ISHCS 2016 (International Symposium on High Confidence Software), PKU, Beijing, Dec. 18, 2016

Probabilistic Detection and Sampling of Concurrency Bugs

Yan Cai (蔡彦)

ycai.mail@gmail.com

State Key Lab. of Computer Science,
Institute of **S**oftware, **C**hinese **A**cademy of **S**ciences

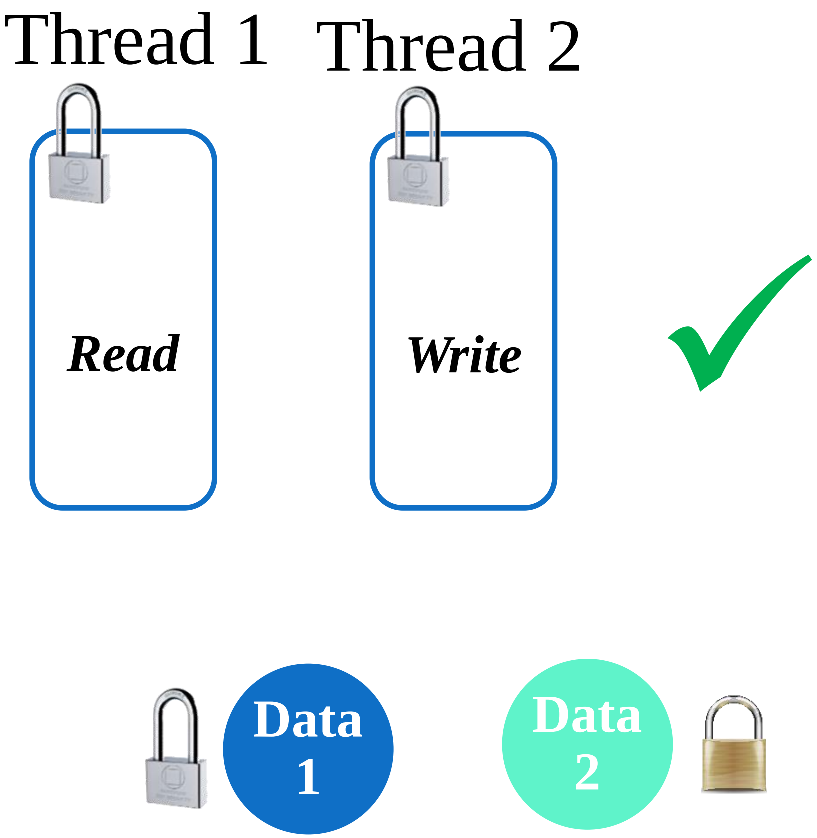
中科院软件所·计算机科学国家重点实验室

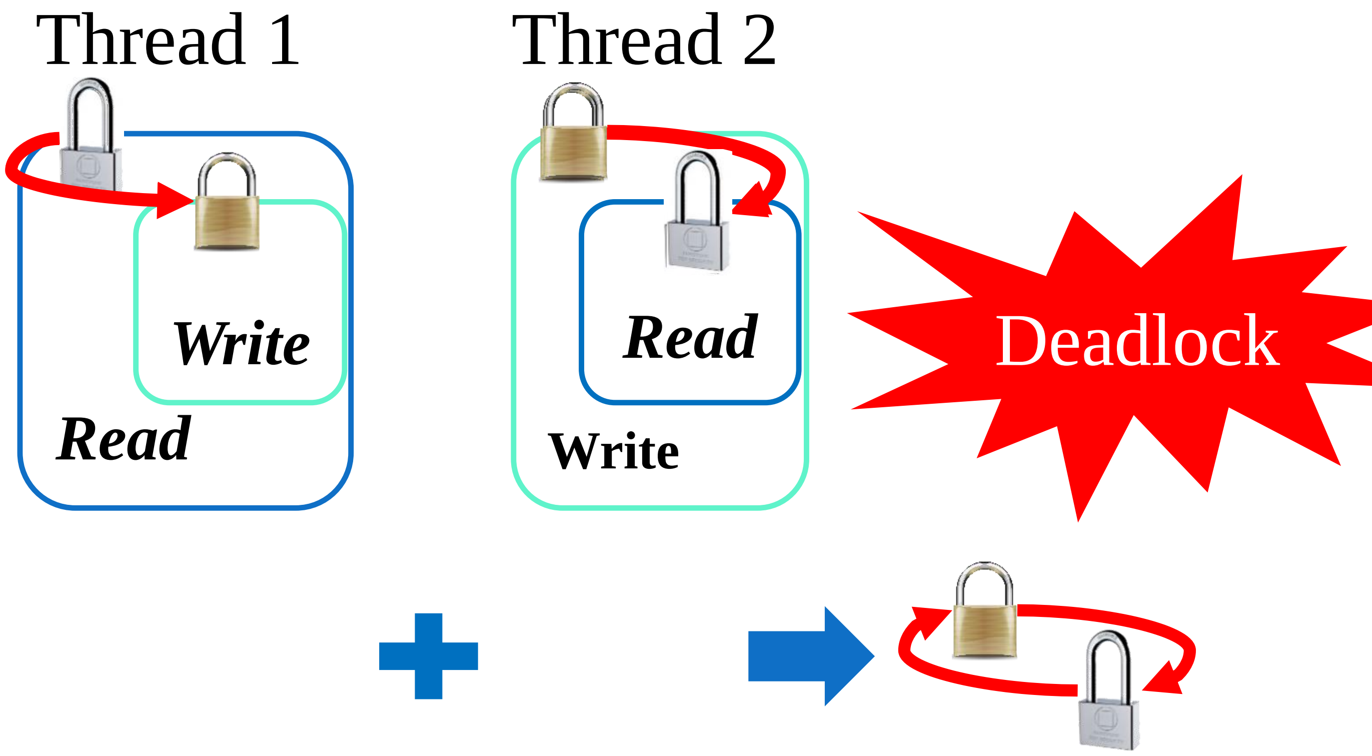
Radius-aware Probabilistic Deadlock detection

ASE’16

Yan Cai and Zijiang Yang

Locks and Deadlocks





Thread t_1	Thread t_2
acq(m)	*acq(n)*
acq(n)	*acq(m)*

Deadlock Testing

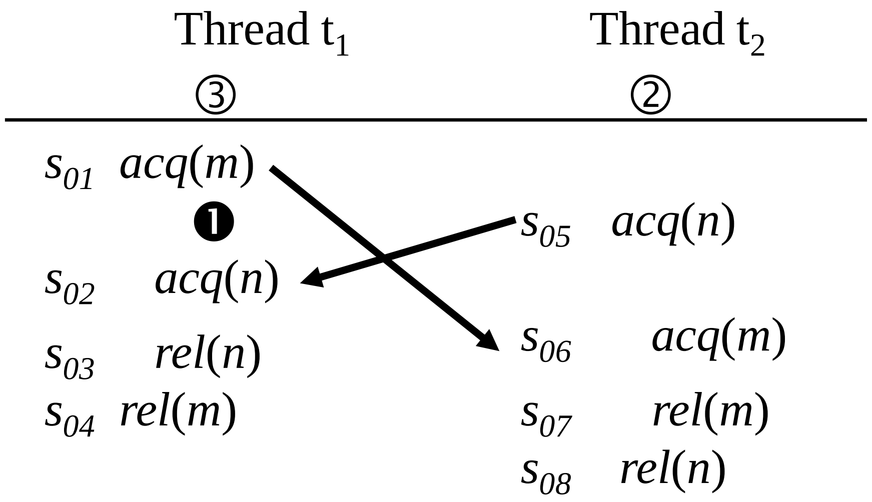
- Random testing
  - OS scheduling + random manipulation
  - Stress testing
  - Heuristic directed random testing
  - Systematic scheduling

No Guarantee to find a concurrency bug (e.g., Deadlock)

PCT – Probabilistic Concurrency Testing

- PCT Algorithm
  - Mathematical randomness with Probabilistic Guarantees

$$\frac{1}{n \times k^{d-1}}$$ n: #threads, k: #events, d: bug depth



k =8, n =2, d =2

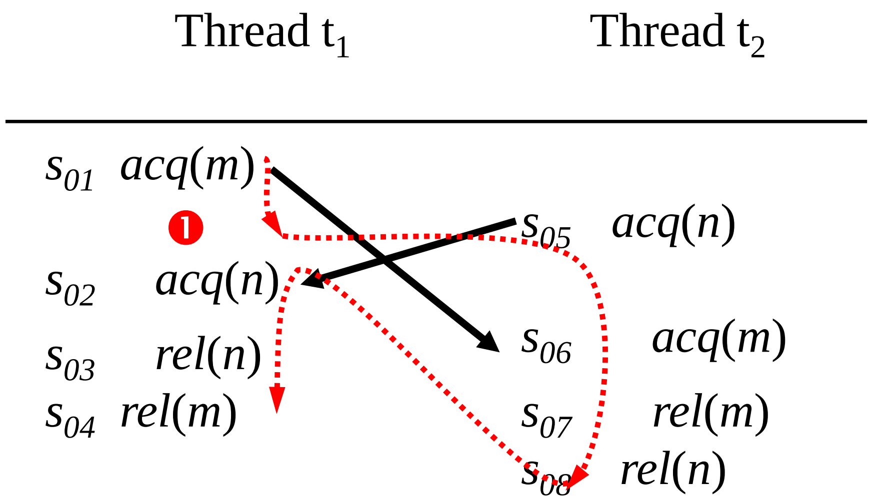
$$\frac{1}{2 \times 8^{2-1}} = 1/16$$

PCT – Probabilistic Concurrency Testing

- PCT :
  - Intuition of guaranteed probability:

    1. satisfy the 1st order by assigning the thread a largest **priority** $(1/n)$

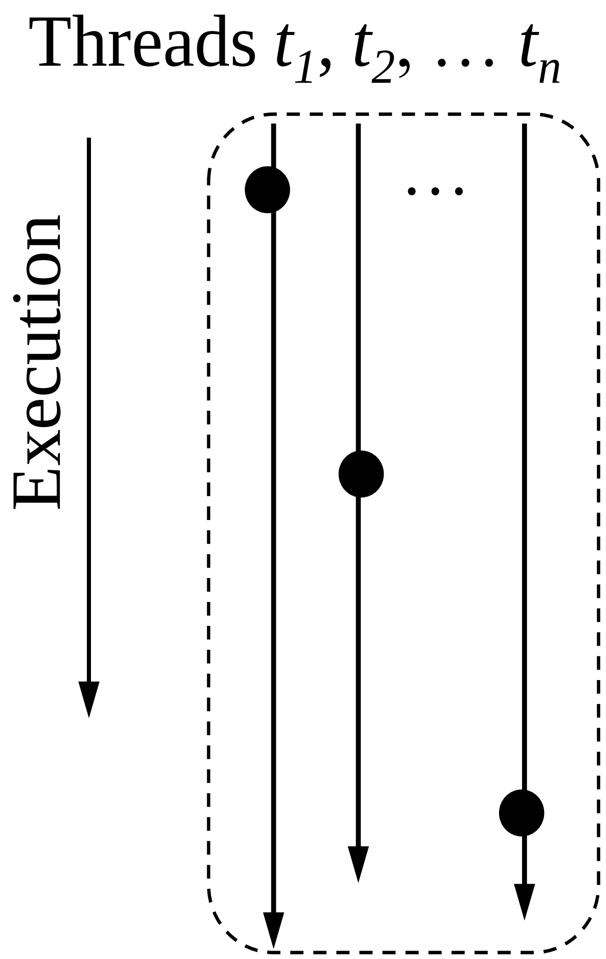
    2. select d – 1 **priority change points** at the remaining d – 1 order position $(1/k \times 1/\mathrm{k} \times \ldots \times 1/k = \frac{1}{k^{d-1}}) \Rightarrow \frac{1}{n \times k^{d-1}}$



$k=8$, $n=2$, $d=2$

$$\frac{1}{2 \times 8^{2-1}} = 1/16$$

PCT – Probabilistic Concurrency Testing



- Provide a guarantee (a probability):

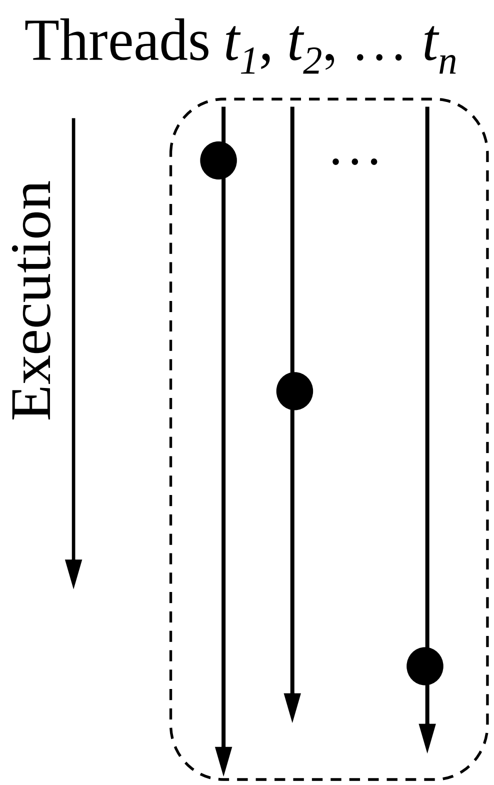
$$\frac{1}{n \times k^{d-1}}$$ n: #threads, k: #events, d: bug depth

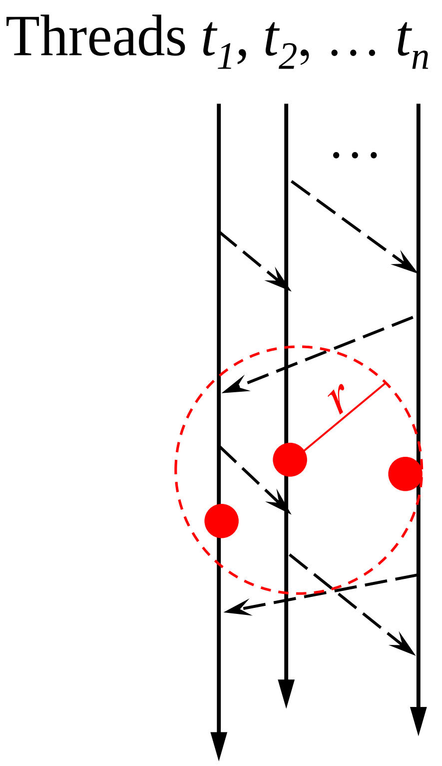
But …

- Theoretical model, not consider thread interaction:
  real executions do not follow designed executions
- Guaranteed probability decreases exponentially with increase of bug depth: due to factor $\frac{1}{k^{d-1}}$.

RPro- Radius aware

- Our approach: RPro – Radius aware Probabilistic testing





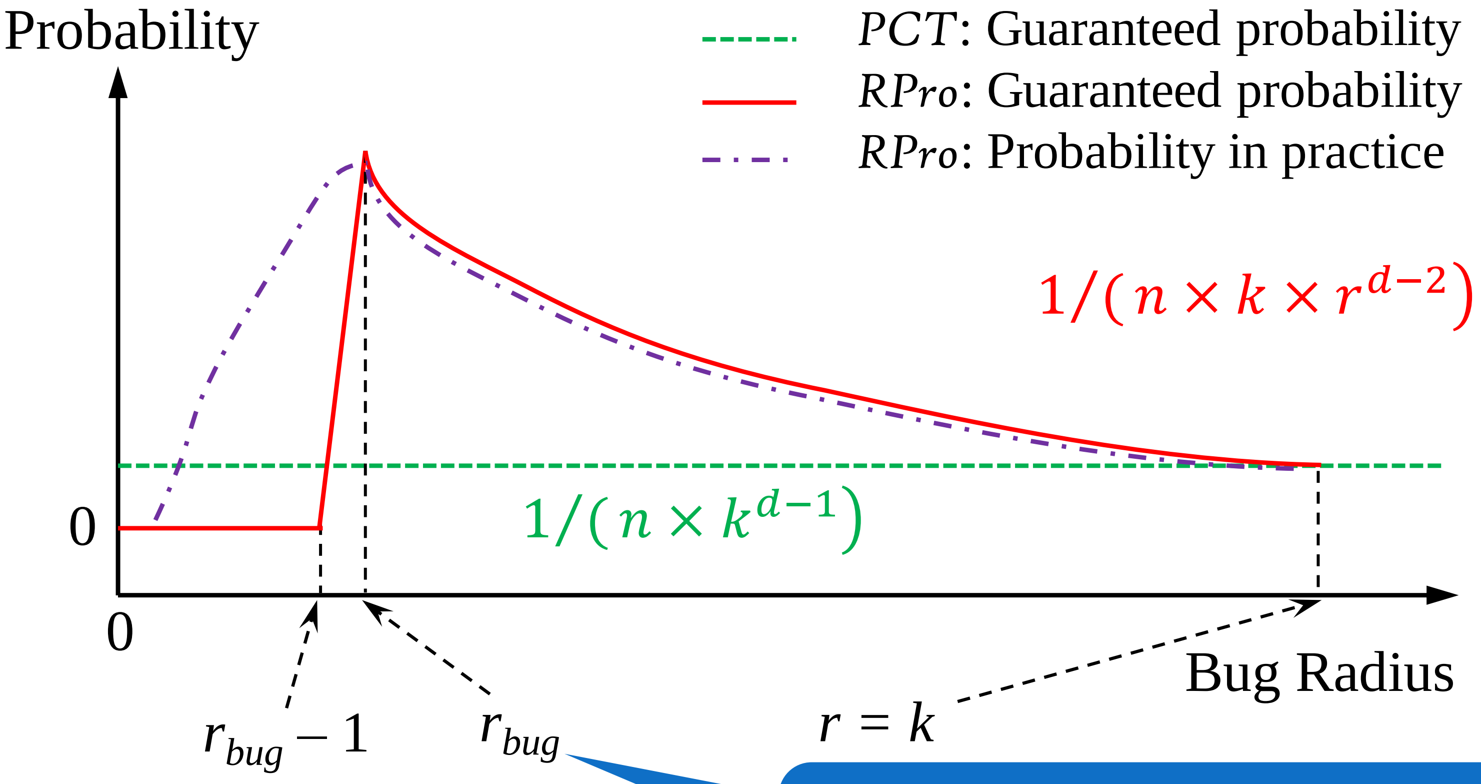
- Consider thread interaction
- Guaranteed probability decreases: $\frac{1}{r}$ (not $\frac{1}{k}$, $r \ll k$)

$$\frac{1}{n \times k^{d-1}} \qquad \frac{1}{n \times k \times r^{d-2}}$$

PCT v.s. RPro

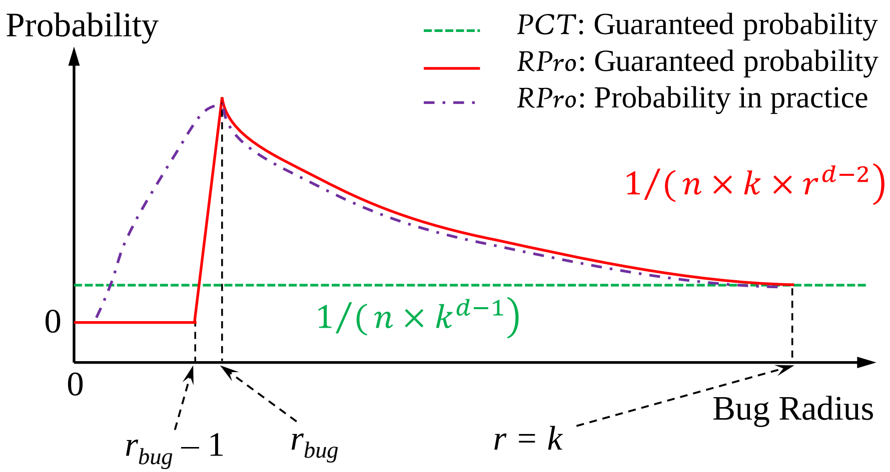
RPro- Radius aware

- RPro: Theoretical guarantee



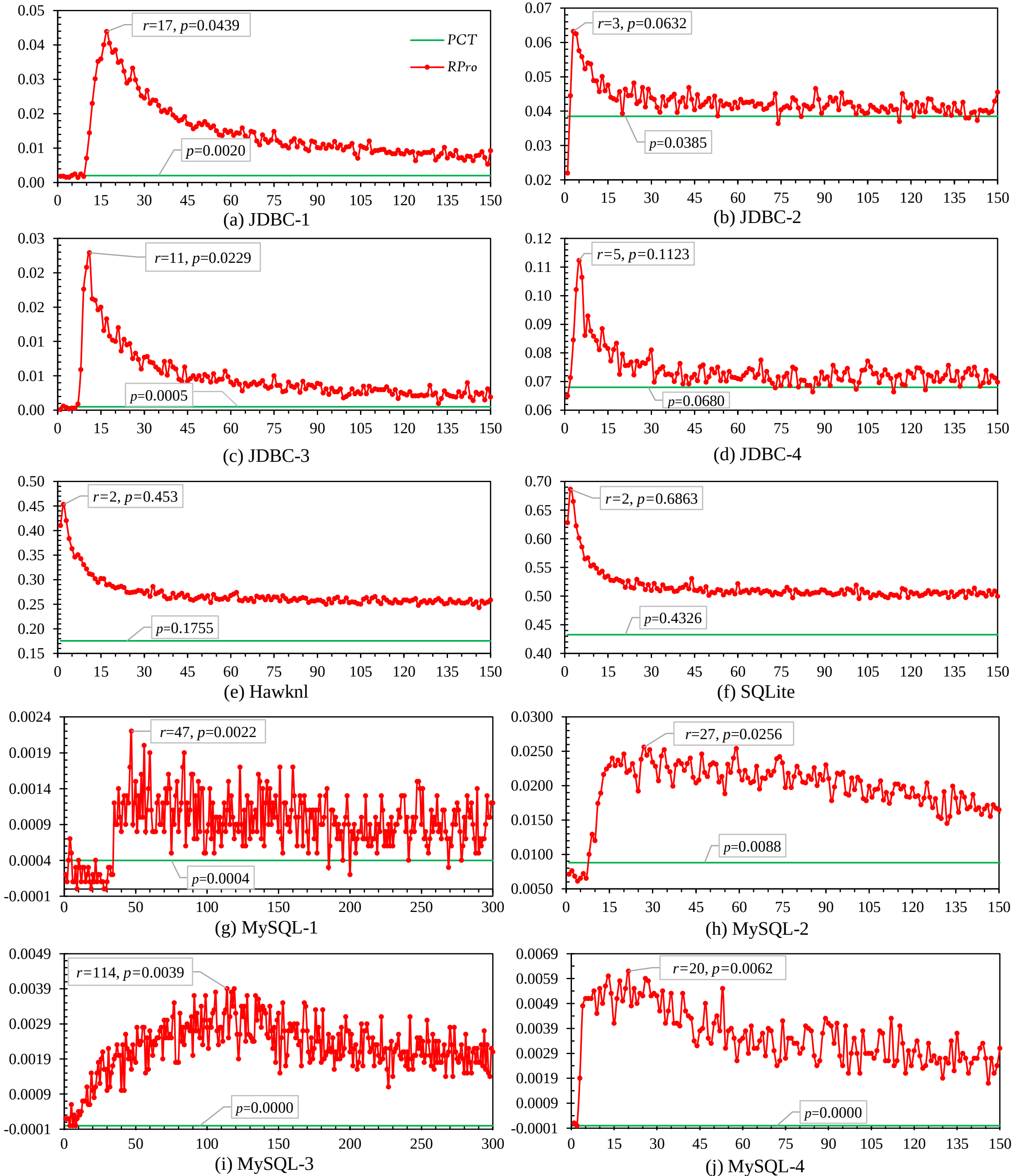
How to find r_{bug}?

Experiment



Benchmark	# events	# threads	*bug depth*	r_{best}*	$\frac{r_{best}}{\#events}$	*Probability*
Hawknl	28	3	3	2	-	0.4530
SQLite	16	3	3	2	-	0.6863
JDBC-2	5,050	3	3	3	0.059%	0.0632
JDBC-4	5,090	3	3	5	0.098%	0.1123
JDBC-3	5,080	3	3	11	0.217%	0.0229
JDBC-1	5,088	3	3	17	0.334%	0.0439
MySQL-4	444,621	19	3	20	0.005%	0.0062
MySQL-2	15,066	17	3	27	0.179%	0.0256
MySQL-1	19,300	16	3	47	0.244%	0.0022
MySQL-3	406,117	22	6	114	0.028%	0.0039

(* All rows are sorted on the data in this column.)



Deployable Data Race Sampling

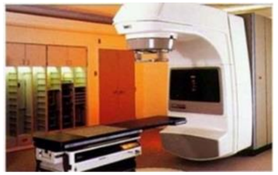
FSE'16

Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu

Concurrency bugs

- Difficult to detect
  - Non-determinism (space explosion)
  - Inadequate test inputs
  - …

- Even after software release, concurrency bugs may still occur



Concurrency bugs

- It is necessary to detect concurrency bugs in deployed products
- Challenges:
  not to disturb normal executions
  - light-weighted <5% overhead
  - …

Sample user executions

Existing works

- Data Race

> Two threads concurrently access the same memory location and at least one access is a write.

- Happens-before (HB Race)
  - Access pairs not ordered by happens-before relation (HBR)

```
Thread t1     Thread t2

x++;
sync(m){}
              sync(m)
              {x++;}
```

Value of x: +2.

```
Thread t1     Thread t2

x++;
              sync(m)
sync(m){}     {x++;}
```

Value of x: +1 or +2?

Existing works

- Happens-before Races
  - Track full Happens-before relation
    - ***Incurring many O(n) operations***

0% sampling rate => ~30% overhead

(Pacer, PLDI'10)

~15% in our experiment

Insight 1:
Not to track Full Happens-before Relation

Existing works

- Hardware based (e.g., DataCollider, OSDI'10)
  - **Code Breakpoints** and **Data Breakpoints** (or **Watchpoints**)
  - Collision Races
- A data race: two accesses
  - Select a memory address =>
    Set a data breakpoint =>
    Wait for the breakpoint to be fired
  - **The waiting time directly increases the sampling overhead**

Insight 2: Not to directly delay executions

Existing works

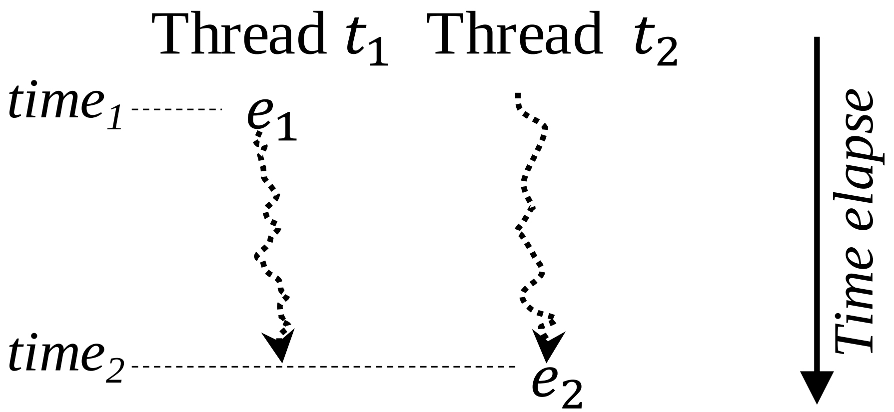
- …
- *See our paper for more insights*

Our Proposal

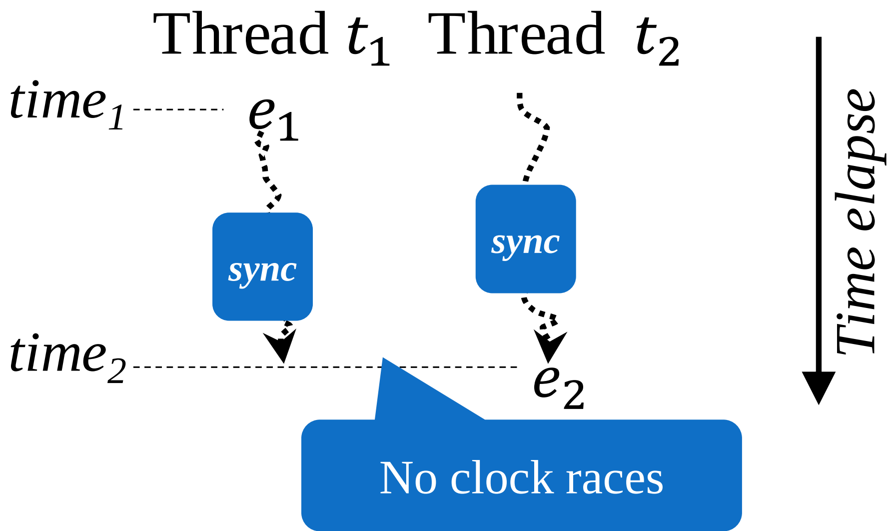
- Clock Race
  - For data race sampling purpose
- CRSampler
  - To detect clock races

Clock Race

- **Clock Race**
  - **Thread-local clock**: an integer for each thread, increased on synchronization operation.
  - Two accesses (with at least a write) form a Clock Race if: ***at least one thread-local clock is not changed in between the two accesses***



$t_1.clock$ is not changed between $time_1$ and $time_2$.



Clock Race

- A Quick Demonstration

Maintain thread-local clocks

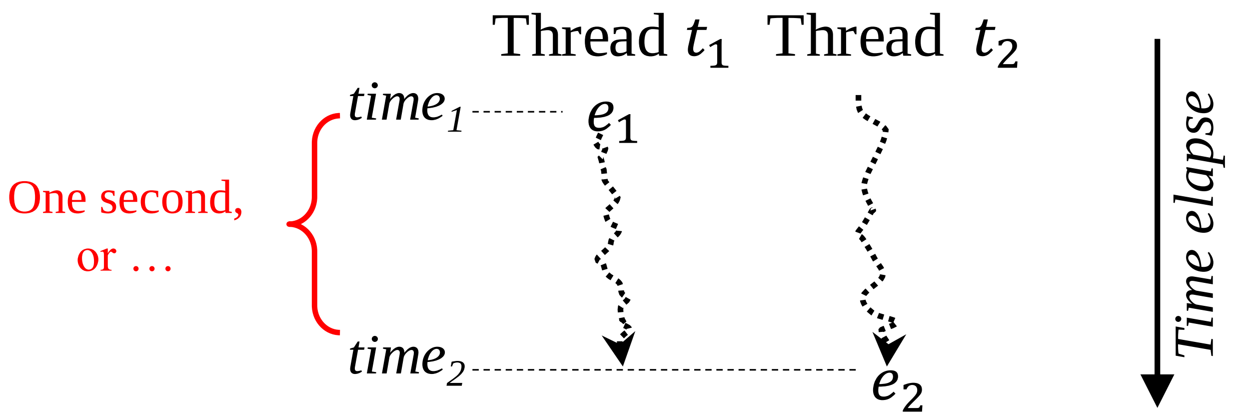
Thread t_1		$t_1.clock$	$t_2.clock$	Thread t_2	
		10	8		
acquire(*l*)	**onSync( );**	11	9	*acquire*(*k*)	**onSync( );**
***x* = 0;**	**sample(*x*);**	11	9	…	
…		11	9		
		11	10	*release*(*k*)	**onSync( );**
		11	10	***x*** ++;	
release(*l*)	**onSync( );**	12			

Sampled access

On this read, $t_1.clock$ remains 11,
a clock race on ***x*** is reported

Clock Race

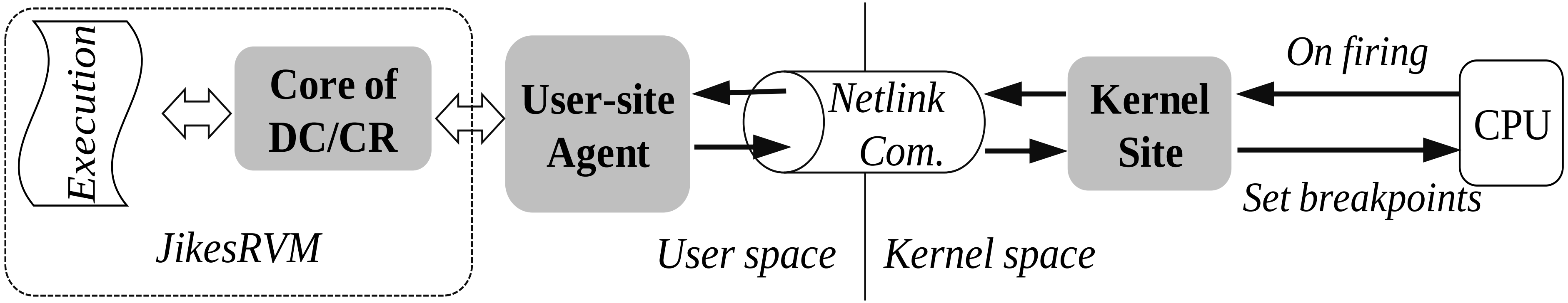
- Clock Race
  - Race checking does not need to delay any thread.
  - But: after e_1 appears, how much time is required to check two accesses?
    - Given a short time, it is not enough to trap the second access.
    - Given a long time, all threads' lock clocks are changed.



$t_1.clock$ is not changed between $time_1$ and $time_2$.

Setup

- Implementation
  - Jikes RVM
  - Sampling: Java class load time
  - Memory accesses ⇔ Linux Kernel



- Benchmarks
  - Dacapo benchmark suite

Setup

- Comparisons
  - Sampling rate: 0.1% to 1.0%
  - **Pacer** (PLDI'10)
  - **Data Collider** (OSDI'10)
  - **CRSampler**

15ms, 30ms

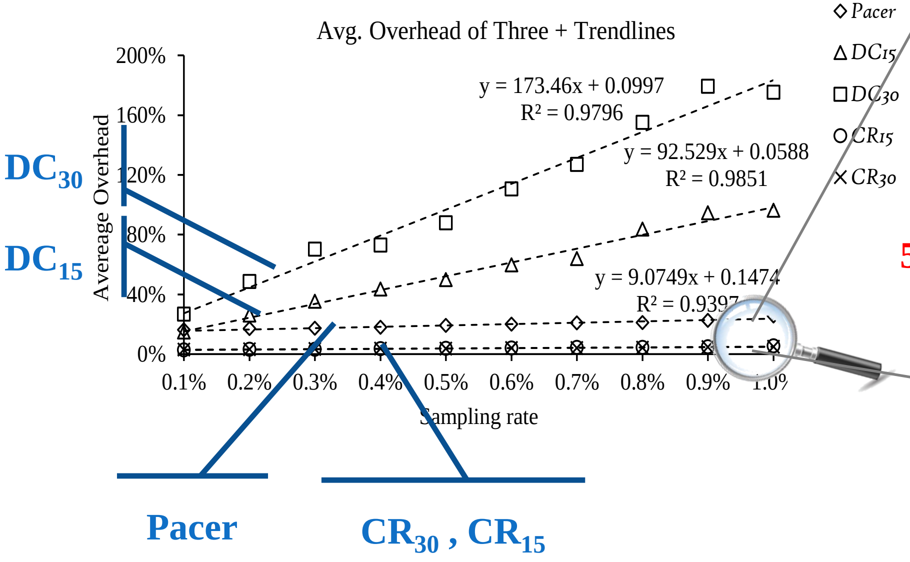
DC_{15}, DC_{30}
CR_{15}, CR_{30}

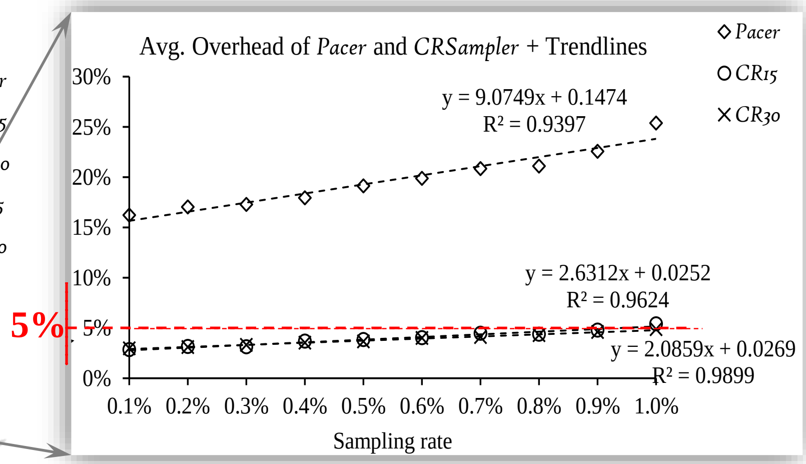
- ThinkPad Workstation
  - I7-4710MQ CPU, four cores, 16G memory, 250G SSD

Experiments

- Overall Results
  - Effectiveness
    - CR: more data races at low sampling rates
  - Overhead

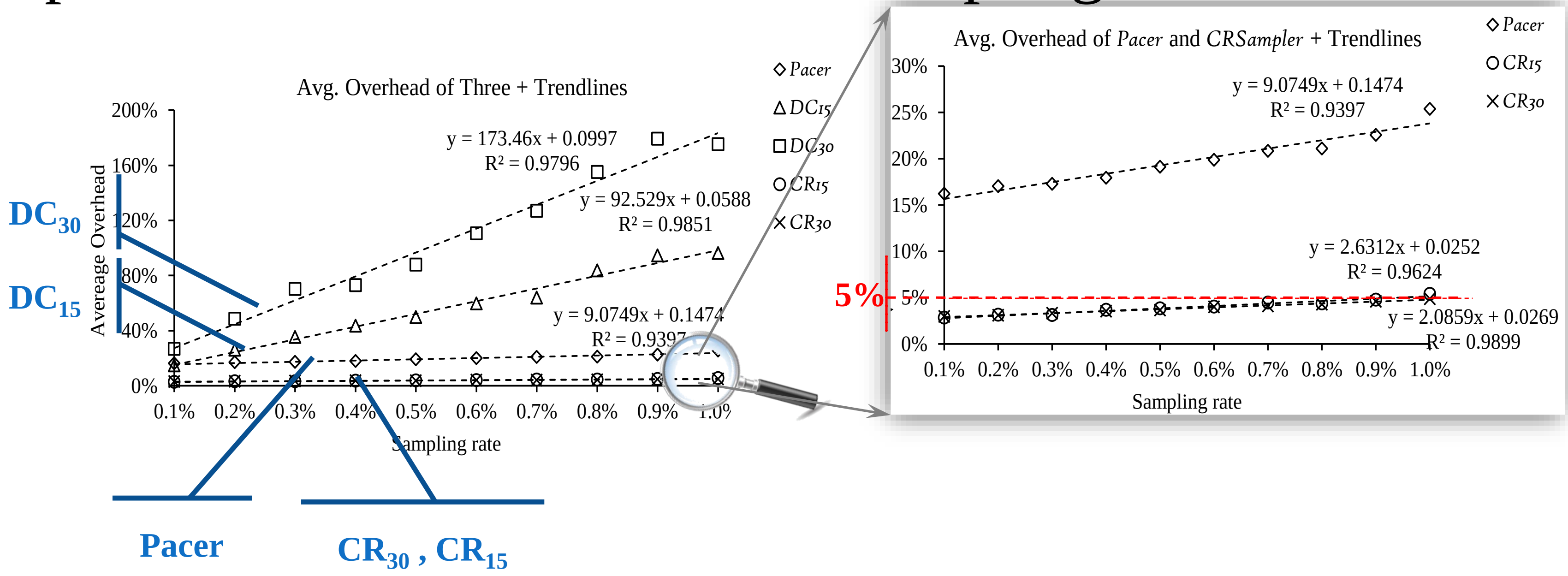
Bench-marks	Binary Size (KB)	# of threads	# of sync.	Pacer*	DC_{15}	DC_{30}	CR_{15}	CR_{30}
$avrora_{09}$	2,086	7	3,312,801	3	3	3	5	3
$xalan_{06}$	1,027	9	35,859,489	5	5	5	87	81
$xalan_{09}$	4,827	9	12,599,144	0	2	2	84	91
$sunflow_{09}$	1,017	17	1,590	0	0	2	46	45
pmd_{09}	2,996	9	20,550	4	2	2	110	121
$eclipse_{06}$	41,822	16	51,131,093	19	2	6	58	63
			Sum:	31	14	20	390	404





Experiments

- Discussions
  - DataCollider: overhead from its delays.
    - DC_{30} has almost **2 times** overhead than DC_{15}.
  - Pacer: basic overhead **~15%**
  - CRSampler: **~5%** overhead at **1.0%** sampling rate.



Thanks~