Towards Fast, Sound and Effective Predictive Analyses

https://arxiv.org/abs/1901.08857

Andreas Pavlogiannis
Supported by INRIA-EPFL Fellowship

Hosts: Viktor Kunčak (EPFL) and Stephan Merz (INRIA Nancy)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Concurrency Bugs in the Wild

Therac-25

- Radiation Therapy Machine
- Concurrency bug lead to radiation overdose (100x)
- 6 accidents, 3 deaths
- Race condition

Concurrency Bugs in the Wild

Northeast Blackout

- Power outage in Northeastern and Midwestern US, also Canada
- US DoE, estimated cost: $6B
- Contributed to ~100 deaths
- Race condition

Concurrency Bugs

Thread 1: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Concurrency Bugs

Thread 1: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Thread 2: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Concurrency Bugs

Thread 1: Withdraw(x)
if balance $\geq x$ **then**
balance $\leftarrow$ balance $- x$

Thread 2: Withdraw(x)
if balance $\geq x$ **then**
balance $\leftarrow$ balance $- x$

Withdraw(5)

Withdraw(5)

balance $= 8$

Concurrency Bugs

Thread 1: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Thread 2: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Withdraw(5)

Withdraw(5)

balance = 8

Concurrency Bugs

Thread 1: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Thread 2: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Withdraw(5)

Withdraw(5)

$\text{balance} = 8$

Concurrency Bugs

Thread 1: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Thread 2: Withdraw(x)

```
if balance ≥ x then
    balance ← balance − x
```

Withdraw(5)

Withdraw(5)

$$\text{balance} = 8 \rightarrow 3$$

Concurrency Bugs

Thread 1: Withdraw(x)

if balance $\geq x$ **then**
    balance $\leftarrow$ balance $- x$

Thread 2: Withdraw(x)

if balance $\geq x$ **then**
    balance $\leftarrow$ balance $- x$

Withdraw(5)

Withdraw(5)

balance $= 8 \rightarrow 3 \rightarrow -2$

Concurrency Bugs

Thread 1: Withdraw(x)	**Thread 2:** Withdraw(x)

if balance $\geq x$ **then**
    balance $\leftarrow$ balance $- x$
	if balance $\geq x$ **then**
	    balance $\leftarrow$ balance $- x$

Withdraw(5)	Withdraw(5)

$$\text{balance} = 8 \to 3 \to -2$$

Testing Concurrent Programs is Particularly Difficult

To find a bug we need to solve two problems:

1) Find the right **program inputs**

2) Find the right **schedule**

- there are exponentially many schedules

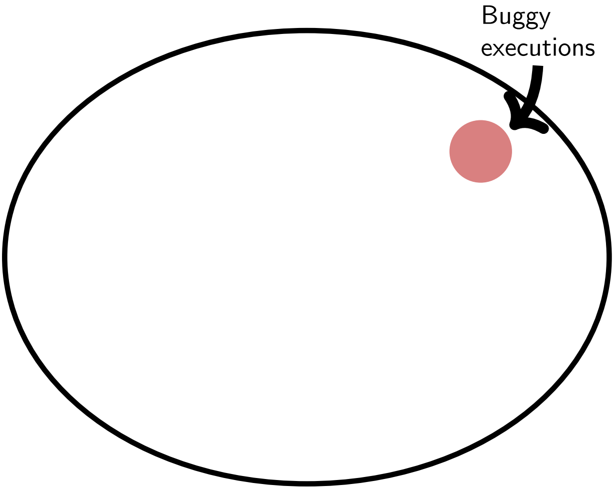
Even if we solve 1), problem 2) remains

Goal

For given inputs, find the schedule exhibiting a bug

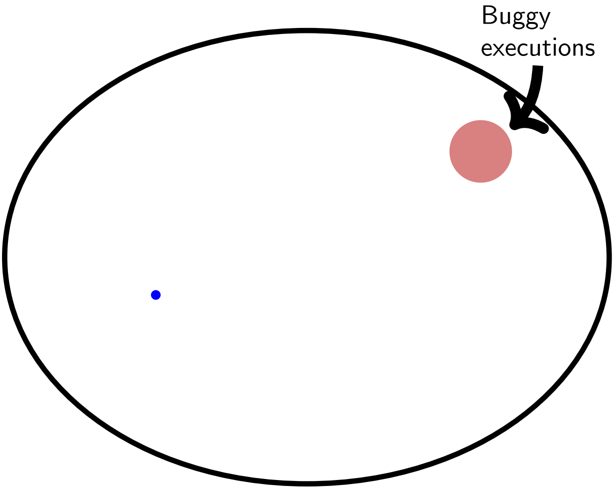
Outcome of Running a Concurrent Program (even for fixed inputs)

"Shooting Darts in the Dark"



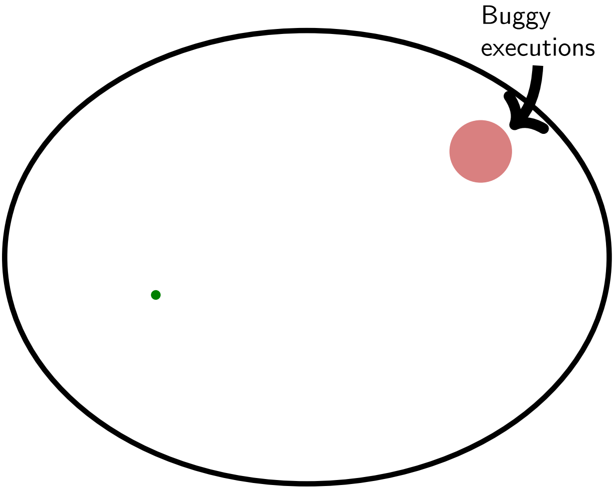
Outcome of Running a Concurrent Program (even for fixed inputs)

"Shooting Darts in the Dark"



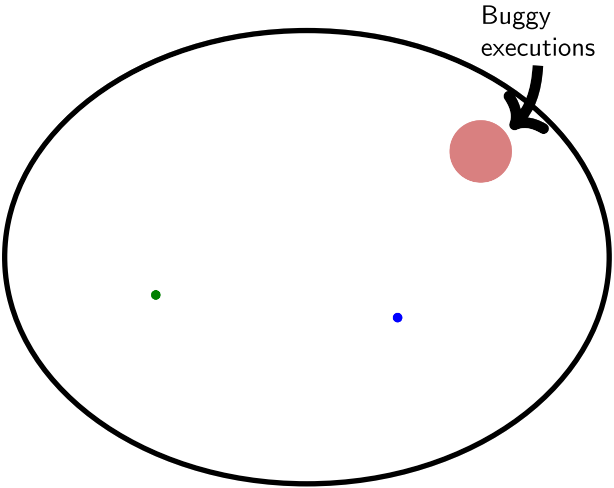
Outcome of Running a Concurrent Program (even for fixed inputs)

"Shooting Darts in the Dark"



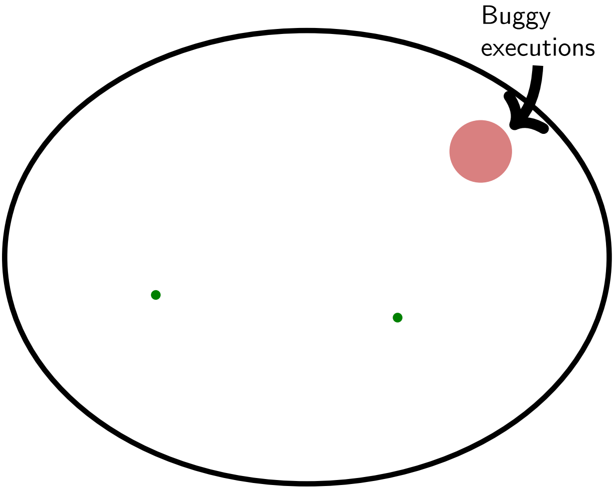
Outcome of Running a Concurrent Program (even for fixed inputs)

"Shooting Darts in the Dark"



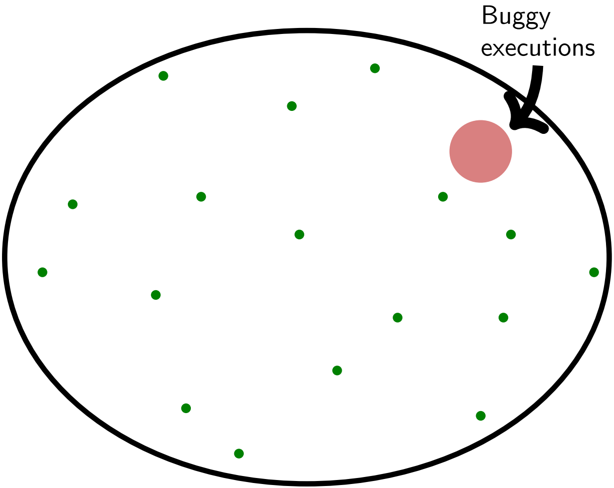
Outcome of Running a Concurrent Program (even for fixed inputs)

"Shooting Darts in the Dark"

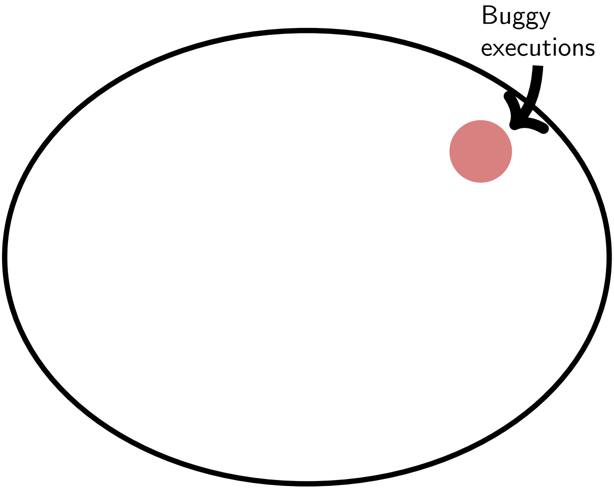


Outcome of Running a Concurrent Program (even for fixed inputs)

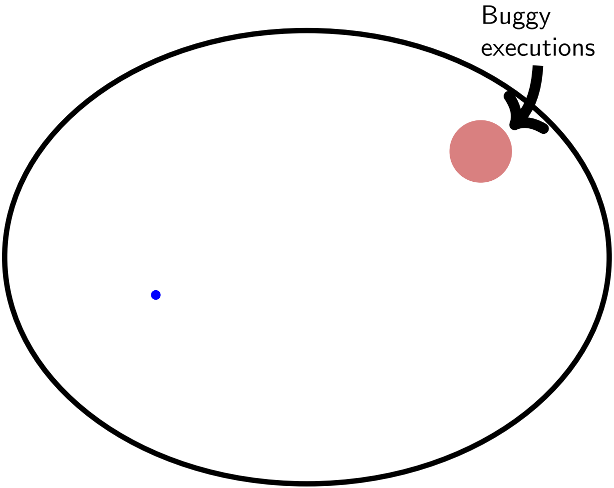
"Shooting Darts in the Dark"



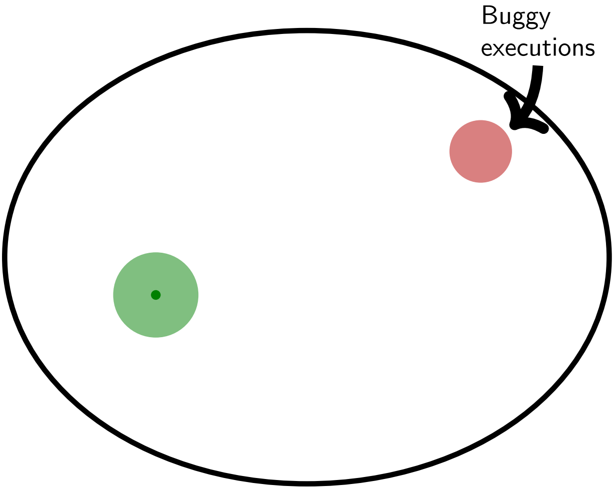
Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



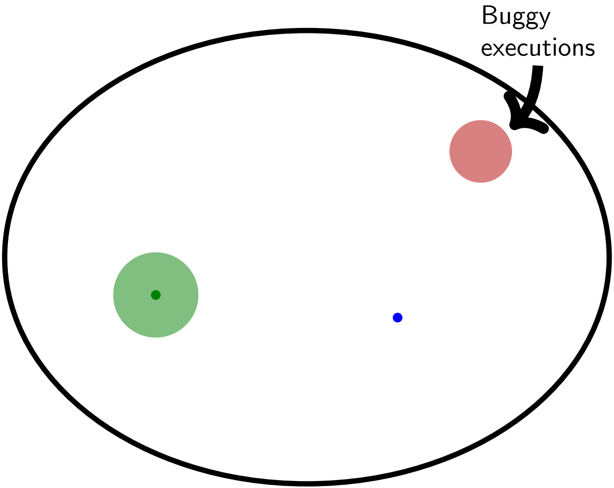
Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



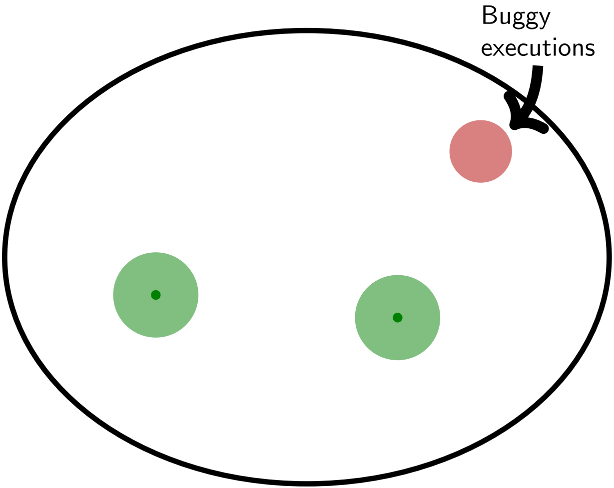
Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



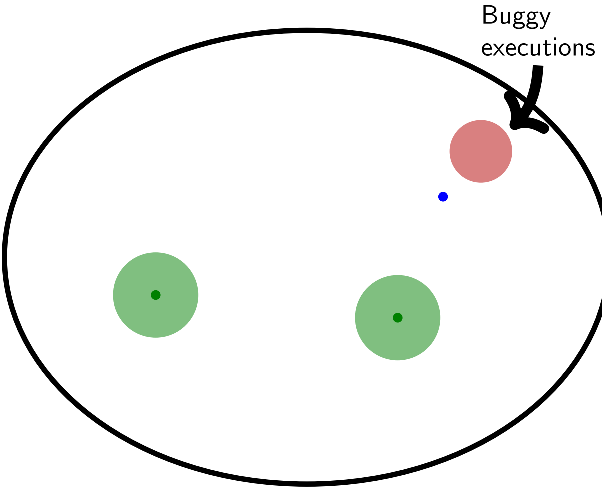
Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



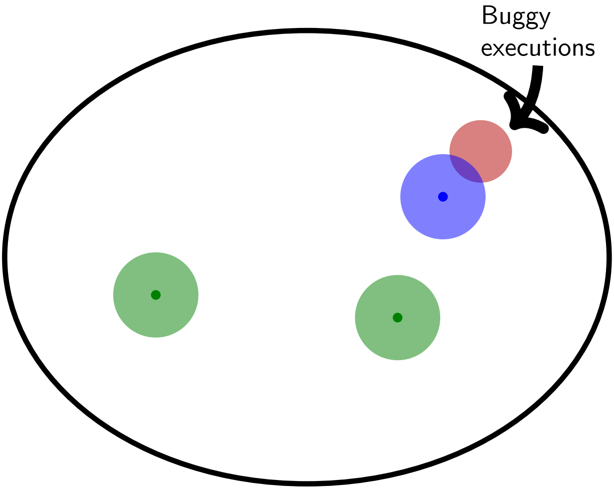
Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



Predictive Techniques: Darts Get Larger (Cover Exponentially Many Executions)



Target Bugs: Data Races

Thread 1: Withdraw(x)

if balance $\geq x$ **then**
    balance $\leftarrow$ balance $- x$

Thread 2: Withdraw(x)

if balance $\geq x$ **then**
    balance $\leftarrow$ balance $- x$

Concurrent access to a shared resource
- At least one modifies it

} **Data Race**

Data races are typically undesirable:

- Even if reads and writes were always atomic, the value seen by read differs depending on whether the write comes before or after
- Non-deterministic result (might read half-written long value)
- Non-portable: may expose, e.g., cache coherence protocols
- Undefined behavior in many memory models: bad by definition (if program has races, you cannot prove it has even trivial properties)

Approaches to Data Races

Static analysis (e.g. type system disciplines):

- good: work for all executions of a program
- bad: spurious warnings, reject certain algorithms and data structures

Our focus: trace known, find bugs for related schedules. Prior work:

- Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas E. Anderson: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. Comput. Syst. 1997.
- Tayfun Elmas, Shaz Qadeer, Serdar Tasiran: Goldilocks: a race and transaction-aware java runtime. PLDI 2007
- Cormac Flanagan, Stephen N. Freund: FastTrack: efficient and precise dynamic race detection. PLDI 2009
- Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, Cormac Flanagan: Sound predictive race detection in polynomial time. POPL 2012
- Dileep Kini, Umang Mathur, Mahesh Viswanathan: Dynamic race prediction in linear time. PLDI 2017
- Jake Roemer, Kaan Gen, Michael D. Bond: High-coverage, unbounded sound predictive race detection. PLDI 2018

Existing Techniques

Simplify the underlying algorithmic issue in different ways:

- precise reordering, but exponential search
  (in practice: apply it in a window, which misses long-distance races)
- weaker reordering: reports non-existing races, not so helpful
- use **under-approximation**: fix certain orders just because they are ordered in the input trace to make it easier to ensure the trace is feasible
  Misses opportunity to report certain races:

$\mathcal{T}_1$	$\mathcal{T}_2$
acq(ℓ)	
$\mathbf{w}(\mathbf{x})$	
rel(ℓ)	
	acq(ℓ)
	$w(x)$
	$r(x)$
	rel(ℓ)
	$\mathbf{r}(\mathbf{x})$

$\longrightarrow$

$\mathcal{T}_1$	$\mathcal{T}_2$
	acq(ℓ)
	$w(x)$
	$r(x)$
	rel(ℓ)
acq(ℓ)	
	$\mathbf{r}(\mathbf{x})$
$\mathbf{w}(\mathbf{x})$	

Contributions

M2:

- A new algorithm for predicting data races
- Efficient (poly-time)
- Sound (no false positives)
- Complete for 2 threads (no false negatives either)
- Dynamic completeness criteria (for given input)

Our Setting

- k threads running in parallel (every trace has a finite k)
- communication over shared variables $x, y, \dots$
- synchronization over locks $\ell_1, \ell_2, \dots$

Each thread executes *global events*:

- Write to global variable $\mathbf{w}(\mathbf{x})$
- Read from global variable $\mathbf{r}(\mathbf{x})$
- Acquire a lock $\mathbf{acq}(\ell)$
- Release a lock $\mathbf{rel}(\ell)$

Ignore local (invisible) computation

Our implementation also supports fork-join
(dependency to first and from last instruction of new thread)

Traces

A (concurrent) **trace** is a sequence of events

$$t = w(x), \mathsf{acq}(\ell), r(x), w(x), w(y), \mathsf{rel}(\ell), \mathsf{acq}(\ell), r(y), w(y), \mathsf{rel}(\ell)$$

- Events belong to different threads
  ($w(x)$ is e.g. $w(x)_2$ where 2 indicates the thread identifier)
- Locks mark **critical sections**
- Each read **observes** the preceding write to the same variable
  - Observation function $\mathcal{O}_t : \mathcal{R}(t) \to \mathcal{W}(t)$

Traces

A (concurrent) **trace** is a sequence of events

$$t = w(x), \mathsf{acq}(\ell), r(x), w(x), w(y), \mathsf{rel}(\ell), \mathsf{acq}(\ell), r(y), w(y), \mathsf{rel}(\ell)$$

- Events belong to different threads
  ($w(x)$ is e.g. $w(x)_2$ where 2 indicates the thread identifier)
- Locks mark **critical sections**
- Each read **observes** the preceding write to the same variable
  - Observation function $\mathcal{O}_t : \mathcal{R}(t) \to \mathcal{W}(t)$

Traces

A (concurrent) **trace** is a sequence of events

$$t = w(x), \mathsf{acq}(\ell), r(x), w(x), w(y), \mathsf{rel}(\ell), \mathsf{acq}(\ell), r(y), w(y), \mathsf{rel}(\ell)$$

- Events belong to different threads
  ($w(x)$ is e.g. $w(x)_2$ where 2 indicates the thread identifier)
- Locks mark **critical sections**
- Each read **observes** the preceding write to the same variable
  - Observation function $\mathcal{O}_t : \mathcal{R}(t) \to \mathcal{W}(t)$

Data Races

Definition (Conflicting Events)

Events e_1, e_2 are **conflicting** if

- they access the same variable
- one (at least) writes

Data Races

Definition (Conflicting Events)

Events e_1, e_2 are **conflicting** if

- they access the same variable
- one (at least) writes

Definition (Data Race in a Trace)

A **data race** in t is a conflicting pair of events e_1, e_2 which

- belong to different processes
- appear next to each other: $t = \ldots, e_1, e_2, \ldots$

$\mathcal{T}_1$	$\mathcal{T}_2$
acq(ℓ)	
$w(x)$	
rel(ℓ)	
	acq(ℓ)
	$r(x)$
	rel(ℓ)
w(x)	
	r(x)

Data Races

Definition (Conflicting Events)

Events e_1, e_2 are **conflicting** if

- they access the same variable
- one (at least) writes

Definition (Data Race in a Trace)

A **data race** in t is a conflicting pair of events e_1, e_2 which

- belong to different processes
- appear next to each other: $t = \ldots, e_1, e_2, \ldots$

$\mathcal{T}_1$	$\mathcal{T}_2$
acq(ℓ)	
$w(x)$	
rel(ℓ)	
	acq(ℓ)
	$r(x)$
	rel(ℓ)
$\mathbf{w(x)}$	
	$\mathbf{r(x)}$

Can this happen for some schedule?

Predictable Data Races

Definition (Predictable Race)

(e_1, e_2) is a **predictable race** in t if $\exists$ witness t^* such that

- t^* is a reordering of t
- $\mathcal{O}_{t^*}(r) = \mathcal{O}_t(r)$ for all reads r of t^*
- (e_1, e_2) is a data race in t^*

$\mathcal{T}_1$	$\mathcal{T}_2$
acq(ℓ)	
w(x)	
rel(ℓ)	
	acq(ℓ)
	$w(x)$
	$r(x)$
	rel(ℓ)
	r(x)

$\longrightarrow$

$\mathcal{T}_1$	$\mathcal{T}_2$
	acq(ℓ)
	$w(x)$
	$r(x)$
	rel(ℓ)
acq(ℓ)	
	r(x)
w(x)	

Predictable Data Races

Definition (Predictable Race)

(e_1, e_2) is a **predictable race** in t if $\exists$ witness t^* such that

- t^* is a reordering of t
- $\mathcal{O}_{t^*}(r) = \mathcal{O}_t(r)$ for all reads r of t^*
- (e_1, e_2) is a data race in t^*

$\mathcal{T}_1$	$\mathcal{T}_2$
acq(ℓ)	
w(x)	
rel(ℓ)	
	acq(ℓ)
	$w(x)$
	$r(x)$
	rel(ℓ)
	r(x)

$\longrightarrow$

$\mathcal{T}_1$	$\mathcal{T}_2$
	acq(ℓ)
	$w(x)$
	$r(x)$
	rel(ℓ)
acq(ℓ)	
	r(x)
w(x)	

We saw this . . .

. . . we predicted this

Predictive Race Detection

Problem Statement

Given a trace t, report *all* predictable races (e_1, e_2) of t

Predictive Race Detection

Problem Statement

Given a trace t, report *all* predictable races (e_1, e_2) of t

Soundness:
if you report (e_1, e_2) then (e_1, e_2) is a true race

Completeness:
if (e_1, e_2) is a true race then you report (e_1, e_2)

Predictive Race Detection

Problem Statement

Given a trace t, report *all* predictable races (e_1, e_2) of t

Soundness:
if you report (e_1, e_2) then (e_1, e_2) is a true race

Completeness:
if (e_1, e_2) is a true race then you report (e_1, e_2)

NP-complete

Sound & Complete
Inefficient

Incomplete
Efficient

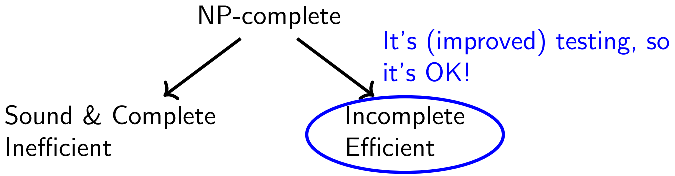
Predictive Race Detection

Problem Statement

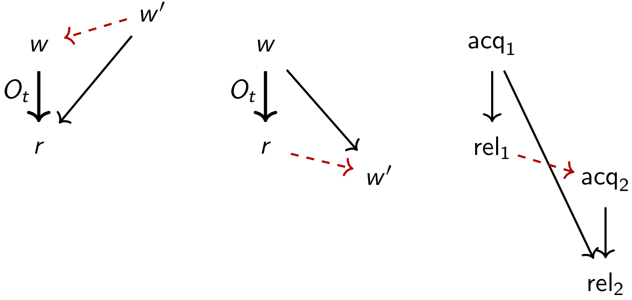
Given a trace t, report *all* predictable races (e_1, e_2) of t

Soundness:
if you report (e_1, e_2) then (e_1, e_2) is a true race

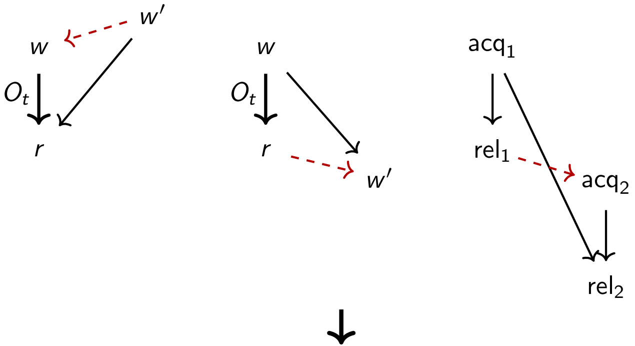
Completeness:
if (e_1, e_2) is a true race then you report (e_1, e_2)



Trace-closed Partial Orders

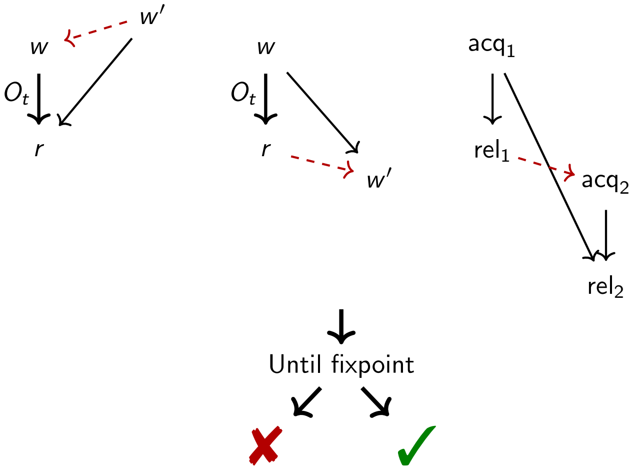


Trace-closed Partial Orders



Until fixpoint

Trace-closed Partial Orders



A pair of conflicting events not ordered
$\rightsquigarrow$ can occur one right after another $\rightsquigarrow$ race

Main Algorithm

Theorem

The trace-closure of a partial order can be computed in $\tilde{O}(n^2)$ time.

The algorithm uses Fenwick tree data structure to incrementally add edges to partial order.

Complexity is parametrized with respect to the number of threads k and relies on bounded tree width of the partial order for bounded k.

More details: `https://arxiv.org/abs/1901.08857`

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.

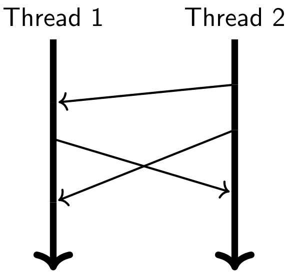
Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.

Max-min linearizations



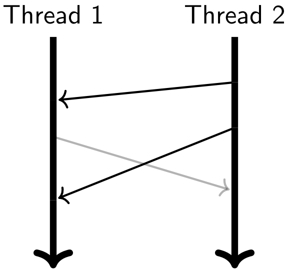
Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.

Max-min linearizations

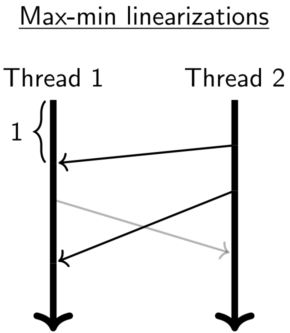


Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.



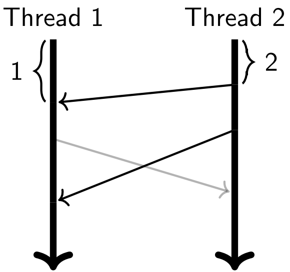
Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.

Max-min linearizations



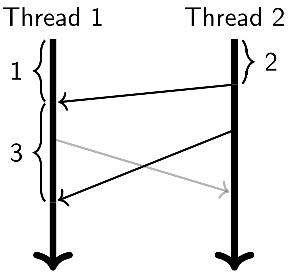
Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.

<u>Max-min linearizations</u>

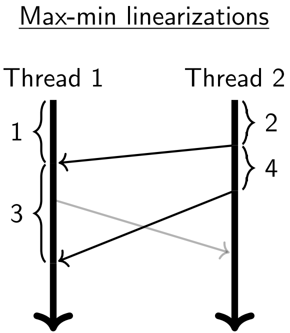


Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.



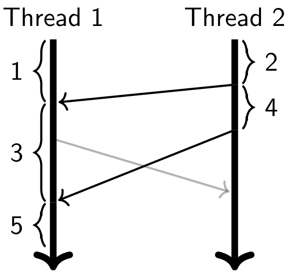
Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

(for 2 threads) A trace-closed partial order is linearizable to a valid trace.

Max-min linearizations

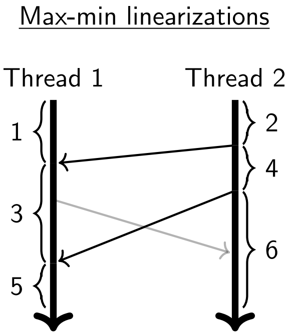


Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

Predictive Race Detection

Theorem

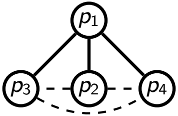
(for 2 threads) A trace-closed partial order is linearizable to a valid trace.



Each event of t_1 before each event of t_2 whenever doing this does not a create cycle. Then linearize arbitrarily.

More than 2 Threads?

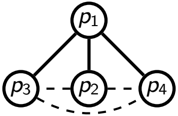
We don't know



- Closure as before
- While ∃ events e_1, e_2 in the leaves and unordered
  - Order them as in the input trace
  - Closure again
- If no cycle, max-min linearization works!

More than 2 Threads?

We don't know



Incomplete in general
Works in practice

- Closure as before
- While ∃ events e_1, e_2 in the leaves and unordered
  - **Order them as in the input trace**
  - Closure again
- If no cycle, max-min linearization works!

Implementation & Experiments

Implementation in Java

Comparison against

- Doesn't-Commute **DC** [RGB, PLDI '18]
- Schedulably-Happens-Before **SHB** [MKV, OOPSLA '18]

on a standard benchmark set of traces

Experimental Results

Benchmark	*k*	*n*	Races			Time			∀
			DC	SHB	M2	DC	SHB	M2	
`mergesort`	4	3.0K	29	2	**53**	2.16s	0.37s	**0.15s**	✓
`bubblesort`	10	4.0K	478	802	**909**	2.28s	0.62s	2.14s	✓
`raytracer`	2	16K	667	667	667	2.57s	0.51s	**0.24s**	✓
`ftpserver`	10	48K	95	87	**116**	2.75s	0.73s	1.79s	✓
`derby`	3	1.0M	38	39	39	15.29s	8.32s	**7.15s**	✓
`jigsaw`	12	3.0M	17	18	**20**	40.89s	17.93s	**12.80s**	✓
`bufwriter`	5	11M	11	11	11	2m59s	47.71s	**2m10s**	✓
`cryptorsa`	6	43M	7	5	**26**	6m18s	2m46s	**2m36s**	✓
`eclipse`	14	90M	465	662	**898**	14m44s	7m11s	1h58m42s	?
`xalan`	6	122M	72	89	**97**	20m12s	9m8s	**7m56s**	✓
`lusearch`	7	217M	170	360	360	2h49m6s	15m28s	**7m36s**	✓

✓ means M2 proved that it found **all** races (even though $k > 2$)

Future Directions

- Testing is lightweight; explore efficiency $\leftrightarrow$ effectiveness
- What's the best you can do in
  - $O(n^c)$? $O(n)$?
- Deadlocks, atomicity violations

Future Directions

- Testing is lightweight; explore efficiency $\leftrightarrow$ effectiveness
- What's the best you can do in
  - $O(n^c)$? $O(n)$?
- Deadlocks, atomicity violations
- Relaxed memory models
- Other communication primitives
  - message passing

Future Directions

- Testing is lightweight; explore efficiency $\leftrightarrow$ effectiveness
- What's the best you can do in
  - $O(n^c)$? $O(n)$?
- Deadlocks, atomicity violations

- Relaxed memory models
- Other communication primitives
  - message passing

- Predict with a *set* of traces
- Drive trace generation
- Static + Predictive

Future Directions

- Testing is lightweight; explore efficiency $\leftrightarrow$ effectiveness
- What's the best you can do in
  - $O(n^c)$? $O(n)$?
- Deadlocks, atomicity violations

- Relaxed memory models
- Other communication primitives
  - message passing

- Predict with a *set* of traces
- Drive trace generation
- Static + Predictive

- Predict quantitative properties
- Worst-case waiting time to acquire a resource
- Least amount of context switches

Conclusion

- A new algorithm for predicting data races
- Efficient (poly-time)
- Sound (no false positives)
- Complete for 2 threads (no false negatives either)
- Effectively complete on our benchmarks (we detect that it is)