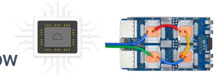
CSSTs: A Dynamic Data Structure for Partial Orders in Concurrent Execution Analysis

Hünkar Can Tunç Ameya Deshmukh Berk Çirişci

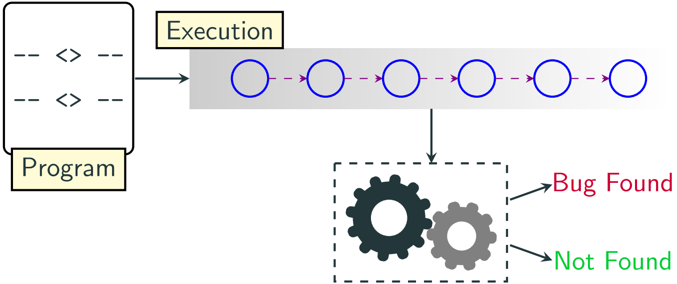
Constantin Enea Andreas Pavlogiannis

Concurrency: Software and Challenges

- Concurrency is everywhere
- Concurrency bugs are also everywhere
  - ↪ Data races
  - ↪ Deadlocks
  - ↪ Atomicity violations
  - ↪ ...

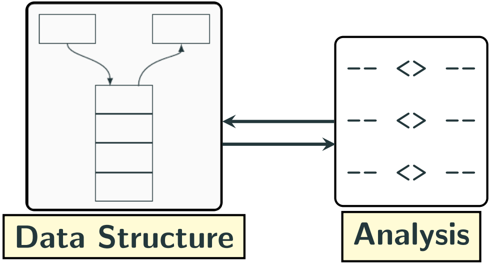


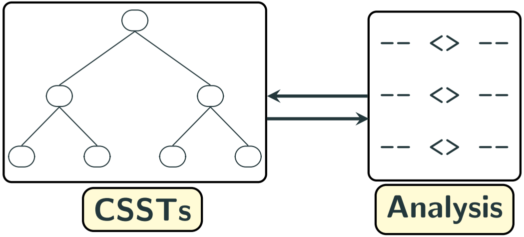
Dynamic Analyses for Detecting Concurrency Bugs



- Popular approach for finding concurrency bugs
- Widely adopted (e.g., ThreadSanitizer, Helgrind)
- **Performance** is crucial

Overview

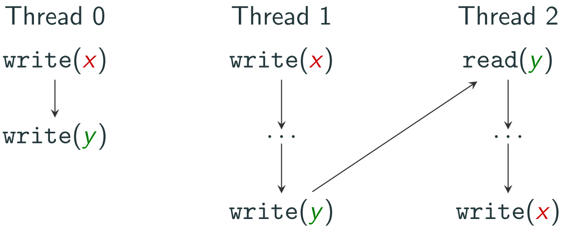




Analysis is **slow**

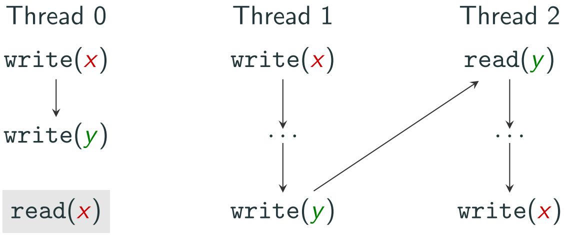
Analysis is **fast**

Partial Orders in Dynamic Analyses

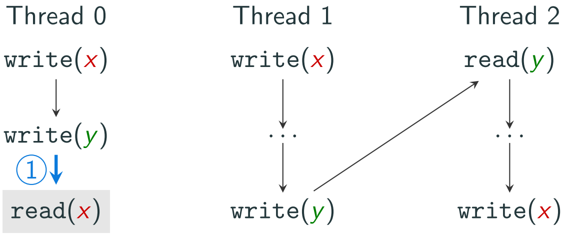


- Analyses require establishing a causal ordering among the events
- Causality is typically represented as a **partial order**

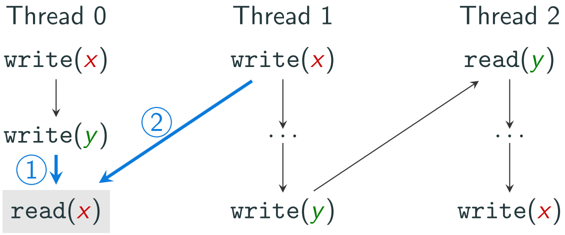
Partial Orders in Dynamic Analyses



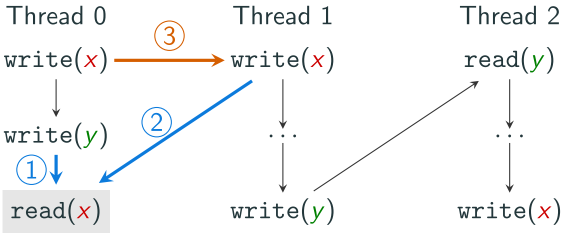
Partial Orders in Dynamic Analyses



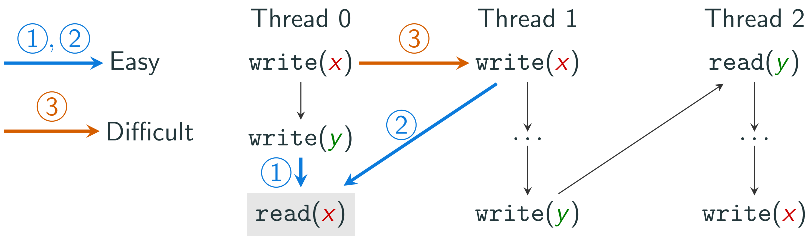
Partial Orders in Dynamic Analyses



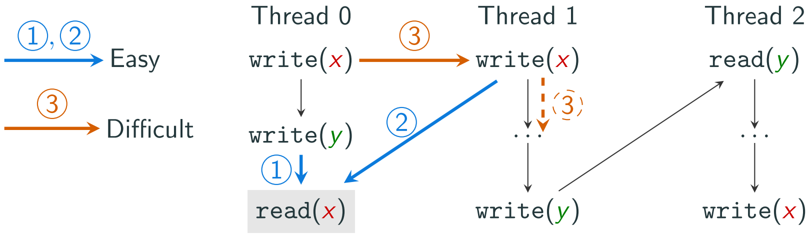
Partial Orders in Dynamic Analyses



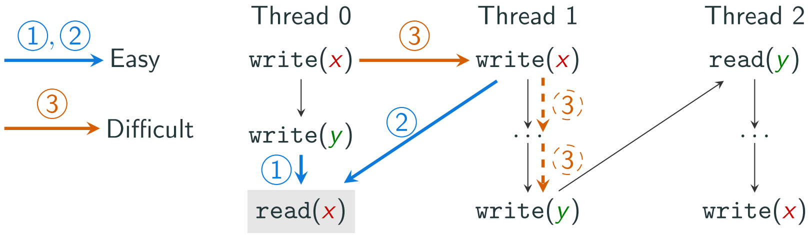
Partial Orders in Dynamic Analyses



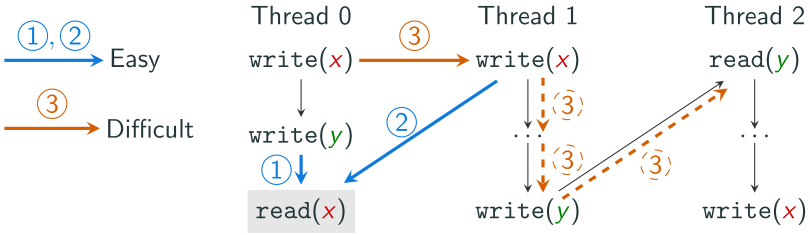
Partial Orders in Dynamic Analyses



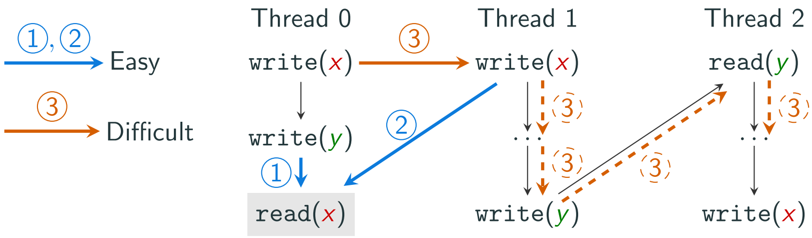
Partial Orders in Dynamic Analyses



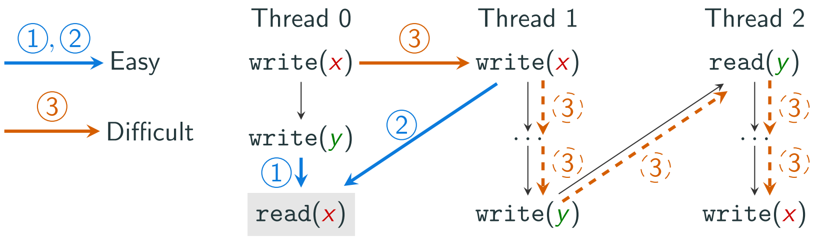
Partial Orders in Dynamic Analyses



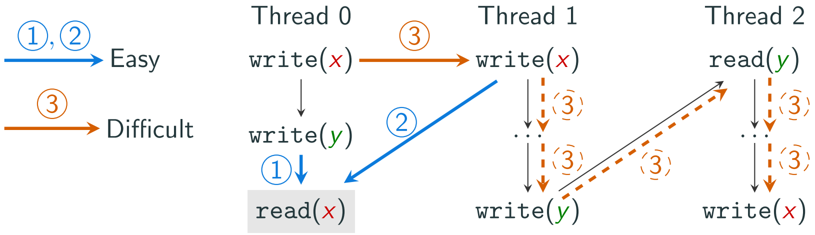
Partial Orders in Dynamic Analyses



Partial Orders in Dynamic Analyses



Partial Orders in Dynamic Analyses



- Partial orders are essential for dynamic analyses

  ↪ Continuously queried and refined

  ↪ Plays a critical role in the overall performance

Maintaining Partial Orders in Dynamic Analysis

	Insert	Query	Delete
Vector Clocks	$O(n)$	$O(1)$	✗

n: number of events

Maintaining Partial Orders in Dynamic Analysis

	Insert	Query	Delete	
Vector Clocks	$O(n)$	$O(1)$	✗	n: number of events

Can we do better?

Maintaining Partial Orders in Dynamic Analysis

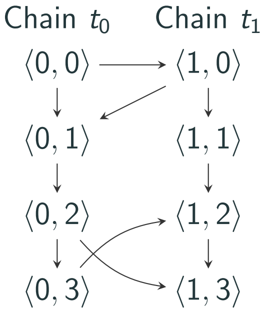
	Insert	Query	Delete
Vector Clocks	$O(n)$	$O(1)$	✗
CSSTs	$O(\log n)$	$O(\log n)$	$O(\log n)$

n: number of events

Dynamic Reachability

- **Input:** (i) Chain DAG

  (ii) Online sequence of operations
  - **Update:**
    - $\hookrightarrow$ $\texttt{insertEdge}(e_1, e_2)$
    - $\hookrightarrow$ $\texttt{deleteEdge}(e_1, e_2)$
  - **Query:**
    - $\hookrightarrow$ $\texttt{reachable}(e_1, e_2)$
    - $\hookrightarrow$ $\texttt{successor}(e_1, t)$
    - $\hookrightarrow$ $\texttt{predecessor}(e_1, t)$



- **Task:** Answer queries correctly

  $\hookrightarrow$ Considering all prior updates

Dynamic Suffix Minima

- Can formulate dynamic reachability
- **Input:** (i) Integer array A

  (ii) Online sequence of operations
  - **Update:**

    ↪ $\texttt{update}(\texttt{A}, i, a)$
  - **Query:**

    ↪ $\texttt{min}(\texttt{A}, i)$

    ↪ $\texttt{argleq}(\texttt{A}, a)$
- **Task:** Answer queries correctly

  ↪ Considering all prior updates

A :

6	9	8	10
0	1	2	3

Dynamic Suffix Minima

- Can formulate dynamic reachability
- **Input:** (i) Integer array A

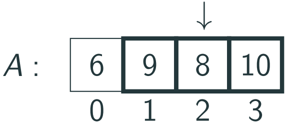
  (ii) Online sequence of operations
  - **Update:**

    $\hookrightarrow$ $\texttt{update}(\texttt{A}, i, a)$
  - **Query:**

    $\hookrightarrow$ $\texttt{min}(\texttt{A}, i)$

    $\hookrightarrow$ $\texttt{argleq}(\texttt{A}, a)$
- **Task:** Answer queries correctly

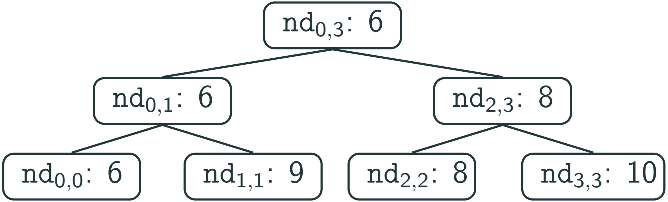
  $\hookrightarrow$ Considering all prior updates



$\texttt{min}(\texttt{A}, 1) = 8$

Segment Trees

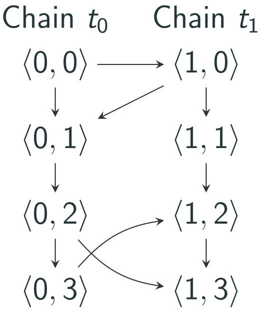
- Classic data structure
- Solves dynamic suffix minima problem efficiently
  - ↪ $O(\log n)$ per query and update



$$A = [6, 9, 8, 10]$$

Dynamic Suffix Minima

- **Formulation of dynamic reachability**
  - ↪ $A_{t_0}^{t_1}$ represents reachability information from t_0 to t_1
  - ↪ The collection $(A_{t_0}^{t_1}, A_{t_1}^{t_0}, \ldots)$ represents global reachability information



$A_{t_0}^{t_1}$:

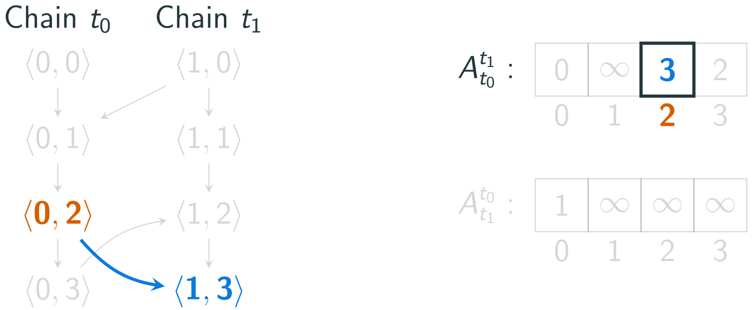
0	1	2	3
0	∞	3	2

$A_{t_1}^{t_0}$:

0	1	2	3
1	∞	∞	∞

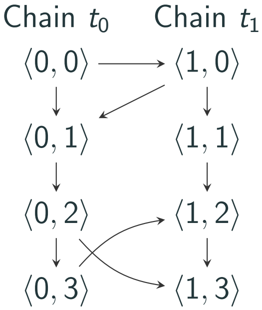
Dynamic Suffix Minima

- **Formulation of dynamic reachability**
  - ↪ $A_{t_0}^{t_1}$ represents reachability information from t_0 to t_1
  - ↪ The collection $(A_{t_0}^{t_1}, A_{t_1}^{t_0}, \ldots)$ represents global reachability information



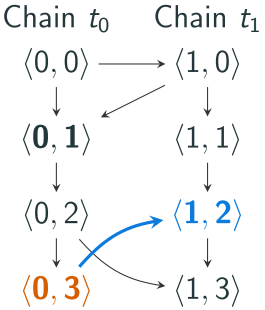
Dynamic Suffix Minima

- **Formulation of dynamic reachability**
  - $\texttt{update}(\texttt{A}, i, a)$ handles
    - $\hookrightarrow$ $\texttt{insertEdge}(e_1, e_2)$
    - $\hookrightarrow$ $\texttt{deleteEdge}(e_1, e_2)$
  - $\texttt{min}(\texttt{A}, i)$ handles
    - $\hookrightarrow$ $\texttt{reachable}(e_1, e_2)$
    - $\hookrightarrow$ $\texttt{successor}(e_1, t)$
  - $\texttt{argleq}(\texttt{A}, a)$ handles
    - $\hookrightarrow$ $\texttt{predecessor}(e_1, t)$



Dynamic Suffix Minima

- **Formulation of dynamic reachability**
  - $\texttt{update}(\texttt{A}, i, a)$ handles
    - ↪ $\texttt{insertEdge}(e_1, e_2)$
    - ↪ $\texttt{deleteEdge}(e_1, e_2)$
  - $\texttt{min}(\texttt{A}, i)$ handles
    - ↪ $\texttt{reachable}(e_1, e_2)$
    - ↪ $\texttt{successor}(e_1, t)$
  - $\texttt{argleq}(\texttt{A}, a)$ handles
    - ↪ $\texttt{predecessor}(e_1, t)$



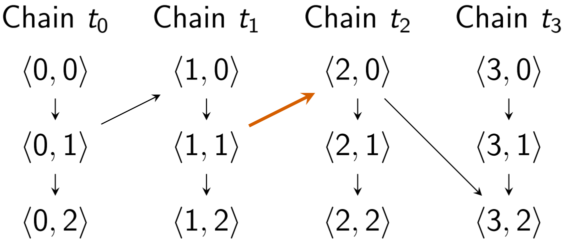
$$\texttt{successor}(\langle 0, 1\rangle, t_1) = \texttt{min}(\texttt{A}_{t_0}^{t_1}, 1)$$

$A_{t_0}^{t_1}$:

0	1	2	3
0	∞	3	2

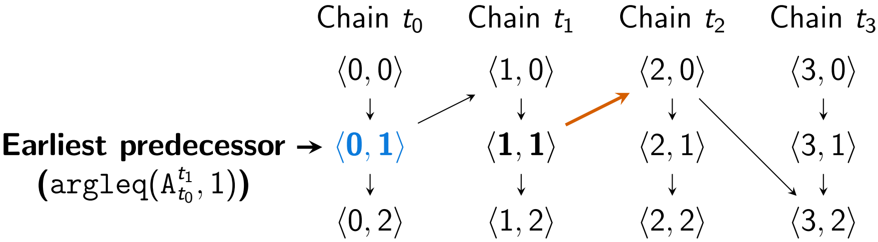
Dynamic Reachability with Segment Trees

- **Query:** $O(\log n)$
- **Insert edge:** $O(k^2 \log n)$
  $\hookrightarrow$ Transitive closure



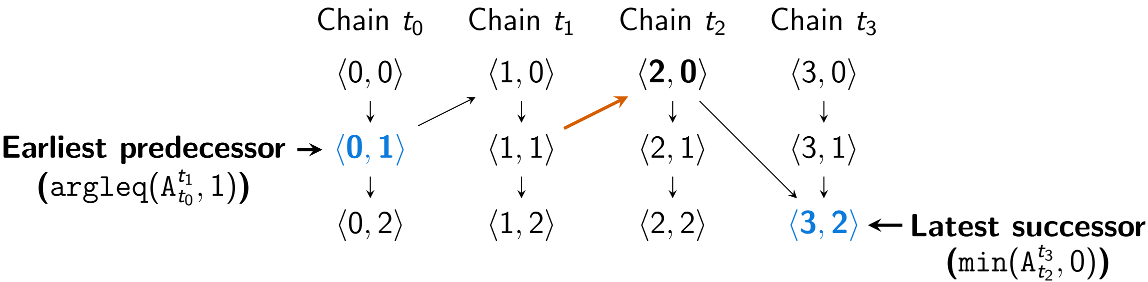
Dynamic Reachability with Segment Trees

- **Query:** $O(\log n)$
- **Insert edge:** $O(k^2 \log n)$
  $\hookrightarrow$ Transitive closure



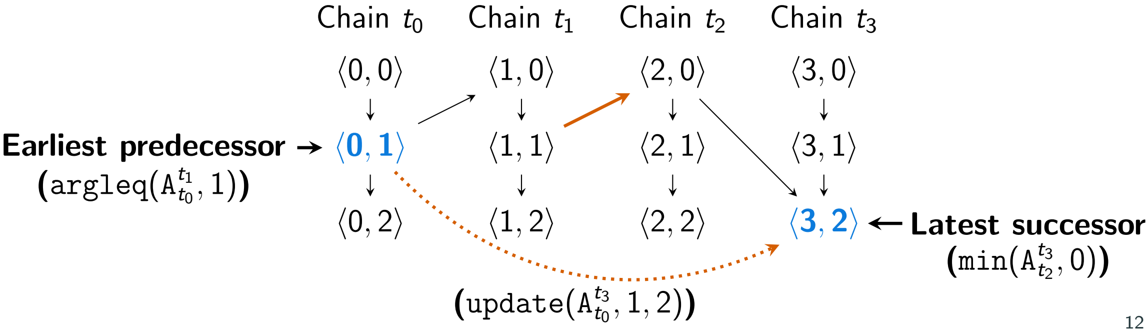
Dynamic Reachability with Segment Trees

- **Query:** $O(\log n)$
- **Insert edge:** $O(k^2 \log n)$
  $\hookrightarrow$ Transitive closure



Dynamic Reachability with Segment Trees

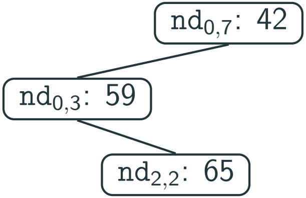
- **Query:** $O(\log n)$
- **Insert edge:** $O(k^2 \log n)$
  $\hookrightarrow$ Transitive closure



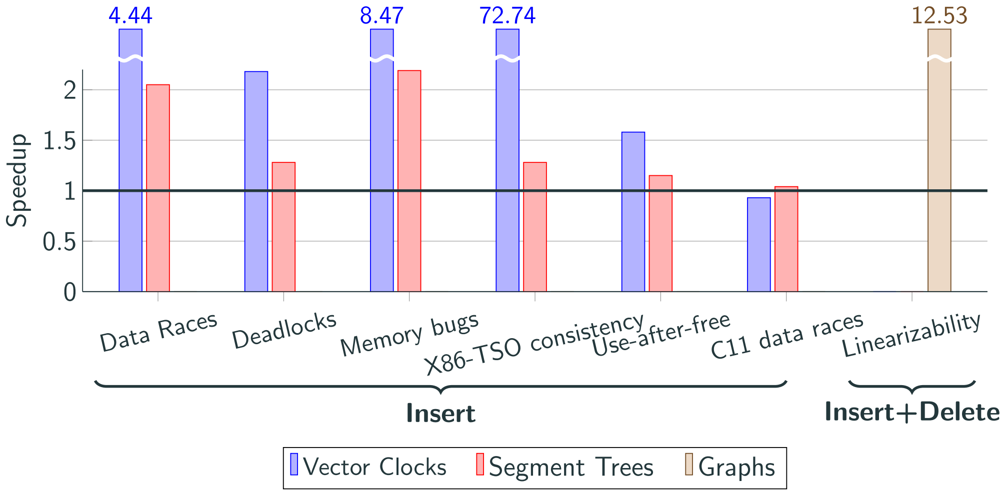
Sparse Segment Trees

- **Key observation:** Arrays $A_t^{t'}$ are typically sparse
- Improved complexity: $O(min(\log n, d))$
  - $\hookrightarrow$ By exploiting sparsity
  - $\hookrightarrow$ d is the maximum number of nodes in a chain that have an outgoing edge
- Sparsity is maintained in transitive closure

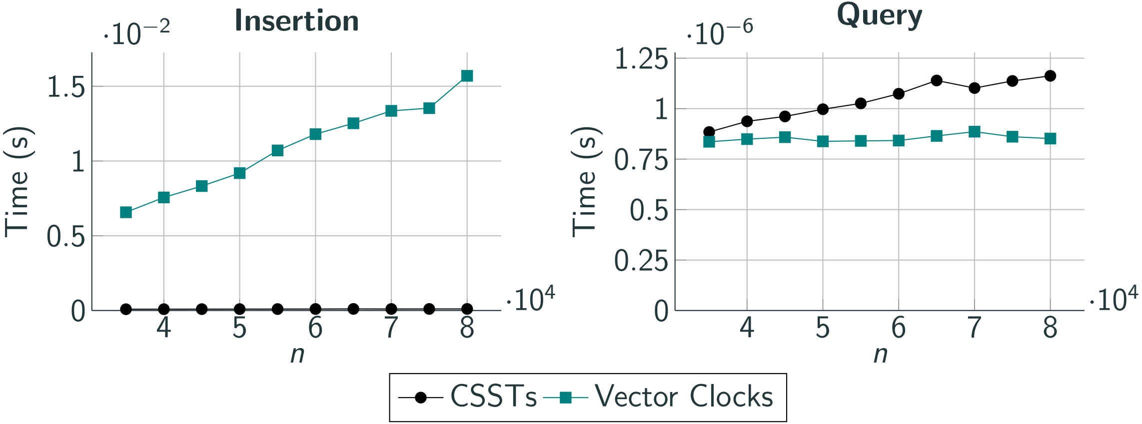
59	∞	65	∞	∞	∞	∞	42
0	1	2	3	4	5	6	7



Experimental Results



Scalability Experiments



CSSTs (Collective Sparse Segment Trees)

Conclusion

CSSTs (Collective Sparse Segment Trees)

- Drop-in replacement of existing data structures

Conclusion

CSSTs (Collective Sparse Segment Trees)

- Drop-in replacement of existing data structures
- Offers complexity improvements in complex dynamic analyses

Conclusion

CSSTs (Collective Sparse Segment Trees)

- Drop-in replacement of existing data structures
- Offers complexity improvements in complex dynamic analyses
- Experimental results confirm the practical impact

Conclusion

CSSTs (Collective Sparse Segment Trees)

- Drop-in replacement of existing data structures
- Offers complexity improvements in complex dynamic analyses
- Experimental results confirm the practical impact

Thank you!

GitHub: @hcantunc/cssts

Appendix

Streaming vs. Non-Streaming Dynamic Analyses

Streaming Analyses

- No propagation
- Vector clocks are **efficient**

  ↪ Maintain the partial order in $O(kn)$

Non-Streaming Analyses

- Requires propagation
- Vector clocks are **inefficient**

  ↪ Maintain the partial order in $O(kn^2)$

n : number of events

↪ Very large in practice

k : number of threads