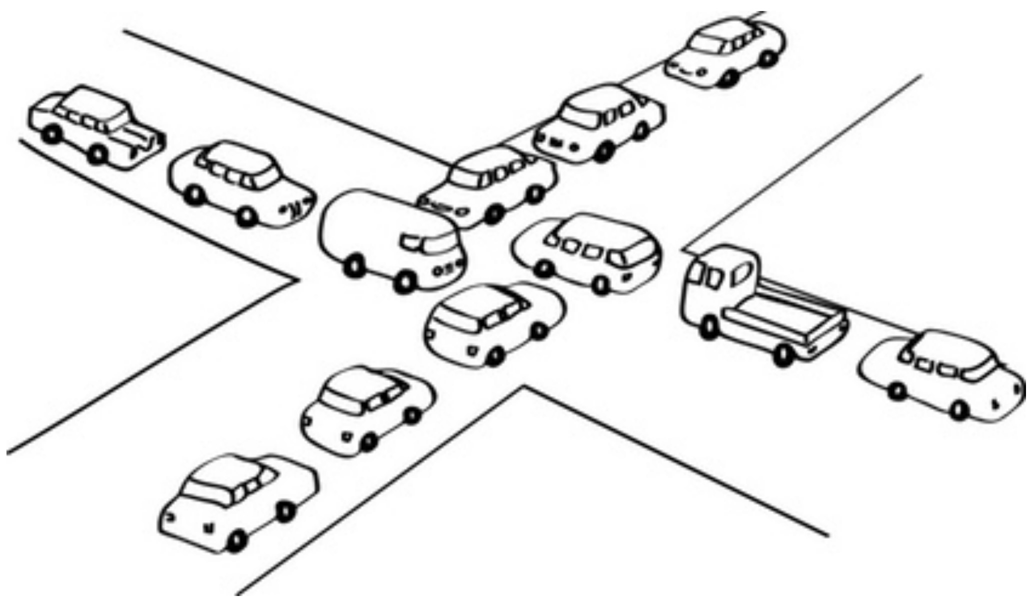
Lockout: Efficient Testing for Deadlock Bugs

Ali Kheradmand, Baris Kasikci, George Candea

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Deadlock

- Set of threads
  - Each holding a lock needed by another thread
  - Waiting for another lock to be released by some other thread



Why Do Deadlocks Matter?

- Common in modern software
- Hard to detect manually
- Occur rarely during execution

MySQL

Chrome

Apache

Eclipse

LimeWire

Deadlock Detection

- Traditional testing
  - Deadlocks manifest rarely ✗
- Static detection
  - Fast (run offline) ✓
  - Few false negatives ✓
  - Many false positives ✗
- Dynamic detection
  - Slow (high runtime overhead) ✗
  - Many false negatives ✗
  - Few false positives ✓

Best of Two Worlds

- Normal tests can't discover enough deadlocks
- Deadlock avoidance or fixing tools tools (Dimmunix [OSDI'08]) take a long time
  - Need to find the schedules that lead to a deadlock
- How to increase the probability of encountering a deadlock?

Steer the program towards schedules that are likely to cause a deadlock

Lockout

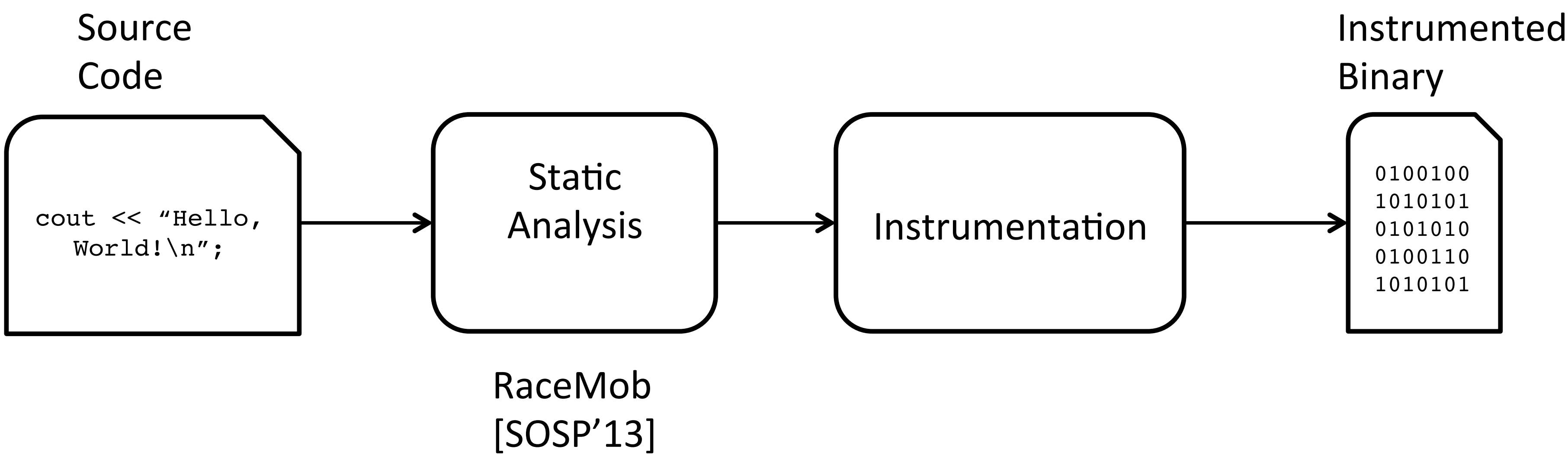
- Systematic deadlock testing
- Increases deadlock probability
  - By steering the scheduling
- Leverages past program executions

Trigger more deadlocks with the same test suite

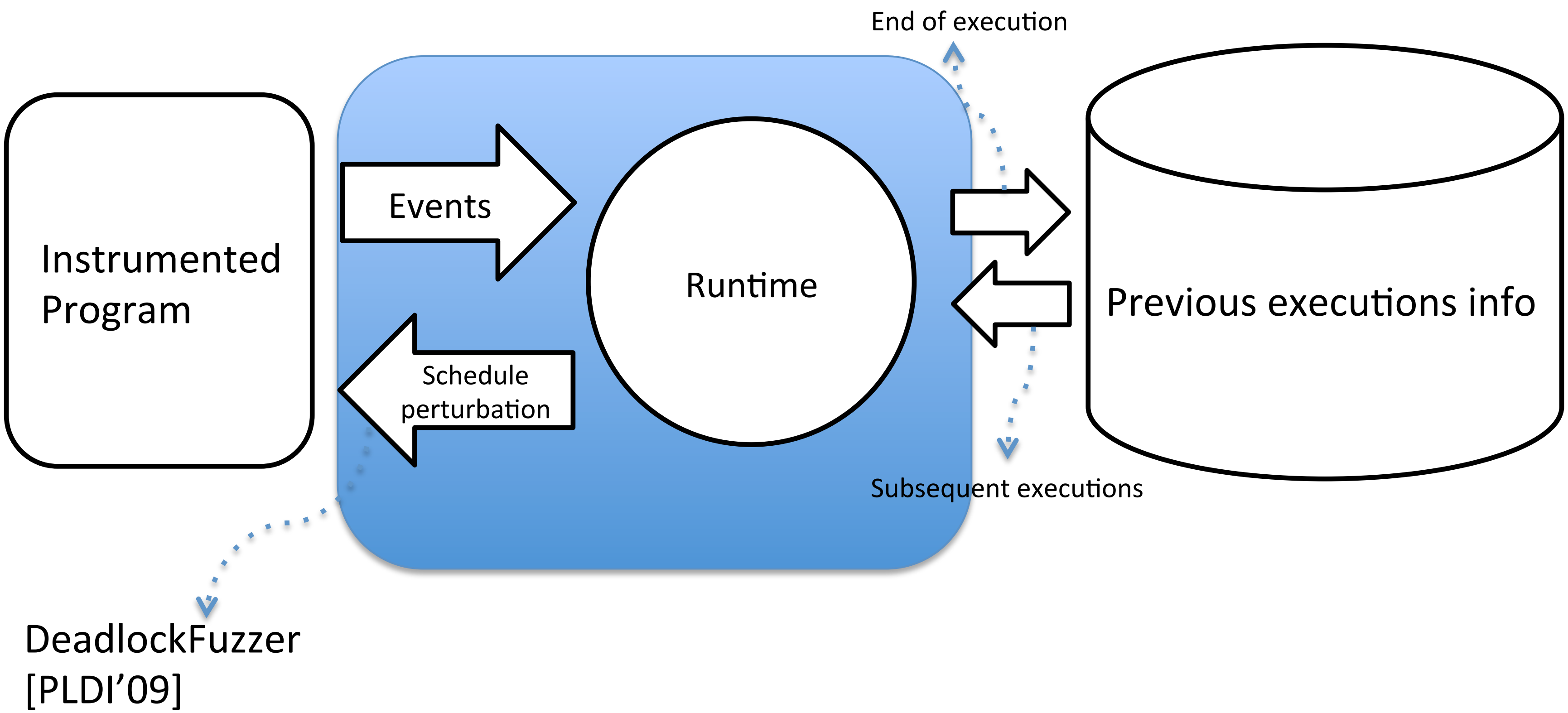
Outline

- Lockout architecture
- Deadlock triggering algorithm
- Preliminary results
- Summary and future work

Lockout Architecture
Static Phase



Lockout Architecture
Dynamic Phase



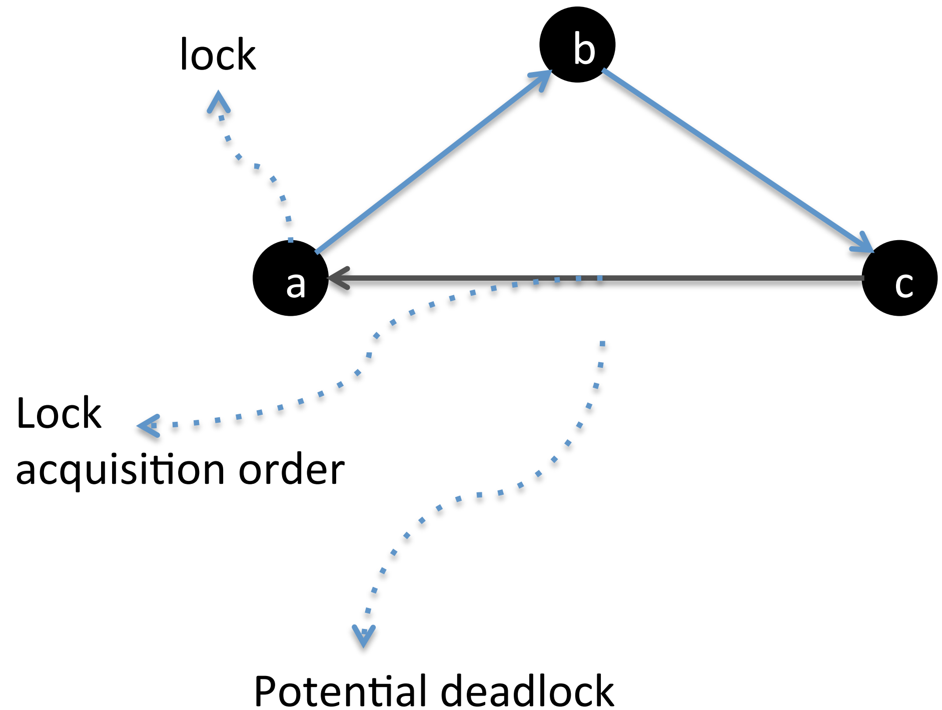
Outline

- Lockout architecture
- Deadlock triggering algorithm
- Preliminary results
- Summary and future work

Runtime Lock Order Graph (RLG)

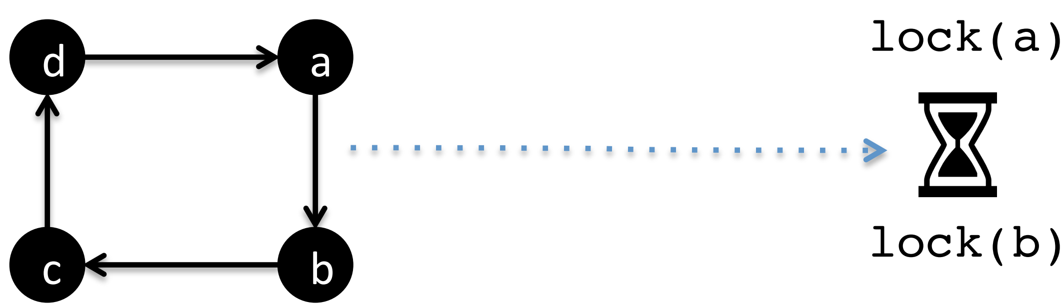
Thread 1

```
lock(a)
…
lock(b)
…
lock(c)
```



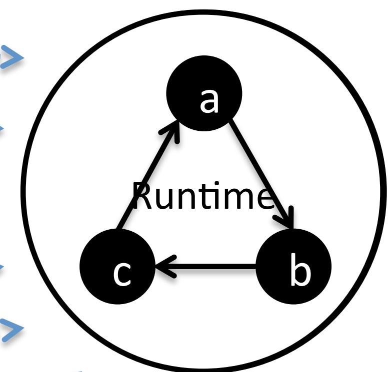
Deadlock Triggering

- Selects a directed cycle in RLG
- Delays threads accordingly
- Improves simple preemption (CHESS [OSDI'08])
  - Preemption before each lock

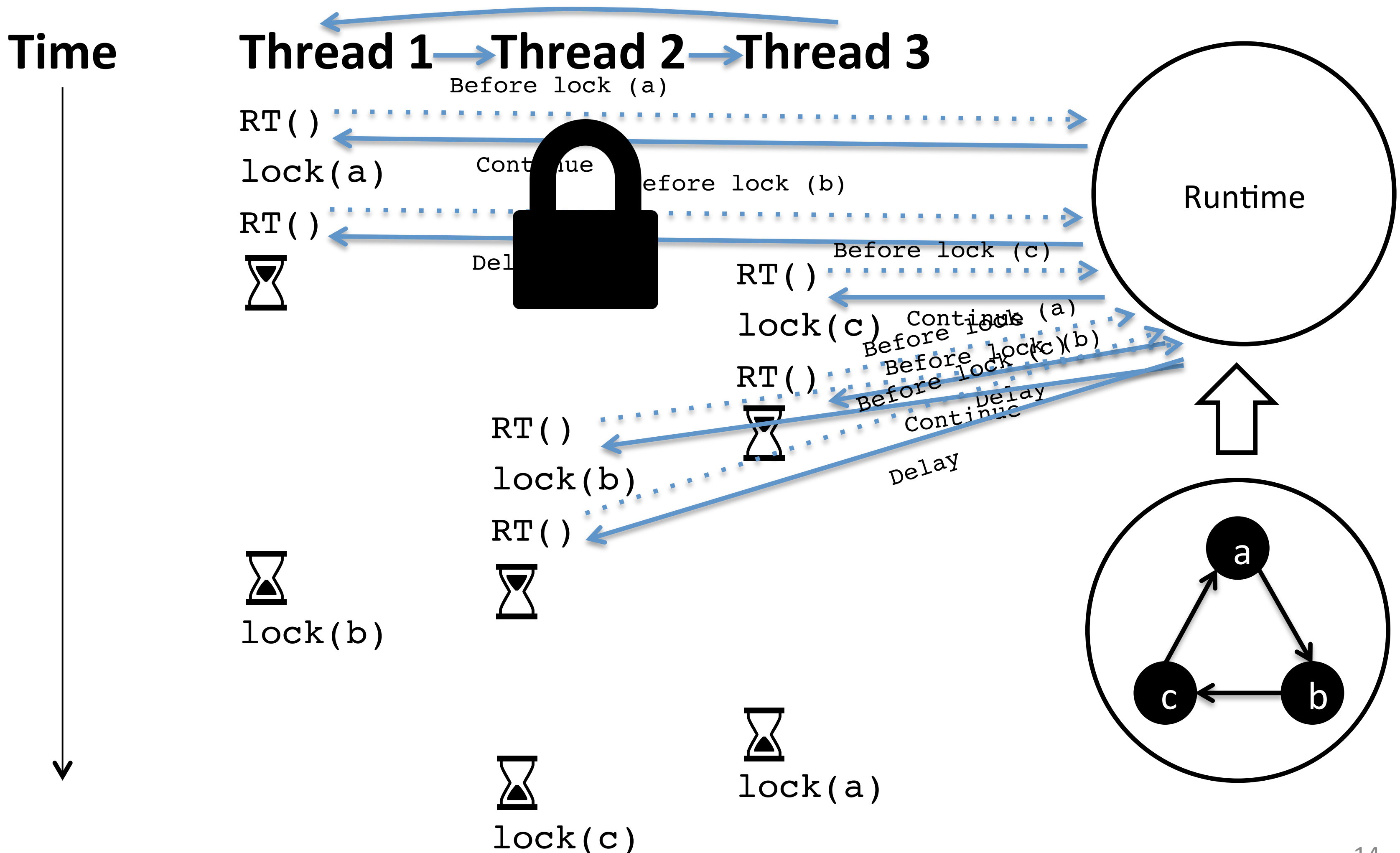


Deadlock Triggering

Time	Thread 1	Thread 2	Thread 3
	`lock(a)`		
	`lock(b)`		
	…		
	`unlock(b)`		
	`unlock(a)`		
		`lock(b)`	
		`lock(c)`	
		…	
		`unlock(c)`	
		`unlock(b)`	
			`lock(c)`
			`lock(a)`
			…
			`unlock(a)`
			`unlock(c)`



Deadlock Triggering



Race Dependent Deadlocks

- Preempt before memory accesses
  - Ideally only shared memory accesses
- Can be approximated by preempting after locks
- Can be improved using static analysis
- Can be improved using data race detection

Outline

- Lockout architecture
- Deadlock triggering algorithm
- Preliminary results
- Summary and future work

Lockout Effectiveness

Program	Fraction of executions resulting in deadlock (%)			
	Native	Simple preemption + Post-lock preemption	Deadlock triggering + Pre-memory access preemption	Deadlock triggering + Post-lock preemption
Microbench	0.00066 %	0.5 %	50 %	50 %
SQLite 3.3.0	0.00064 %	4 %	50 %	50 %
HawkNL 1.6b3	23 %	64 %	50 %	50 %
Pbzip2 1.1.6	0 %	0 %	0 %	0 %
Httpd 2.0.65	0 %	0 %	0 %	0 %

Fraction of executions with deadlocks increased up to three orders of magnitude

Outline

- Lockout architecture
- Deadlock triggering algorithm
- Preliminary results
- Summary and future work

Lockout

- Increases deadlock probability
- Leverages past program executions
- Effective
  - Up to 3 orders of magnitude more deadlock prone
- Open source:
  - **https://github.com/dslab-epfl/lockout**

Future Work

- Increasing effectiveness with low overhead
  - Static analysis (data races, shared variables)
- Lockout + Automatic failure fixing/avoidance (Dimmunix [OSDI'08], CFix [OSDI'12], Aviso [ASPLOS'13])
  - In production
  - For testing
- Crowdsourcing (Aviso [ASPLOS'13], RaceMob [SOSP'13])