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

Abstract
Dynamic analyses are a standard approach to analyzing and testing concurrent programs. Such techniques observe program traces \( \sigma \) 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 \( \sigma \). 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 \( \sigma \), 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

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 \( \sigma \). The analysis undergoes a sequence of interleaved operations of (i) inserting new orderings \( e_1 \rightarrow e_2 \) in \( P \), (ii) querying whether \( e_1 \rightarrow e_2 \) in \( P \), and even (iii) deleting existing orderings \( e_1 \rightarrow e_2 \) from \( P \), as it attempts to prove that \( \sigma \) 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 \( \sigma \) 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 \( G \)
transiently closed after each update. This bound is pro-
hibitively large for typical values of $n$ in the range $10^4$-$10^{10}$.

The scalability of dynamic analyses has been driven pri-
marily by a simple data structure known as Vector Clocks [28],
based on the realization that (i) each $\sigma$ 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 summa-
rized 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 inser-
tion cost of Vector Clocks is further reduced to $O(k)$. Here 
the events of $\sigma$ are processed one by one, and new order-
ings $e_1 \rightarrow e_2$ 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 atomic-
city 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 repre-
sent 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 $\sigma$. 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 depen-
dency 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 Fig-
ure 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 inter-
leaving 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 $w(x, v) \rightarrow r(x, v)$, 
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 
y 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 mark-
ing the current reads-from map. The analysis proceeds on 
the read event $e_2: r(x, 3)$. It first inserts in $P$ the program or-
der 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 $e_5$ 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 $e_5$ — the process of 
infering 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 $e_4$ 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 
especially 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 \( \delta \) 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, \( \delta \) 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 mental 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 \( \sigma \) of concurrent programs, representing the history of operations in an execution. We write \( \text{Events}_\sigma \) to denote the set of events occurring in \( \sigma \). Each event of \( \sigma \) is a tuple \( e = (t, i, m) \), where \( t \) is an identifier of the thread that performed \( e \) and \( i \) is the sequence id of \( e \). The pair \((t, i)\) 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 \((t, i)\), while the meta information \( m \) will be used only in our examples. We write \( \text{tid}(e) \), \( \text{id}(e) \) for the thread identifier and sequence id of \( e \), respectively. Depending on the application, the events in \( \sigma \) 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 antisymmetric binary relation defined on the elements of a given set \( S \). We denote by \( \leq_p \) a partial order \( P \) over the set \( \text{Events}_\sigma \). Given \( e_i, e_j \in \text{Events}_\sigma \), we write \( e_i \leq_p e_j \) to represent \((e_i, e_j) \in \leq_p \). We write \( e_i <_p e_j \) to represent \( e_i \leq_p e_j \) and \( e_i \neq e_j \). The program order \( \leq_{po} \) represents a partial order where \( e_i \leq_{po} e_j \) iff \( \text{id}(e_i) \leq \text{id}(e_j) \) and \( \text{tid}(e_i) = \text{tid}(e_j) \). We write width\((P)\) to represent the largest size of a set \( S \subseteq \text{Events}_\sigma \) such that for all distinct pairs \( e_i, e_j \in S \), we have that \( e_i \neq_{po} e_j \) and \( e_i \neq_{po} e_j \).

In concurrency analyses, 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 \( \leq_{po} \leq_{po} \leq_p \), implying that width\((P)\) \leq \( k \), where \( k \) is the number of threads. Although in other settings (e.g., those involving weak-memory concurrency) width\((P)\) might be larger than \( k \), it still remains bounded by some small factor of \( k \). For example, analyses involving TSO normally satisfy width\((P)\) \leq 2\( k \) (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 \( u = (t, i) \in [k] \times [n] \), where \([j] = \{0, 1, \ldots, j-1\} \). The set of edges normally captures the program order, which is reflected in the fact that for any two nodes \((t, i), (t, i + 1) \in V \), we have \((t, i), (t, i + 1) \in E \). We can thus represent \( G \) as \( k \) totally-ordered chains with additional edges across chains, and call such graphs chain DAGs. A node \( u \) is said to be earlier than another node in the same chain \((t, j)\) if \( i \leq j \), and later otherwise. Then, \( G \) can be written as the composition of \( k(k-1) \) subgraphs \( \text{Sub}(G)^{\text{ch}}_{k-1} \) 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.
Collective Sparse Segment Trees

Consider a 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. A 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 \( u \rightarrow^* v \).
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 \( \delta \) and cross-chain density \( d \). CSSTs maintain \( G \) under dynamic updates, costing \( O(\max(\log \delta, \log(\log n, d))) \) time per update and \( O(k^2 \log(\log n, d)) \) 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 in realizing 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 \( A[i] \) and \( A[i:j] \) to denote the value of \( A \) at index \( i \) and the sequence of values \( A[i], \ldots, A[j] \), respectively, and write \( A[i:] \) as shorthand for \( A[i:|A| - 1] \). We say that \( A \) is empty on index \( i \) if \( A[i] = \infty \).

The density of \( A \) is the number of non empty entries of \( A \).

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

1. min \( \{A[i], \alpha\} \), returning \( \min_{i \leq j \leq \alpha} A[j] \), i.e., the minimum value in the suffix \( A[i:] \).
2. argleq \( A[i], \alpha \), returning \( \max_{i \leq j \leq \alpha} \{A[j], i\} \), i.e., the largest index \( i \) of \( A \) such that \( A[i] \leq \alpha \); and
3. \( set(A, i, \alpha) \), setting \( A[i] = \alpha \).

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 \( nd_{ij} \), each associated with a range \( [i, j] \). Each \( nd_{ij} \) maintains a value \( nd_{ij}.\min = \min\{A[i:j]\} \). We write \( \text{depth}(nd_{ij}) \) to denote the depth of \( nd_{ij} \) in \( T \). We let \( T.\text{root} \) 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.

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

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

**Dynamic reachability on \( k = 2 \) chains.** Recall that a chain DAG \( G \) of \( k \) chains is the composition of \( k \) \((k - 1)\) subgraphs \( \text{Sub}(G_{ij}^k) \) (Figure 2). Now assume that \( k = 2 \). We represent each \( \text{Sub}(G_{ij}^2) \) as a graph \( A_{ij}^2 : [n] \rightarrow [n] \cup \{\infty\} \). \( A_{ij}^2[j_i] \) stores...
a unique neighbor of node \((t_1, j_1)\) in chain \(t_2\). What if \((t_1, j_1)\) has multiple neighbors to \(t_2\) then? It suffices to store the earliest neighbor of \((t_1, j_1)\) 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

\[
A_{t_1}^{t_2} = \min\{j : (t_1, j_1) \rightarrow (t_2, j_2)\}
\]

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

How can we maintain the invariant Eq. (1) under edge insertions and deletions? We achieve this by storing all current successor \((t_1, j_1)\) to chain \(t_2\). However, a path \((t_1, j_1) \rightarrow ^* (t_2, j_2)\) in \(G\) might not be the minimum value in \(A_{t_1}^{t_2}\) \(j_2\). 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-\(\infty\) entries it has), stemming from the observation that \(\infty\) 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 \(nd_{i,j}\) of a Sparse Segment Tree \(T\) maintains a field \(pos\) \(\in [i, j]\), that is defined recursively: it points to the largest index of \(A[i : j]\), after excluding all indexes that are stored in the pos fields of all ancestors of \(nd_{i,j}\). Formally, let \(nd_1, \ldots, nd_m\) be the ancestors of \(nd_{i,j}\) in \(T\). Then

\[
nd_{i,j}.pos = \max_{\ell \in [i,j]}\left(\arg\min_{\substack{\ell \in [i,j]\{\text{nd}_{\ell,\text{pos},\ldots,\text{nd}_{m,\text{pos}}}}\right) A[\ell]
\]

Handing \(k > 2\) chains. Naturally, when \(k > 2\), we can have a suffix minima array \(A_{t_1}^{t_2}\) for every pair of threads \(t_1, t_2\) \(\in [k]\) with \(t_1 \neq t_2\). However, a path \((t_1, j_1) \rightarrow ^* (t_2, j_2)\) might go through intermediate nodes \((t_2, j_1)\), i.e., reachability is formed transitively across chains (e.g., the path \(e_6 \rightarrow ^* e_2\) 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 \(A_{t_1}^{t_2}\) 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, \(A_{t_1}^{t_2}\) 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 \(A[i : j]\). 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.

**Example 2.** Consider again the partial order \(\preceq_{pos}^G\) in Figure 1a. Some of the arrays \(A_{t_1}^{t_2}\) are shown in Figure 4. Observe that successor\((e_2, 2) = 1\) can be determined by the suffix minima query \(\min(A_{t_1}^{t_2}, 0)\). Similarly, predecessor\((e_2, 1) = 5\) can be determined by the suffix arg-minima query \(\arg\leq\min(A_{t_1}^{t_2}, 1)\).

**Example 3.** Consider the array \(A\) in Figure 5a, and part of the corresponding ST in Figure 5b. We have \(nd_{0,7}.pos = 1 = \arg\min A[1]\). A query \(\min A[0]\) is now directly answered at the root, as \(nd_{0,7}.pos \in [0 : 7]\). On the other hand, the query \(\min A[2]\) is not answered at the root as \(nd_{0,7}.pos \notin [2 : 7]\). The procedure then traverses its child \(nd_{0,3}\). We have \(nd_{0,3}.pos = 3\),
one non-empty entry, at index \( A \). Also true: every non-empty entry of \( A \) of the tree. Every intermediate node \( \text{SST} \) of the tree to shrink, which consequently results in more Sparse representation. Sparse Segment Trees are tailored for the smallest element in \( A \). The traversal stops here, as \( n_{d,l}. \text{pos} \) is larger than a value deeming that the node \( n_{d,l} \) is the root, the function checks and deletes if \( n_{d,l} \) is 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 \( n_{d,i,j} \), if \( \text{depth}(n_{d,i,j}) \) 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 \( n_{d,i,j} \) a block node. This representation is valuable when the array contains segments which are densely populated but also far apart from each other.

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 \( n_d \) is such that \( n_d. \text{pos} \) 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 \( n_{d,2,2} \) becomes the left child of the root, with its pos and min fields set according to Eq. (2). Again, in the standard \( T \) representation, \( n_{d,2,2} \) would have nodes \( n_{d,0,3} \) and \( n_{d,0,2} \) as ancestors, but these nodes are never created in our SST. Figure 6g, shows another update of \( T \) after setting \( A[0] = 59 \). As \( 59 \leq n_{d,0,7}. \text{min} \), the root remains intact. The index 0 belongs to the left subtree of the root and 59 < \( n_{d,2,2}. \text{min} \). This causes the new entry to be inserted to a fresh node \( n_{d,0,3} \) and assigned as the left child of the root, taking the existing node \( n_{d,2,2} \) 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.

Example 5. Consider the array \( A \) and its corresponding SST in Figure 7. \( A \) contains only a single entry in the range 0, . . . , 31. This entry is sparsely represented in the left subtree of the root. On the other hand, \( A \) is dense in the range 32, . . . , 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, arg\_leq on a SST. Function update \((n_d, pos, v)\). This function inserts a node \( n_d' \) into the subtree of \( n_d \), with \( n_d'. \text{pos} = pos \) and \( n_d'. \text{min} = v \). It first checks if \( n_d \) is initialized; if not, it creates a new node (Line 2). If \( n_d \) is the root, the function checks and deletes if a node \( n_d'' \) exists in the subtree of \( n_d \) where \( n_d''. \text{pos} = pos \) (Line 3). Then, it checks if \( 2^{\text{depth}(n_d)} \geq n_d. \text{pos} \) (Line 4). Otherwise, it checks if \( v \leq n_d. \text{min} \). If so, it swaps the values of \( n_d. \text{pos} \) and \( n_d. \text{min} \) with \( pos \) and \( v \) (Line 7). This captures that
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 $\infty$ if nd is nil or $i > \text{nd.end}$ (Line 19), which captures the end of the recursion. Otherwise, it returns the smallest $\min$ value in the subtree of nd whose corresponding $\min$ 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. First, it 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 $A$ 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., $\neq \infty$) 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 $A_{t_i}$ storing direct edges from $t_1$ to $t_2$, together with a min-heap $\text{heap}_{t_i}^{\text{min}}[j_1]$, for each $j_1 \in [n]$, as described in Section 3.1. Suffix-minima queries on $A_{t_i}$ 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 $A_{t_i}$ and min-heaps $\text{heap}_{t_i}^{\text{min}}[j_1]$. 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 $(t_1, j_1)$ to discover transitive reachability across chains. This is done by repeated
Algorithm 2: Fully dynamic CSSTs.

```plaintext
function successor((t₁, j₁), t₂)
for each t'₂ ∈ [k] \ {t₁}
do
    closure[t'₂] ← min(A_{t'₂}^{j₁}, j₁)
do
    changed ← false
for each t'₂ ∈ [k] \ {t₁}
do
    if t'₂ ≠ t₂ then
        Let v ← min(A_{t'₂}^{j₁}, closure[t'₂])
        if v < closure[t'₂] then
            closure[t'₂] ← v
            changed ← true
while changed
return closure[t₂]
```

function predecessor((t₁, j₁), t₂)

// Similar to successor

function reachable((t₁, j₁), (t₂, j₂))
if successor((t₁, j₁), t₂) ≤ t₂ then return True
else return False

function insertEdge((t₁, j₁), (t₂, j₂))
if j₂ < edgeHeap_{t₁}^{j₁}[j₁].min then update(A_{t₂}^{j₁}, j₁, j₂)
edgeHeap_{t₁}^{j₁}[j₁].insert(j₂)

function deleteEdge((t₁, j₁), (t₂, j₂))
if j₂ = edgeHeap_{t₁}^{j₁}[j₁].min then
    edgeHeap_{t₁}^{j₁}[j₁].delete(j₂)
else edgeHeap_{t₁}^{j₁}[j₁].delete(j₂)
```

Figure 8. Query successor((0,0), 3).

min queries in the suffix minima arrays $A_{t₂}^{j₁}$ (Line 9), which find the earliest node of $t'₂$ that has an edge from the currently earliest node of $t'₂$ found to be reachable from $(t₁, j₁)$. 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((0,0), 3). The first step is identifying the earliest successors of (0,0) reachable via direct edges from (0,0) and its thread-local successors, in all other threads. This discovers (1,0) via 1 and (3,2) via 2. While (3,2) is indeed in chain 3, it is not necessarily the earliest successor of (0,0) (indeed, the earliest successor is (3,1)). Hence the function proceeds iteratively for each newly discovered node until reaching a fixed point. This leads to the discovery of (2,1) via 3 and eventually the discovery of (3,1) via 4.

Analysis. We now state the properties of fully dynamic CSSTs. First, since the arrays $A_{t₂}^{j₁}$ 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 $A_{t₂}^{j₁}$ is bounded by $d$.

The use of min-heaps guarantees the following invariant.

Lemma 3 (Invariant). For every distinct $t₁, t₂ ∈ [k]$, we have $A_{t₂}^{j₁}[j₁] = min \{j₂ ∈ [n]. (t₁, j₁) → (t₂, j₂)\}$.

A crossing path is a sequence of nodes $π: (t₀, j₀), \ldots, (t_{m−1}, j_{m−1})$ such that for each $i ∈ [m−2]$ we have $tᵢ ≠ tᵢ₊₁$ and there are $i₁, i₂ ∈ [n]$ such that (i) $i₁ ≥ jᵢ$, (ii) $i₂ ≤ jᵢ₊₁$, and (iii) $(tᵢ, i₁) → (tᵢ₊₁, i₂)$. Clearly, for any two nodes $(t₁, j₁)$ and $(t₂, j₂)$, if $(t₁, j₁) →^* (t₂, j₂)$, 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 $i ∈ [k]$, after the $i$-th iteration of the do-while loop in Line 4, closure[t'₁] points to the earliest node in chain $t'$ that is reachable from $(t₁, j₁)$ via a crossing path of length $≤ i + 1$.

Regarding the time complexity, due to Lemma 2 and Lemma 1, each operation in a suffix minima array $A_{t₂}^{j₁}$ takes $O(\min(\log n, d))$ time. Since the maximum out-degree is $δ$, each operation on a min heap $\text{edgeHeap}_{t₂}^{j₁}[j₁].min$ runs in $O(\log δ)$ time. Thus, updates take $O(\max(\log δ, \min(\log n, d)))$ time, while, due to Lemma 4, queries take $O(\delta^3 \min(\log n, d))$ 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 $\text{edgeHeap}_{t₂}^{j₁}[j₁]$, as edge deletions require to remember all edges added so far. The suffix minima arrays $A_{t₂}^{j₁}$ 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 $\leq k$ other chains. Naturally, $d ≤ n$, 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 $d ≪ n$, 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
Consider a chain DAG $G$. Theorem 2.

(a) A chain DAG $G$.

(b) Array $A^0_0$.

(c) Array $A^2_0$.

(d) Array $A^3_0$

Figure 9. Inserting an edge $\langle 1, 1 \rangle \rightarrow (2, 0)$ and inferring the transitive path $(0, 1) \rightarrow (3, 2)$.

In the following 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 $A^i_j$, with $i \neq j$, each implemented using an SST $T^i_j$. These arrays now store transitive reachability, as opposed to direct edges between chains. Algorithm 3 shows how each array $A^i_j$ is queried and updated, following the reachability queries and edge insertions in CSSTs. The functions $\text{successor}(t_i, j_1, j_2)$, $\text{predecessor}(t_i, j_1, j_2)$ and $\text{reachable}(t_i, j_1, j_2)$ handle successor, predecessor, and reachability queries in a straightforward manner: since each $A^i_j$ is transitively closed, these operations reduce to suffix-minima queries on $A^i_j$.

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

**Example 7.** Consider the chain DAG in Figure 9a and the operation $\text{insertEdge}(1, 1, 2, 0)$. We obtain the transitive path $(0, 1) \rightarrow (3, 2)$ as follows. First, the predecessor of $(1, 1)$ in thread 0 is identified via the query $\text{predecessor}(1, 1, 0)$. This query performs an $\text{argleq}$ operation on $A^1_j$ (Figure 9b) with value 1, returning node $(0, 1)$. Then, the successor of $(2, 0)$ in thread 3 is identified via the query $\text{successor}(2, 0, 3)$. This query performs a $\min$ operation on $A^3_0$ (Figure 9c) with index 0, and returns the node $(3, 2)$. Next, the update operation on $A^3_0$ adds the transitive edge, resulting in $A^3_0$ in Figure 9d.

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

**Lemma 5 (Soundness).** $A^i_j[j_1] = j_2 \Rightarrow (t_i, j_1) \rightarrow^* (t_2, j_2)$.

**Lemma 6 (Completeness).** $(t_i, j_1) \rightarrow^* (t_2, j_2)$ and $t_i \neq t_2 \Rightarrow \exists j_1' \geq j_1. A^i_j[j_1'] \leq j_2$.

Recall that the key feature of SSTs is their ability to handle sparse arrays efficiently. As incremental CSSTs insert transitive edges in the arrays $A^i_j$, 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 $A^i_j$ 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 $A^i_j$ 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 $A^i_j$ is bounded both by $\log n$ and by the cross-chain density of $G$. We thus arrive at Theorem 2.
As some tools are not public, we implemented their analysis. Vector Clocks and STs are the fully dynamic setting we compare the data structure used in the respective paper is marked by †. CSSTs, as introduced in this paper.

Most analyses make only incremental updates, in which case 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, by design transitively closed [28].
- **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 edge $e_1 \rightarrow e_2$ 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 $e_1 \rightarrow e_2'$. 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.

**Space usage.** Due to Lemma 7, the total space used by the suffix minima arrays $A^k_i$ 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.

### 5.2 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 analysis concerns the whole analysis, not just the corresponding data structure. 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 $q$ which might be data race free, and attempts to permute them to new traces $q'$ 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 unscaleable 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

---

1The 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 |
| bfswriter | 9 | 22.3K | 0.2 | 130 | 24.6 | 12.9 |
| ftsparser | 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 | T.O. | T.O. | 197 |
| jigsaw | 15 | 3.1M | 0.06 | 154 | 45.6 | 32.0 |
| sunflow | 17 | 11.7M | 0.01 | T.O. | 780 | 505 |
| xalan | 6 | 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 | -591992 | 732111 | 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 incrementally 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 |
| Hed | 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 | T.O. | 15.1 | 13.9 |
| lbzip2 | 11 | 9.3M | 0.04 | 273 | 96.6 | 46.9 |
| x264 | 7 | 10.4M | 0.03 | 474 | 165 | 62.0 |
| libvpx | 2 | 190.0M | 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
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.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>T</th>
<th>N</th>
<th>q</th>
<th>VCs (s)</th>
<th>STs (s)</th>
<th>CSSTs (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bbuf</td>
<td>3</td>
<td>27.7K</td>
<td>0.02</td>
<td>426</td>
<td>459</td>
<td>415</td>
</tr>
<tr>
<td>BoundedBuffer</td>
<td>11</td>
<td>325.7K</td>
<td>0.02</td>
<td>835</td>
<td>736</td>
<td>679</td>
</tr>
<tr>
<td>DiningPhil</td>
<td>21</td>
<td>1.4M</td>
<td>0.01</td>
<td>1559</td>
<td>1191</td>
<td>1110</td>
</tr>
<tr>
<td>fangerdi-ck</td>
<td>5</td>
<td>93.4K</td>
<td>0.06</td>
<td>366</td>
<td>1194</td>
<td>190</td>
</tr>
<tr>
<td>ksort</td>
<td>6</td>
<td>41.7M</td>
<td>0.01</td>
<td>104</td>
<td>48.2</td>
<td>31.9</td>
</tr>
<tr>
<td>pbzip</td>
<td>5</td>
<td>269.3K</td>
<td>0.01</td>
<td>T.O.</td>
<td>15894</td>
<td>14471</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-42199</td>
<td>18521</td>
<td>16898</td>
</tr>
</tbody>
</table>

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 readers-writers 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.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>T</th>
<th>N</th>
<th>q</th>
<th>VCs (s)</th>
<th>STs (s)</th>
<th>CSSTs (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dq</td>
<td>5</td>
<td>72.8K</td>
<td>0.48</td>
<td>2.5</td>
<td>2.6</td>
<td>3.1</td>
</tr>
<tr>
<td>mabain</td>
<td>7</td>
<td>16.5K</td>
<td>0.33</td>
<td>0.6</td>
<td>1.2</td>
<td>1.1</td>
</tr>
<tr>
<td>seqlock</td>
<td>18</td>
<td>693.9K</td>
<td>0.2</td>
<td>4.5</td>
<td>6.7</td>
<td>6.3</td>
</tr>
<tr>
<td>iris-1</td>
<td>13</td>
<td>1.1M</td>
<td>0.52</td>
<td>54.8</td>
<td>66.5</td>
<td>60.3</td>
</tr>
<tr>
<td>qu</td>
<td>11</td>
<td>1.3M</td>
<td>0.43</td>
<td>5.9</td>
<td>7.5</td>
<td>7.2</td>
</tr>
<tr>
<td>indexer</td>
<td>18</td>
<td>1.6M</td>
<td>1</td>
<td>1.0</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>exp-bug</td>
<td>5</td>
<td>2.5M</td>
<td>0.25</td>
<td>2.6</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>twalock</td>
<td>12</td>
<td>4.5M</td>
<td>0.34</td>
<td>29.6</td>
<td>53.3</td>
<td>49.0</td>
</tr>
<tr>
<td>gcl</td>
<td>4</td>
<td>4.5M</td>
<td>1</td>
<td>2.6</td>
<td>2.8</td>
<td>2.7</td>
</tr>
<tr>
<td>spinlock</td>
<td>12</td>
<td>4.8M</td>
<td>0.2</td>
<td>23.0</td>
<td>37.9</td>
<td>34.0</td>
</tr>
<tr>
<td>ttaslock</td>
<td>12</td>
<td>4.9M</td>
<td>0.25</td>
<td>26.7</td>
<td>48.0</td>
<td>42.7</td>
</tr>
<tr>
<td>silo</td>
<td>5</td>
<td>5.6M</td>
<td>0.01</td>
<td>5.9</td>
<td>6.2</td>
<td>6.1</td>
</tr>
<tr>
<td>fib-bench</td>
<td>4</td>
<td>6.0M</td>
<td>1</td>
<td>6.5</td>
<td>6.3</td>
<td>6.5</td>
</tr>
<tr>
<td>linuxrwlocks</td>
<td>7</td>
<td>6.9M</td>
<td>0.39</td>
<td>30.8</td>
<td>40.6</td>
<td>37.7</td>
</tr>
<tr>
<td>barrier</td>
<td>6</td>
<td>8.3M</td>
<td>0.51</td>
<td>19.6</td>
<td>24.5</td>
<td>23.7</td>
</tr>
<tr>
<td>mmpc</td>
<td>11</td>
<td>9.2M</td>
<td>0.24</td>
<td>37.6</td>
<td>44.9</td>
<td>42.6</td>
</tr>
<tr>
<td>spsc</td>
<td>9</td>
<td>9.4M</td>
<td>0.55</td>
<td>12.2</td>
<td>12.2</td>
<td>12.1</td>
</tr>
<tr>
<td>mcs-lock</td>
<td>12</td>
<td>9.9M</td>
<td>0.37</td>
<td>55.2</td>
<td>86.8</td>
<td>80.1</td>
</tr>
<tr>
<td>treiber</td>
<td>7</td>
<td>10.1M</td>
<td>0.11</td>
<td>31.3</td>
<td>36.6</td>
<td>35.7</td>
</tr>
<tr>
<td>gax</td>
<td>8</td>
<td>13.5M</td>
<td>0.97</td>
<td>8.5</td>
<td>7.6</td>
<td>7.6</td>
</tr>
<tr>
<td>ticketlock</td>
<td>7</td>
<td>14.2M</td>
<td>0.48</td>
<td>45.1</td>
<td>70.4</td>
<td>64.2</td>
</tr>
<tr>
<td>mutex</td>
<td>12</td>
<td>16.5M</td>
<td>0.39</td>
<td>48.5</td>
<td>68.5</td>
<td>63.7</td>
</tr>
<tr>
<td>readers-writers</td>
<td>13</td>
<td>10.1M</td>
<td>0.33</td>
<td>16.3</td>
<td>7.6</td>
<td>7.5</td>
</tr>
<tr>
<td>atomicblocks</td>
<td>33</td>
<td>15.5M</td>
<td>0.5</td>
<td>9.7</td>
<td>1.7</td>
<td>1.6</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-1724M</td>
<td>496</td>
<td>659</td>
</tr>
</tbody>
</table>

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 is 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.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>T</th>
<th>N</th>
<th>q</th>
<th>Graphs (s)</th>
<th>CSSTs (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogicalOrderingAVL</td>
<td>3</td>
<td>615</td>
<td>0.1</td>
<td>1.2</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.6K</td>
<td>0.07</td>
<td>6.1</td>
<td>1.6</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2.6K</td>
<td>0.06</td>
<td>13.5</td>
<td>3.8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5.0K</td>
<td>0.05</td>
<td>49.5</td>
<td>10.8</td>
</tr>
<tr>
<td>OptimisticListSorted-SetWaitFreeContains</td>
<td>3</td>
<td>219</td>
<td>0.19</td>
<td>1.4</td>
<td>0.2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>399</td>
<td>0.18</td>
<td>9.8</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>759</td>
<td>0.17</td>
<td>114</td>
<td>0.9</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.1K</td>
<td>0.17</td>
<td>512</td>
<td>1.7</td>
</tr>
<tr>
<td>RWLockCoarse-GrainedListIntSet</td>
<td>3</td>
<td>978</td>
<td>0.04</td>
<td>3.6</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1.6K</td>
<td>0.03</td>
<td>11.4</td>
<td>1.7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3.2K</td>
<td>0.03</td>
<td>73.6</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4.8K</td>
<td>0.03</td>
<td>249</td>
<td>10.1</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-22.9K</td>
<td>1045</td>
</tr>
</tbody>
</table>
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 \( \ell \) 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 \((i, j) \mapsto (i', j')\) 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 \( 20\ell \) 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 \( 10^\ell \) 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


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

https://doi.org/10.1145/3373718.3394783

https://doi.org/10.1145/3373736.3378475


https://doi.org/10.1145/2509136.2509514

https://doi.org/10.1145/3371085

https://doi.org/10.1145/3385412.3385993

https://doi.org/10.1145/3192366.3192385

https://doi.org/10.1007/11817963_46

https://doi.org/10.48550/arXiv.2401.05642

https://doi.org/10.1145/3591251

https://doi.org/10.1145/3591291


https://doi.org/10.1145/3468264.3468572

https://doi.org/10.1007/978-3-030-59152-6_20

https://doi.org/10.1007/978-3-030-25543-5_16