	40.6	37.7
barrier	6	8.3M	0.51	19.6	24.5	23.7
mpmc	11	9.2M	0.24	37.6	44.9	42.6
spsc	4	9.7M	0.55	12.2	12.2	12.1
mcs-lock	12	9.9M	0.37	55.2	86.8	80.1
treiber	7	10.1M	0.11	31.3	36.6	35.7
iris-2	4	11.5M	0.2	15.5	14.7	14.7
gdax	8	13.5M	0.97	8.5	7.6	7.6
ticketlock	7	14.2M	0.48	45.1	70.4	64.2
mutex	12	15.6M	0.39	48.5	68.5	63.7
readerswriters	13	10.1M	0.33	16.3	7.6	7.5
atomicblocks	33	15.5M	0.5	9.7	1.7	1.6
Total	-	**172.4M**	-	**496**	**659**	**614**

7. Root-causing linearizability violations. Here we incorporate CSSTs in a recent root-cause analysis for linearizability violations [12]. The reachability queries are both incremental and decremental, hence the only baseline for maintaining the partial order are plain Graphs, as indeed used in [12]. We measure the total time taken in the analysis.

Our results are shown in Table 7. The experimental setup of [12] consists of three data structures, accessed an increasing number of times. CSSTs are typically orders of magnitude faster than plain Graphs used in [12], which strongly supports CSSTs as an efficient data structure also for analyses utilizing decremental partial-order updates.

Table 7. Root causing linearizability violation results.

benchmark	T	N	q	Graphs† (s)	CSSTs (s)
LogicalOrderingAVL	3	615	0.1	1.2	0.4
	3	1.6K	0.07	6.1	1.6
	3	2.6K	0.06	13.5	3.8
	3	5.0K	0.05	49.5	10.8
OptimisticListSorted-SetWaitFreeContains	3	219	0.19	1.4	0.2
	3	399	0.18	9.8	0.4
	3	759	0.17	114	0.9
	3	1.1K	0.17	512	1.7
RWLockCoarse-GrainedListIntSet	3	978	0.04	3.6	0.8
	3	1.6K	0.03	11.4	1.7
	3	3.2K	0.03	73.6	5.3
	3	4.8K	0.03	249	10.1
Total	-	**22.9K**	-	**1045**	**38**

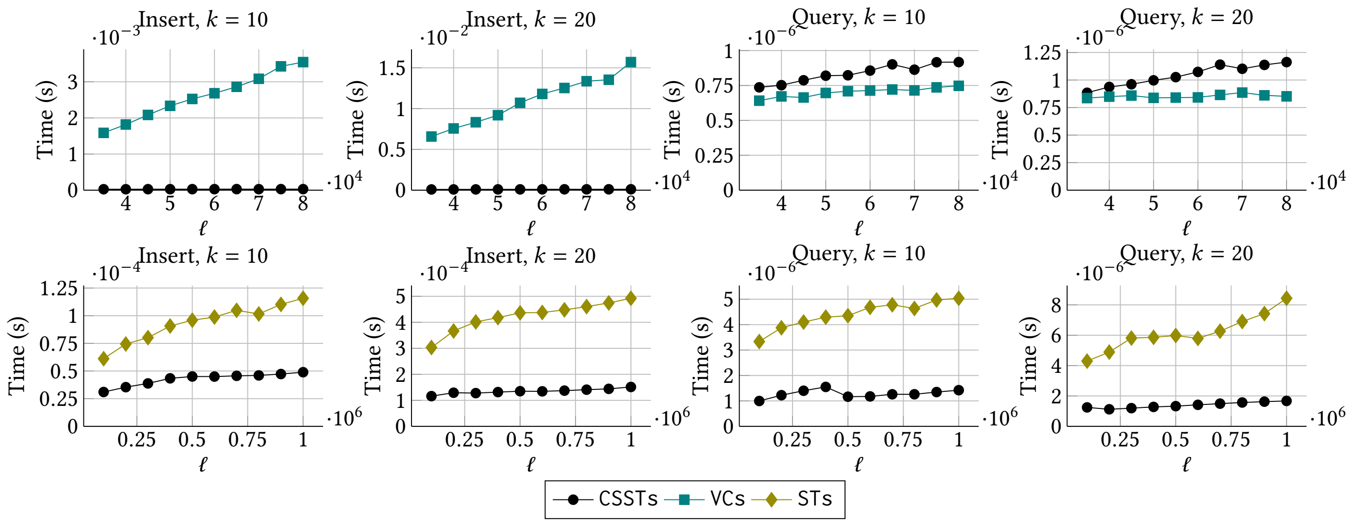


Figure 11. Scalability experiments on CSSTs, STs and VCs. Each partial order contains k chains and $\ell = n/k$ events per chain.

5.3 Scalability Analysis

Here we perform controlled scalability experiments on the performance of CSSTs for edge insertions and queries. We consider partial orders P in two settings, consisting of $k = 10, 20$ chains, and varying number of events ℓ in each chain. Initially, P has no cross-chain edges. First, to measure the performance of edge insertions, we repeatedly insert random cross-chain edges $\langle t, i\rangle \mapsto \langle t', j\rangle$ such that the two endpoints are unordered and $|i - j| \leq b$. The last condition captures the fact that cross-chain orderings are typically between events that execute within the same time-window. We report on window $b = 10^4$, though we have observed similar results with other values of b. This condition also prevents P from becoming close to total. We attempt to insert 20ℓ edges in each case, and report the average time of edge insertions. Second, to measure the performance of edge queries, we use the partial order P at the end of the above process. We make 10^6 random queries, and report the average time per query.

The results are shown in Figure 11. As expected by theory, VCs suffer linear complexity for insertions but achieve queries in constant time. On the other hand, CSSTs and STs achieve logarithmic complexity in both insertions and deletions. CSSTs are far more performant than VCs in edge insertions, but also achieve query times that are similar to VCs, though, naturally, VCs are slightly faster, as each query is a simple lookup. This is achieved by our minima indexing and sparse representation, which makes queries run with minimal overhead in practice. Finally, CSSTs are considerably faster than STs in both insertions and queries, an improvement stemming from the Sparse Segment Tree (Section 3.2).

6 Related Work

The efficient maintenance of partial orders is a key component in concurrent program analysis. Vector Clocks [28] were originally proposed as an efficient mechanism for causality tracking in distributed systems [21], and have been used extensively in program analysis. Their efficient implementation has been approached in various ways, e.g., using varying-size arrays [11], AVL trees [22], Chain Clocks [3], and Tree Clocks [25]. These variations offer performance improvements over vanilla Vector Clocks, but only when the partial order is built incrementally and in a streaming fashion.

For non-streaming updates, partial orders are normally represented as plain graphs [2, 6, 12, 30, 33, 41] and Vector Clocks [17, 23]. The closest data structure to CSSTs has been STs, developed in [31] for data-race detection, and subsequently used in other dynamic analyses [8, 35, 40]. The development of CSSTs lies on a few technical novelties. For example, our minima indexing and sparse tree representation (Section 3.2) are novel, and lead to tighter complexity bounds. Moreover, in order to handle the fully dynamic setting, CSSTs do not store transitive reachability. This requires overcoming the challenge of discovering transitive reachability during queries in an efficient manner (Section 3.3).

7 Conclusion

We have introduced CSSTs, a data structure for the fully-dynamic maintenance of partial orders with small width, supporting a plethora of dynamic analyses for concurrency. Our experimental results indicate that CSSTs are an efficient, drop-in replacement of existing approaches to maintaining partial orders in various application domains of recent literature. Our aim is for CSSTs to become a standard reference for driving future scalable dynamic analyses.

Acknowledgments

This work was partially supported by a research grant (VIL42117) from VILLUM FONDEN.

A Artifact Appendix

A.1 Abstract

This artifact contains all the source codes and experimental data required to replicate our evaluation in Section 5. In particular, it contains the following content: (i) source code of the data structures `CSSTs`, `STs`, `VCs`, and `Graphs` (ii) source code of the analyses used in generating Tables 1–7 and the corresponding experimental data. We additionally provide Python scripts that automate the process of replicating our results.

Besides the accompanying artifact, `CSSTs` are available in a repository [1] as a stand alone data structure for dynamic graph reachability that can be used in other settings.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Collective Sparse Segment Trees (CSSTs).
- **Run-time environment:** Docker.
- **Metrics:** Execution time.
- **Output:** CSV files.
- **How much disk space required (approximately)?:** 20 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)?:** 80 hours. We also provide the users with the option to run the experiments on a smaller set of benchmarks or use a shorter timeout.
- **Publicly available?:** Yes [38].
- **Code licenses (if publicly available)?:** MIT License.
- **Archived (provide DOI)?:** 10.5281/zenodo.10798906

A.3 Description

A.3.1 How to access.

Obtain the artifact from [38].

A.3.2 Hardware dependencies.

Replicating the results of large benchmarks requires up to 60 GB RAM. Moreover, we remark that users may encounter problems in running the artifact on Apple M1 silicon due to certain incompatibility issues in running Docker on these machines. There are otherwise no special hardware requirements.

A.3.3 Software dependencies.

Docker.

A.3.4 Data sets.

All data sets are provided with the artifact.

A.4 Installation

- Install Docker (https://www.docker.com).
- Obtain the artifact's Docker image from [38].
- Import the image:

```
> docker import asplos24-139.tar asplos24-139
```

- Start a container:

```
> docker run -it asplos24-139 bash
```

A.5 Experiment workflow

Figure 12 displays the directory structure of our artifact. The directory `src` contains the source code of the data structures `CSSTs`, `STs`, and `VCs`. The directory `experiment` is the primary folder for experimental evaluation. The subdirectories `analyses` and `data` contain the source code of the analyses and their corresponding experimental data, respectively. The directories `output` and `result` serve as destinations for the generated outputs and are initially empty. The file `experiment.py` defines the experimental workflow. Users do not need to interact directly with this program. The files `run.py` and `compile_results.py` are helper scripts that interface with `experiment.py` to execute the experiments and generate results in a tabular format.

```
root/
|-- src/
|-- experiment/
    |-- analyses/
    |-- data/
    |-- output/
    |-- result/
    |-- experiment.py
    |-- run.py
    |-- compile_results.py
```

Figure 12. Directory structure of the artifact

A.6 Evaluation and expected results

Running the script `run.py` will execute all the experiments.

```
> python3 run.py
```

The generated raw outputs can be located under the folder `output`. The file `compile_results.py` can be used to generate outputs in a tabular format.

```
> python3 compile_results.py
```

This will generate a `CSV` file for each analyses under the folder `results`. The contents of these files can be displayed in the console as follows.

```
> csvtool readable /root/experiments/results/<NAME>.csv
```

The main goal of this evaluation is to measure the performance benefits of `CSSTs` over `STs`, `VCs` and `Graphs` in various dynamic concurrency analyses. We expect that for each analyses the overall speedups would remain similar to the results reported in Tables 1–7.

A.7 Experiment customization

Users may customize experiments by passing arguments to the script `run.py`. For instance, executing the following command will run the experiments on a smaller subset of benchmarks:

```
> python3 run.py -s 1
```

Whereas the following command sets the timeout for each individual experiment to 30 minutes.

```
> python3 run.py -t 30m
```

For a comprehensive list of supported options, please refer to the documentation accompanying the script.

```
> python3 run.py --help
```

References

[1] [n.d.]. CSSTs GitHub Repository. https://github.com/hcantunc/cssts.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 29. https://doi.org/10.1145/3360576

[3] Anurag Agarwal and Vijay K. Garg. 2005. Efficient Dependency Tracking for Relevant Events in Shared-Memory Systems. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing* (Las Vegas, NV, USA) *(PODC '05)*. Association for Computing Machinery, New York, NY, USA, 19–28. https://doi.org/10.1145/1073814.1073818

[4] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. 2021. Stateless Model Checking Under a Reads-Value-From Equivalence. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Vol. 12759. Springer International Publishing, Cham, 341–366. https://doi.org/10.1007/978-3-030-81685-8_16

[5] Lars Arge, Johannes Fischer, Peter Sanders, and Nodari Sitchinava. 2013. On (Dynamic) Range Minimum Queries in External Memory. In *Algorithms and Data Structures*, Frank Dehne, Roberto Solis-Oba, and Jörg-Rüdiger Sack (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 37–48. https://doi.org/10.1007/978-3-642-40104-6_4

[6] 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)*. Association for Computing Machinery, New York, NY, USA, 28–39. https://doi.org/10.1145/2594291.2594323

[7] Truc Lam Bui, Krishnendu Chatterjee, Tushar Gautam, Andreas Pavlogiannis, and Viktor Toman. 2021. The Reads-from Equivalence for the TSO and PSO Memory Models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485541

[8] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and Efficient Concurrency Bug Prediction. 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, 255–267. https://doi.org/10.1145/3468264.3468549

[9] Soham Chakraborty, Shankara Narayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2024. How Hard Is Weak-Memory Testing? *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 66:1978–66:2009. https://doi.org/10.1145/3632908

[10] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2018. Data-Centric Dynamic Partial Order Reduction. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 1–30. https://doi.org/10.1145/3158119

[11] Mark Christiaens and Koenraad De Bosschere. 2001. Accordion Clocks: Logical Clocks for Data Race Detection. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing (Euro-Par '01)*. Springer-Verlag, Berlin, Heidelberg, 494–503. https://doi.org/10.1007/3-540-44681-8_73

[12] Berk Çirisci, Constantin Enea, Azadeh Farzan, and Suha Orhun Mutluergil. 2020. Root Causing Linearizability Violations. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 350–375. https://doi.org/10.1007/978-3-030-53288-8_17

[13] Ariel Eizenberg, Yuanfeng Peng, Toma Pigli, William Mansky, and Joseph Devietti. 2017. BARRACUDA: Binary-Level Analysis of Runtime RAces in CUDA Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 126–140. https://doi.org/10.1145/3062341.3062342

[14] Michael Emmi and Constantin Enea. 2019. Weak-Consistency Specification via Visibility Relaxation. *Proc. ACM Program. Lang.* 3, POPL (2019), 28. https://doi.org/10.1145/3290373

[15] 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)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[16] 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)*. Association for Computing Machinery, New York, NY, USA, 293–303. https://doi.org/10.1145/1375581.1375618

[17] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 603–616. https://doi.org/10.1145/3575693.3575729

[18] Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

[19] Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

[20] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 29. https://doi.org/10.1145/3276516

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

[22] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[23] Weiyu Luo and Brian Demsky. 2021. C11Tester: A Race Detector for C/C++ Atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 630–646. https://doi.org/10.1145/3445814.3446711

[24] 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 (2018), 29. https://doi.org/10.1145/3276515

[25] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A Tree Clock Data Structure for Causal Orderings in Concurrent Executions. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association

for Computing Machinery, New York, NY, USA, 710–725. https://doi.org/10.1145/3503222.3507734

[26] 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*. ACM, Saarbrücken Germany, 713–727. https://doi.org/10.1145/3373718.3394783

[27] 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

[28] 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.

[29] 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, USA, 267–280.

[30] Brian Norris and Brian Demsky. 2013. CDSchecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 131–150. https://doi.org/10.1145/2509136.2509514

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

[32] 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 (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 747–762. https://doi.org/10.1145/3385412.3385993

[33] 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)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

[34] Amitabha Roy, Stephan Zeisset, Charles J. Fleckenstein, and John C. Huang. 2006. Fast and Generalized Polynomial Time Memory Consistency Verification. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 503–516. https://doi.org/10.1007/11817963_46

[35] Zheng Shi, Umang Mathur, and Andreas Pavlogiannis. 2024. Optimistic Prediction of Synchronization-Reversal Data Races. https://doi.org/10.48550/arXiv.2401.05642

[36] Hünkar Can Tunç, Parosh Aziz Abdulla, Soham Chakraborty, Shankaranarayanan Krishna, Umang Mathur, and Andreas Pavlogiannis. 2023. Optimal Reads-From Consistency Checking for C11-Style Memory Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 137:761–137:785. https://doi.org/10.1145/3591251

[37] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI (2023), 26. https://doi.org/10.1145/3591291

[38] Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Cirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis. (2024). https://doi.org/10.5281/zenodo.10798906 Artifact.

[39] Hünkar Can Tunç, Ameya Prashant Deshmukh, Berk Çirisci, Constantin Enea, and Andreas Pavlogiannis. 2024. CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis. https://doi.org/10.48550/arXiv.2403.17818 Technical Report.

[40] 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

[41] Rachid Zennou, Mohamed Faouzi Atig, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2020. Boosting Sequential Consistency Checking Using Saturation. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 360–376. https://doi.org/10.1007/978-3-030-59152-6_20

[42] Rachid Zennou, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2019. Gradual Consistency Checking. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 267–285. https://doi.org/10.1007/978-3-030-25543-5_16