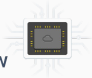
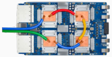
A Tree Clock Data Structure for Causal Orderings in Concurrent Executions

Umang Mathur
Andreas Pavlogiannis
Hünkar Can Tunç
Mahesh Viswanathan

National University of Singapore
Aarhus University
University of Illinois Urbana-Champaign

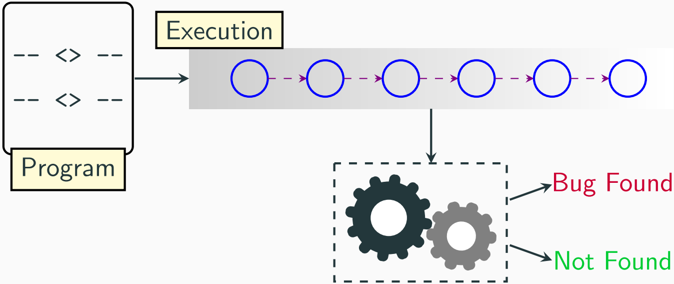
Concurrency: Software and Challenges





- Ubiquitous computing paradigm.
- Analysis of concurrent programs is a major challenge.
- We need more **efficient** algorithms and data structures.

Dynamic Analyses for Detecting Concurrency Bugs



- Widely adopted (e.g., ThreadSanitizer, Helgrind).
- Requires establishing a causal ordering between the events.
- Causality is typically represented as a partial order.

Happens-Before (HB) Partial Order

	t_1	t_2
1	acquire(ℓ_1)	
2	*write*(x)	
3	release(ℓ_1)	
4		*read*(x)
5		acquire(ℓ_1)
6		*read*(x)
7		release(ℓ_1)

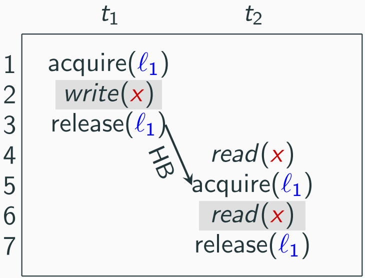
Happens-Before (HB) Partial Order

	t_1	t_2
1	acquire(ℓ_1)	
2	*write*(x)	
3	release(ℓ_1)	
4		*read*(x)
5		acquire(ℓ_1)
6		*read*(x)
7		release(ℓ_1)

HB

$\rightarrow$ Events e_2 and e_4 are concurrent.

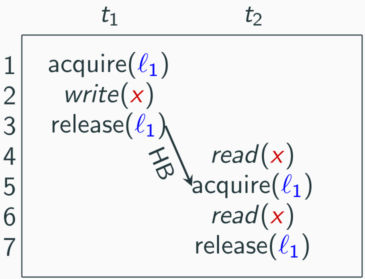
Happens-Before (HB) Partial Order



→ Events e_2 and e_4 are concurrent.

→ Events e_2 and e_6 are **not** concurrent.

Happens-Before (HB) Partial Order



→ Events e_2 and e_4 are concurrent.

→ Events e_2 and e_6 are **not** concurrent.

Happens-Before defines data races in various memory models.

Contributions

Tree Clocks: A new data structure

- Can be used to compute Happens-Before efficiently.
  - Optimal data structure for Happens-Before.

Contributions

Tree Clocks: A new data structure

- Can be used to compute Happens-Before efficiently.
  - Optimal data structure for Happens-Before.
- Versatile data structure.
  - Other partial orders can also be computed efficiently.
  - Schedulable-Happens-Before
  - Mazurkiewicz

Contributions

Tree Clocks: A new data structure

- Can be used to compute Happens-Before efficiently.
  - Optimal data structure for Happens-Before.
- Versatile data structure.
  - Other partial orders can also be computed efficiently.
  - Schedulable-Happens-Before
  - Mazurkiewicz
- Significant speedups compared to vector clocks.

Background: Vector Timestamps

- The knowledge set of a thread t can be succinctly captured by a function:

$$\mathsf{V}_t : \text{Threads} \to \mathbb{N}$$

- $\mathsf{V}_t(t')$ gives the last event of t' that t knows about.
- t knows about all preceding events as well.

Background: Vector Timestamps

- The knowledge set of a thread t can be succinctly captured by a function:

$$\mathsf{V}_t : \text{Threads} \to \mathbb{N}$$

- $\mathsf{V}_t(t')$ gives the last event of t' that t knows about.
- t knows about all preceding events as well.

$$\mathsf{V}_{t_2} = [\overset{t_1}{27}, \overset{t_2}{3}, \overset{t_3}{9}, \overset{t_4}{45}, \overset{t_5}{17}, \overset{t_6}{26}]$$

- t_2 knows of the first 27 events of t_1.
- t_2 has performed 3 events.

Background: Vector Timestamps

- The knowledge set of a thread t can be succinctly captured by a function:

$$\mathsf{V}_t \colon \text{Threads} \to \mathbb{N}$$

- $\mathsf{V}_t(t')$ gives the last event of t' that t knows about.
- t knows about all preceding events as well.

$$\mathsf{V}_{t_2} = [\overset{t_1}{27}, \overset{t_2}{3}, \overset{t_3}{9}, \overset{t_4}{45}, \overset{t_5}{17}, \overset{t_6}{26}]$$

- t_2 knows of the first 27 events of t_1.
- t_2 has performed 3 events.

Operations

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

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

Background: Implementing Vector Timestamps

Just use a vector clock $\mathsf{VC}_t = [27, 3, 9, 45, 17, 26]$

Vector Clock Join $\mathsf{VC}_1 \leftarrow \mathsf{VC}_1 \sqcup \mathsf{VC}_2$

- For each thread t:
  - If $\mathsf{VC}_1[t] < \mathsf{VC}_2[t]$
    - $\mathsf{VC}_1[t] \leftarrow \mathsf{VC}_2[t]$

Vector Clock Copy $\mathsf{VC}_1 \leftarrow \mathsf{VC}_2$

- For each thread t:
  - $\mathsf{VC}_1[t] \leftarrow \mathsf{VC}_2[t]$

Each operation takes $O(\mathcal{T})$ time, for $\mathcal{T}$ threads

Background: Computing Happens-Before with Vector Clocks

- One vector clock $\mathbb{C}_t$ per thread t
- One vector clock $\mathbb{C}_\ell$ per lock ℓ

Algorithm: Happens-Before (HB)

```
procedure acquire(t, ℓ)
    C_t ← C_t ⊔ C_ℓ;      /* Vector clock join */
procedure release(t, ℓ)
    C_ℓ = C_t;            /* Vector clock copy */
```

Background: Computing Happens-Before with Vector Clocks

- One vector clock $\mathbb{C}_t$ per thread t
- One vector clock $\mathbb{C}_\ell$ per lock ℓ

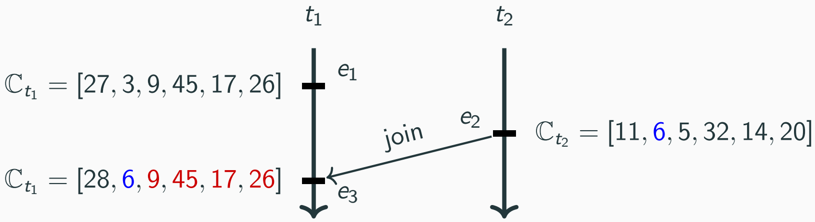
Algorithm: Happens-Before (HB)

procedure `acquire`(t, ℓ)
    $\mathbb{C}_t \leftarrow \mathbb{C}_t \sqcup \mathbb{C}_\ell$; `/* Vector clock join */`
procedure `release`(t, ℓ)
    $\mathbb{C}_\ell = \mathbb{C}_t$; `/* Vector clock copy */`

- Every vector clock operation costs $O(\mathcal{T})$
  - $\mathcal{T}$ is the number of threads
- When threads are many, the complexity is quadratic $O(\mathcal{N} \cdot \mathcal{T})$
  - $\mathcal{N}$ is the number of acquire/release events

Overhead of Vector Clocks

- Every vector clock join takes $O(\mathcal{T})$ time.
- Certain steps in the join operation can be vacuous.



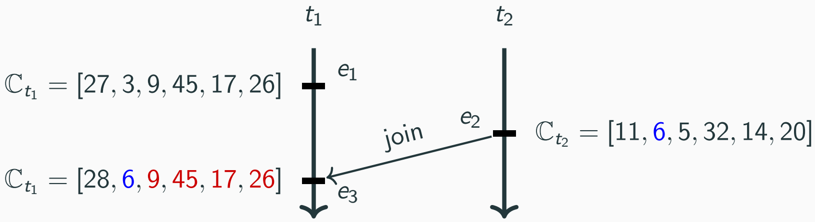
Overhead of Vector Clocks

- Every vector clock join takes $O(\mathcal{T})$ time.
- Certain steps in the join operation can be vacuous.

Can we do sub-linear joins?

→ Sub-linear means skip looking at certain entries. How?

→ Tree clocks address this challenge.



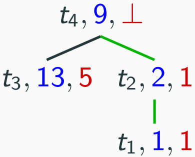
Our Contribution: Tree Clock Data Structure

- **Drop-in** replacement of vector clocks.
- Tree clocks maintain information hierarchically.
  - Nodes store local times of a thread + metadata.
  - Tree structure records how information has been obtained transitively.

Our Contribution: Tree Clock Data Structure

- **Drop-in** replacement of vector clocks.
- Tree clocks maintain information hierarchically.
  - Nodes store local times of a thread + metadata.
  - Tree structure records how information has been obtained transitively.

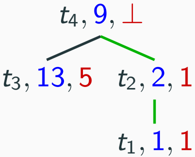
$$\mathbb{C}_{t_4} = [\overset{t_1}{1}, \overset{t_2}{2}, \overset{t_3}{13}, \overset{t_4}{9}] \longrightarrow$$



Our Contribution: Tree Clock Data Structure

- **Drop-in** replacement of vector clocks.
- Tree clocks maintain information hierarchically.
  - Nodes store local times of a thread + metadata.
  - Tree structure records how information has been obtained transitively.

$$\mathbb{C}_{t_4} = [\overset{t_1}{1}, \overset{t_2}{2}, \overset{t_3}{13}, \overset{t_4}{9}] \longrightarrow$$

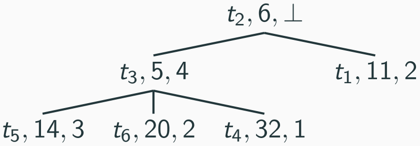


- Only slightly more information is stored compared to vector clocks.

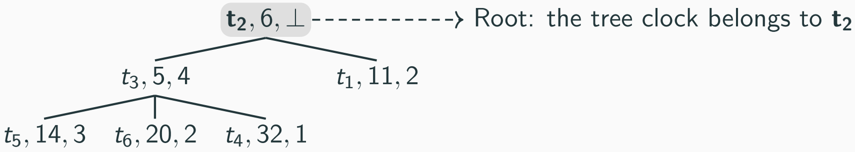
↓

Enough to enable tree clocks to support sub-linear join and (monotone) copy!

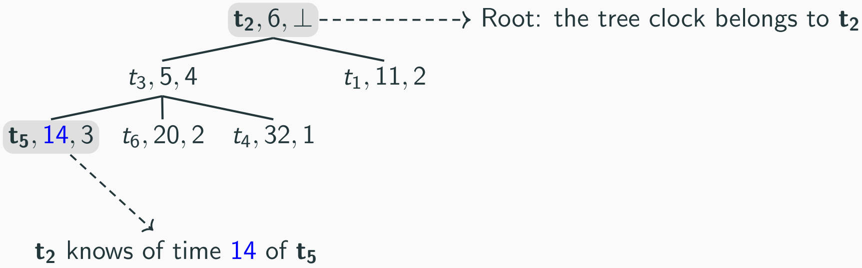
Tree Clock Data Structure



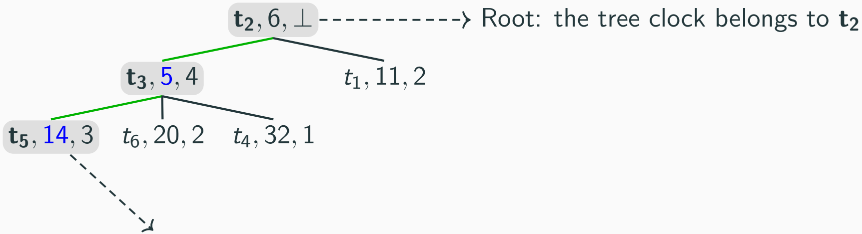
Tree Clock Data Structure



Tree Clock Data Structure



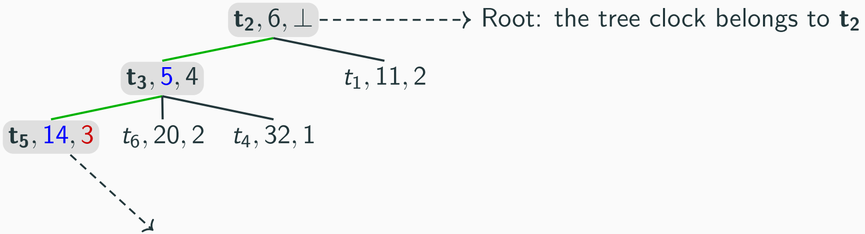
Tree Clock Data Structure



$\mathbf{t_2}$ knows of time 14 of $\mathbf{t_5}$

$\hookrightarrow$ It learned this transitively, by learning of time 5 of $\mathbf{t_3}$

Tree Clock Data Structure

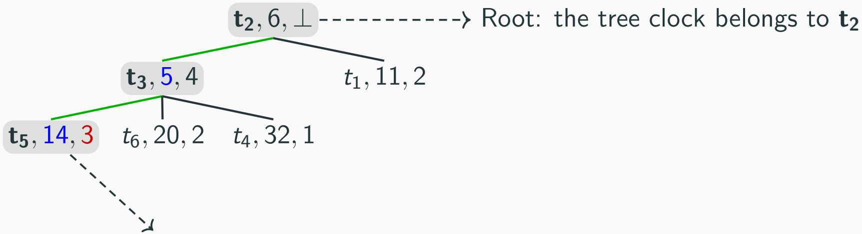


$\mathbf{t_2}$ knows of time 14 of $\mathbf{t_5}$

$\hookrightarrow$ It learned this transitively, by learning of time 5 of $\mathbf{t_3}$

$\hookrightarrow$ $\mathbf{t_3}$ learned of time 14 of $\mathbf{t_5}$ when $\mathbf{t_3}$'s time was 3

Tree Clock Data Structure



$\mathbf{t_2}$ knows of time 14 of $\mathbf{t_5}$

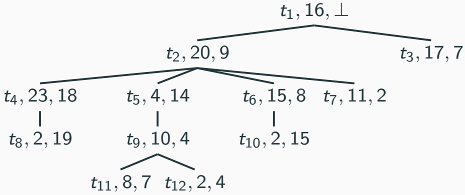
$\hookrightarrow$ It learned this transitively, by learning of time 5 of $\mathbf{t_3}$

$\hookrightarrow$ $\mathbf{t_3}$ learned of time 14 of $\mathbf{t_5}$ when $\mathbf{t_3}$'s time was 3

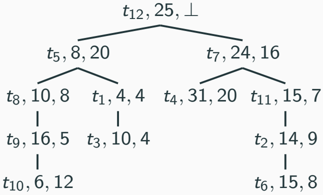
This structure allows for sub-linear time join and (monotone) copy.

TC_{12} .`Join`(TC_1)

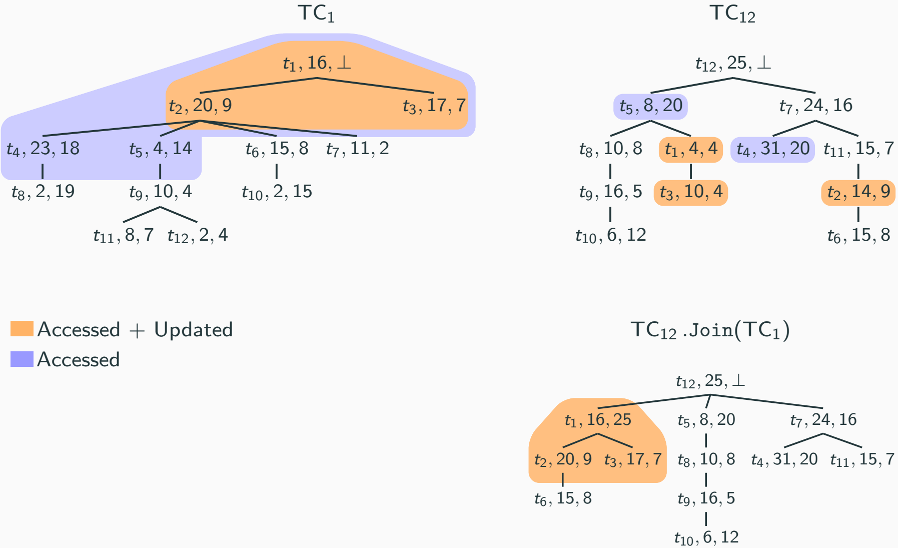
TC_1



TC_{12}

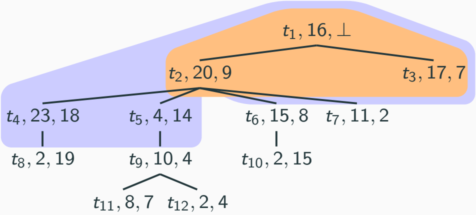


TC_{12}.Join(TC_1)

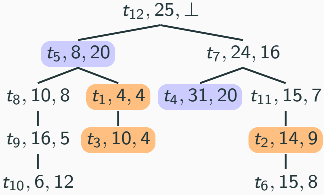


TC_{12} .Join(TC_1)

TC_1



TC_{12}

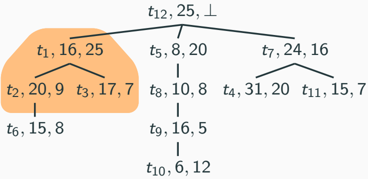


Accessed + Updated

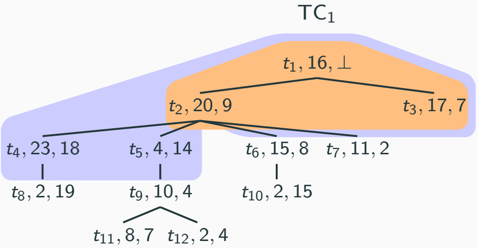
Accessed

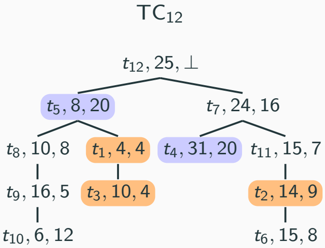
Performed join without accessing the whole tree!

TC_{12} .Join(TC_1)



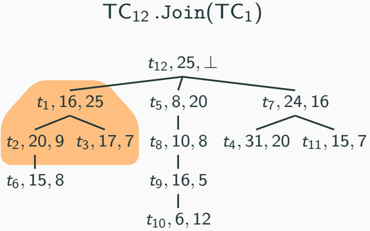
TC_{12}.Join(TC_1)





Accessed + Updated

Accessed



Performed join without accessing the whole tree!

$\mathbb{C}_{t_1} = [16, 20, 17, 23, 4, 15, 11, 8, 10, 2, 8, 7]$

t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
16	20	17	23	4	15	11	8	10	2	8	7

Drop-in Replacement

Algorithm: Happens-Before with **Vector Clocks**.

procedure `acquire`(t, ℓ)
 $\mathbb{C}_t$.`VectorClockJoin`($\mathbb{C}_\ell$)

procedure `release`(t, ℓ)
 $\mathbb{C}_\ell$.`VectorClockCopy`($\mathbb{C}_t$)

Algorithm: Happens-Before with **Tree Clocks**.

procedure `acquire`(t, ℓ)
 $\mathbb{C}_t$.`TreeClockJoin`($\mathbb{C}_\ell$)

procedure `release`(t, ℓ)
 $\mathbb{C}_\ell$.`TreeClockCopy`($\mathbb{C}_t$)

Tree Clock Optimality

What is the optimal data structure for Happens-Before?

Tree Clock Optimality

What is the optimal data structure for Happens-Before?

Treeclock optimality for Happens-Before

No other data structure can offer asymptotically better performance.

Tree Clock Optimality

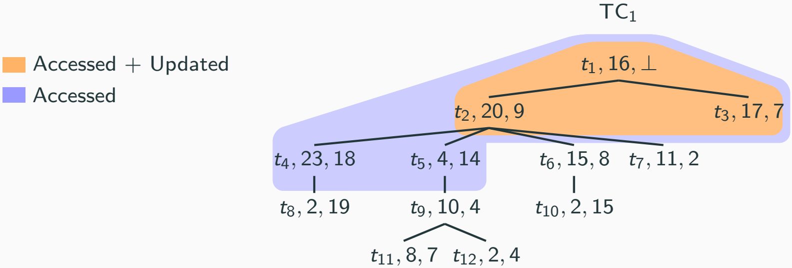
What is the optimal data structure for Happens-Before?

Treeclock optimality for Happens-Before

No other data structure can offer asymptotically better performance.

- Tree clocks perform at most 3 times more work than necessary.

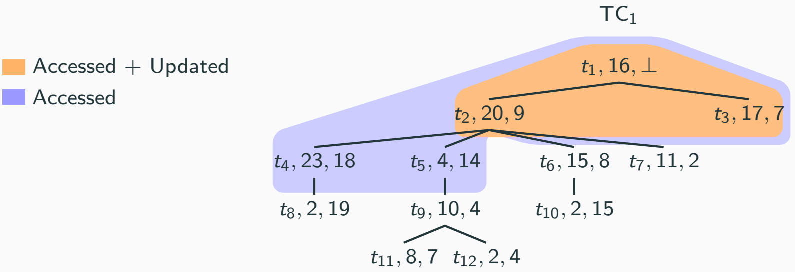
Tree Clock Optimality



Vector time work VTWork(σ)

VTWork(σ) = the *smallest* number of data-structure accesses for processing σ

Tree Clock Optimality



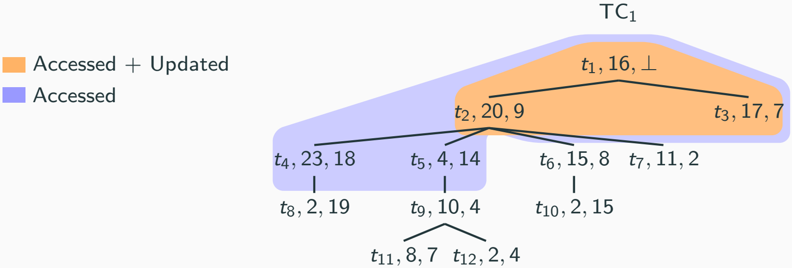
Vector time work VTWork(σ)

VTWork(σ) = the *smallest* number of data-structure accesses for processing σ

Tree clock work TCWork(σ)

TCWork(σ) = the total number of *tree* clock entries accessed for processing σ

Tree Clock Optimality



Vector time work VTWork(σ)

VTWork(σ) = the *smallest* number of data-structure accesses for processing σ

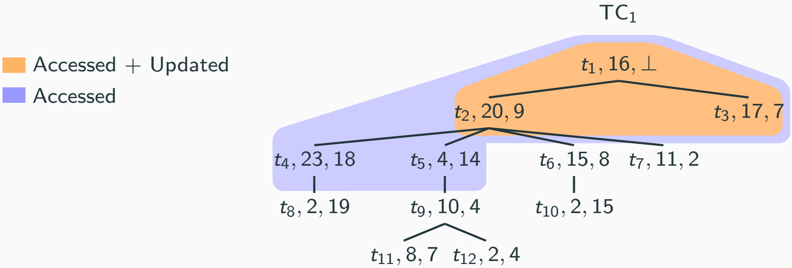
Tree clock work TCWork(σ)

TCWork(σ) = the total number of *tree* clock entries accessed for processing σ

Vector clock work VCWork(σ)

VCWork(σ) = the total number of *vector* clock entries accessed for processing σ

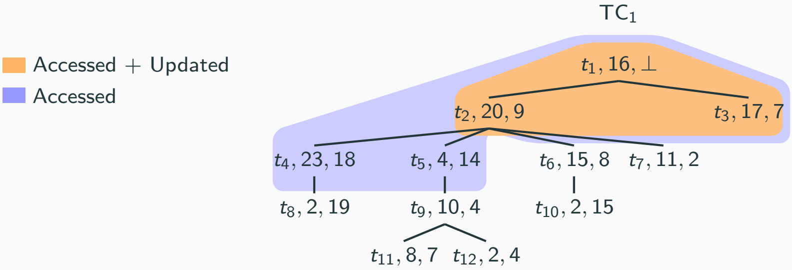
Data Structure Optimality for Happens-Before



$\mathsf{VCWork}(\sigma) \leq \mathcal{T} \cdot \mathsf{VTWork}(\sigma)$

VCWork(σ) can be $\mathcal{T}$ times worse

Data Structure Optimality for Happens-Before



$\text{VCWork}(\sigma) \leq \mathcal{T} \cdot \text{VTWork}(\sigma)$

VCWork(σ) can be $\mathcal{T}$ times worse

Theorem

$\text{TCWork}(\sigma) \leq 3 \cdot \text{VTWork}(\sigma)$

Tree clocks are (asymptotically) VT-optimal!

Beyond Happens-Before

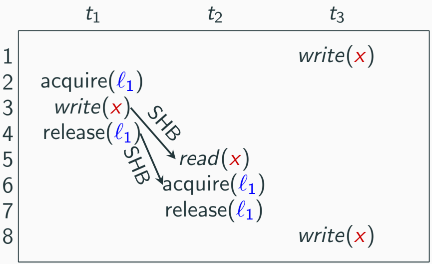
- Tree clocks can be used to compute other partial orders.

Beyond Happens-Before

- Tree clocks can be used to compute other partial orders.

Schedulable-Happens-Before (SHB)

- Used in sound data race detection[1]



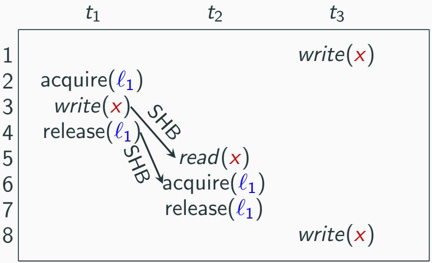
[1] U. Mathur, D. Kini, M. Viswanathan. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. OOPSLA'18.

Beyond Happens-Before

- Tree clocks can be used to compute other partial orders.

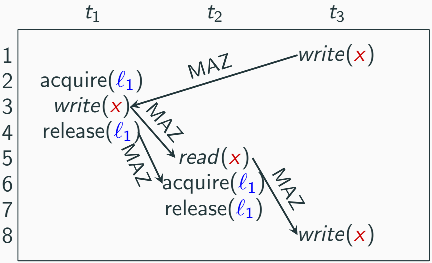
Schedulable-Happens-Before (SHB)

- Used in sound data race detection[1]



Mazurkiewicz (MAZ)

- Used in dynamic partial order reduction in model checking of concurrent programs[2].



[1] U. Mathur, D. Kini, M. Viswanathan. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. OOPSLA'18.

[2] C. Flanagan, P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. POPL'05.

Experimental Results

- 153 benchmark traces
  - Based on standard Java and OpenMP benchmark suites.
- Implemented HB, SHB and MAZ with both tree clocks and vector clocks.
- Measured the time in the following tasks:
  - Compute the partial order.
  - Perform the race-detection analysis.

Experimental Results

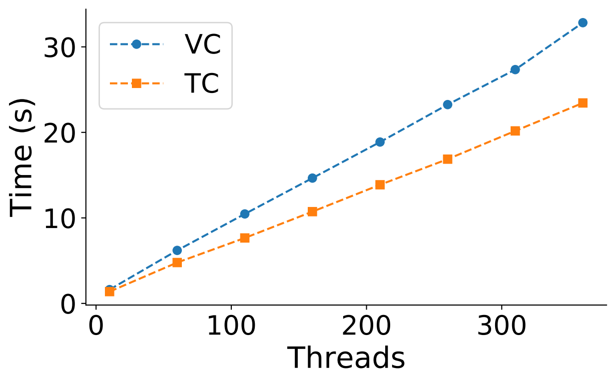
- 153 benchmark traces
  - Based on standard Java and OpenMP benchmark suites.
- Implemented HB, SHB and MAZ with both tree clocks and vector clocks.
- Measured the time in the following tasks:
  - Compute the partial order.
  - Perform the race-detection analysis.

	Mazurkiewicz	*Schedulable-Happens-Before*	*Happens-Before*
Partial Order	2.02×	2.66×	2.97×
Partial Order + Analysis	1.49×	1.80×	1.11×

Significant speedup by just replacing vector clocks with tree clocks!

Experimental Results - Scalability

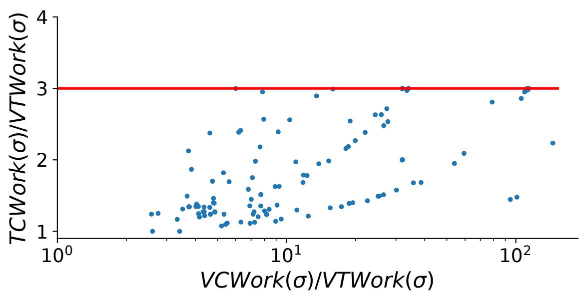
- Controlled experiment: threads randomly communicate over a single lock.
  - Theoretical speedup: 4×
  - Observed speedup: 1.33×



Experimental Results - Data Structure Optimality for HB

Theorem

$\mathsf{TCWork}(\sigma) \leq 3 \cdot \mathsf{VTWork}(\sigma)$



Conclusion

Tree clocks:

- Drop-in replacement of vector clocks.

Conclusion

Tree clocks:

- Drop-in replacement of vector clocks.
- Tree clocks support join and (monotone) copy in sub-linear time.

Conclusion

Tree clocks:

- Drop-in replacement of vector clocks.
- Tree clocks support join and (monotone) copy in sub-linear time.
- Optimal data structure for Happens-Before (asymptotically).

Conclusion

Tree clocks:

- Drop-in replacement of vector clocks.
- Tree clocks support join and (monotone) copy in sub-linear time.
- Optimal data structure for Happens-Before (asymptotically).
- Experimental results confirm the potential of tree clocks.

Conclusion

Tree clocks:

- Drop-in replacement of vector clocks.
- Tree clocks support join and (monotone) copy in sub-linear time.
- Optimal data structure for Happens-Before (asymptotically).
- Experimental results confirm the potential of tree clocks.

Thank you!