Deterministic Replay and Data Race Detection for Multithreaded Programs

Dongyoon Lee

Computer Science Department

VirginiaTech

Invent the Future

The Shift to Multicore Systems

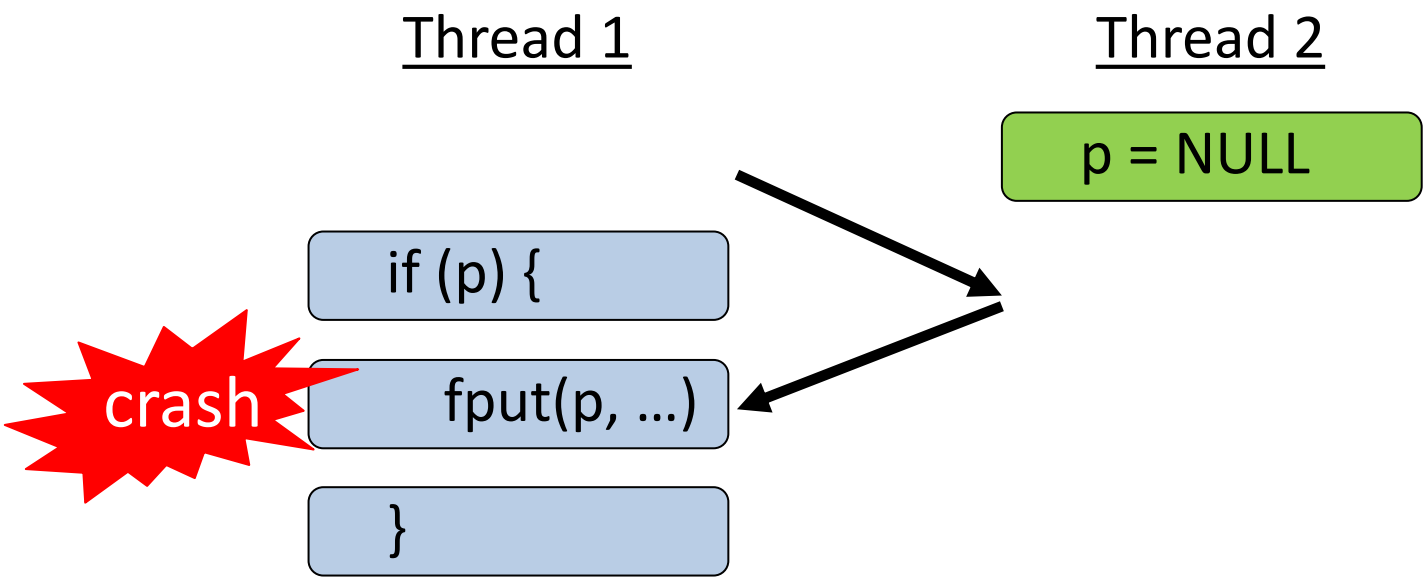
100+ cores
Desktop/Server

8+ cores
Smartphones

2+ cores
Smart Things

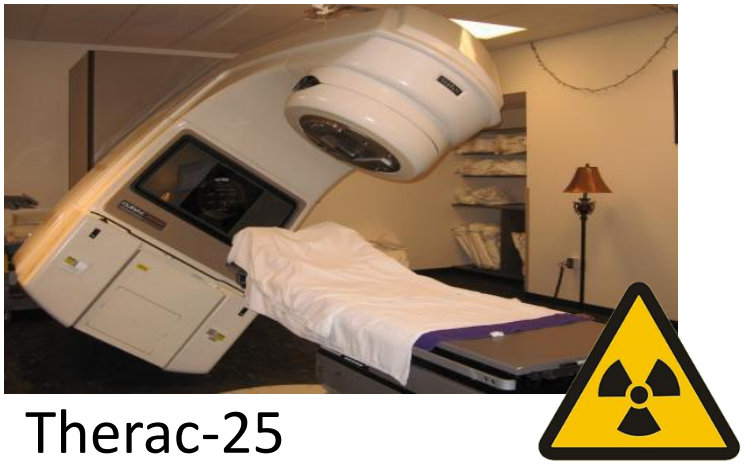
Data Races in Multithreaded Programs

- Two threads access the same shared location (one is write)
- Not ordered by synchronizations



MySQL bug #3596

Race Conditions Caused Severe Problems



Therac-25

Northeast Blackout

NEWS

Nasdaq's Facebook glitch came from 'race conditions'

Nasdaq may pay out as much as $13 million due to a hard-to-find software bug

By Joab Jackson FOLLOW

IDG News Service | May 21, 2012 4:18 PM PT

NASDAQ Stock Price Mismatch

Vulnerability Details : CVE-2011-0990

Race condition in the FastCopy optimization in the Array.Copy me buffer overflow and modify internal data structures, and cause a change after a type check but before a copy action.

Publish Date : 2011-04-13 Last Update Date : 2011-04-15

Security Vulnerability

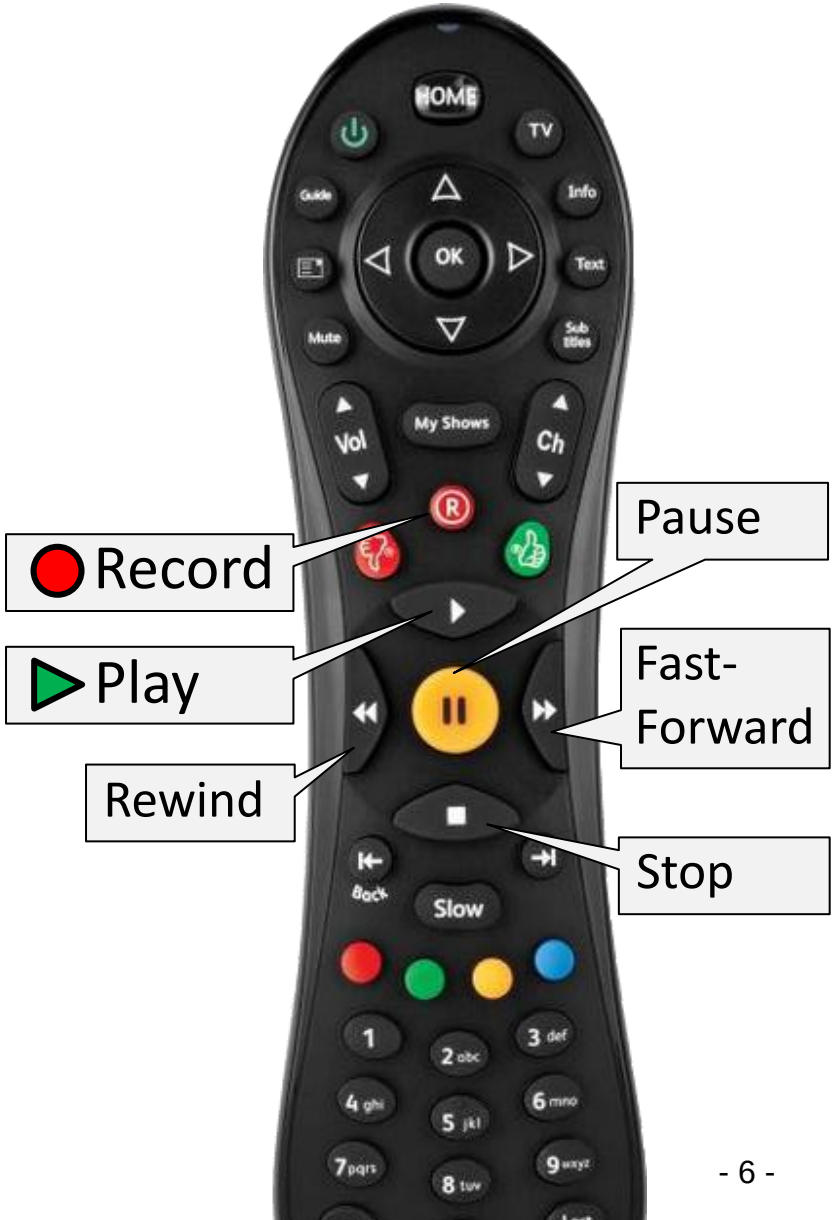
Multithreaded programming is HARD

Goal: to help programmers develop more reliable and secure multithreaded programs

Solutions:

- Deterministic replay → **DoublePlay**
- Data race detector → **TxRace**
- Systematic testing
- Program analysis
- …

DVR: Never Miss a Show



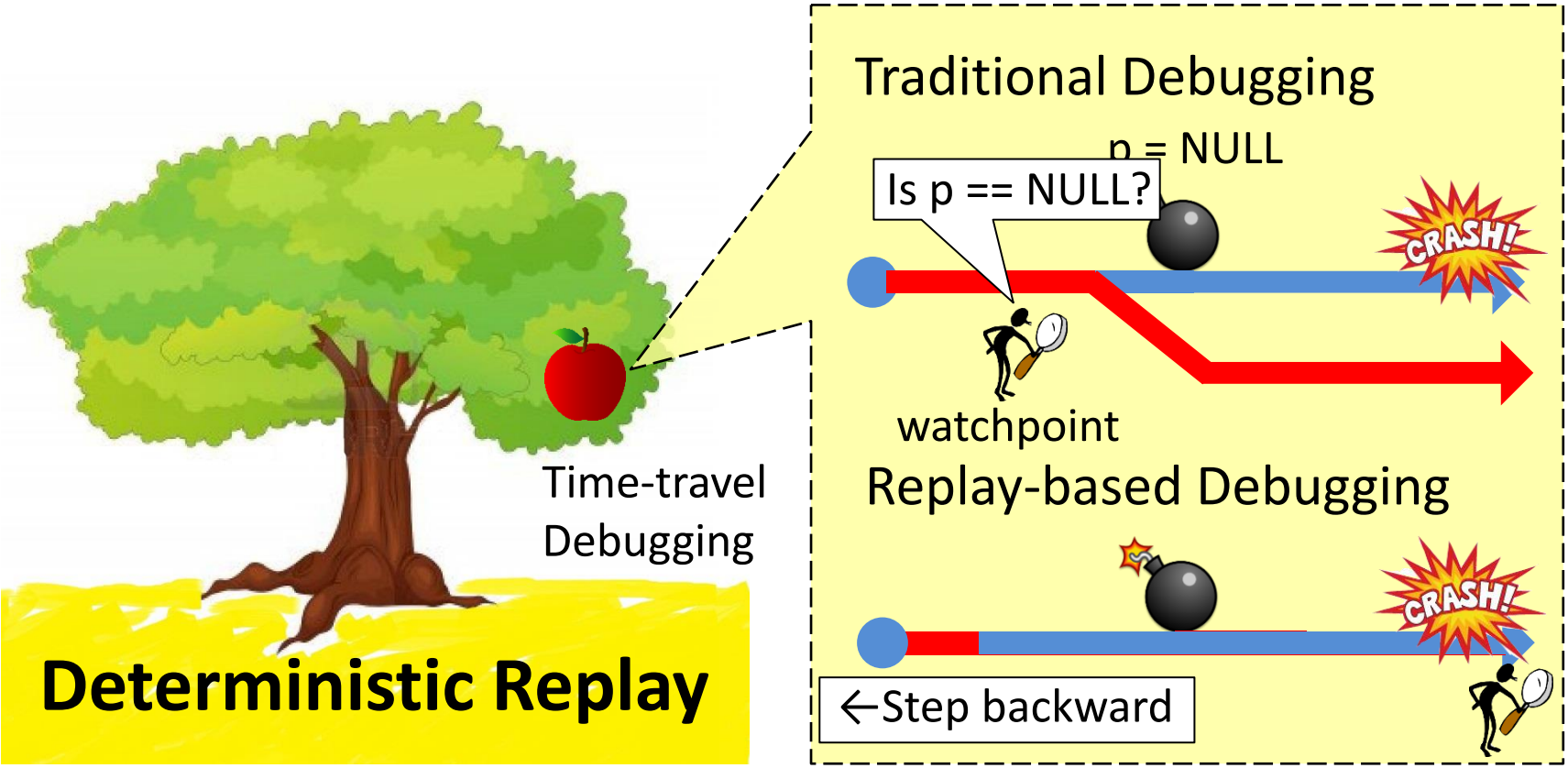
DVR - Control recorded TV

Deterministic Replay

- Record an execution
- Reproduce the same execution

≈

< DVR >

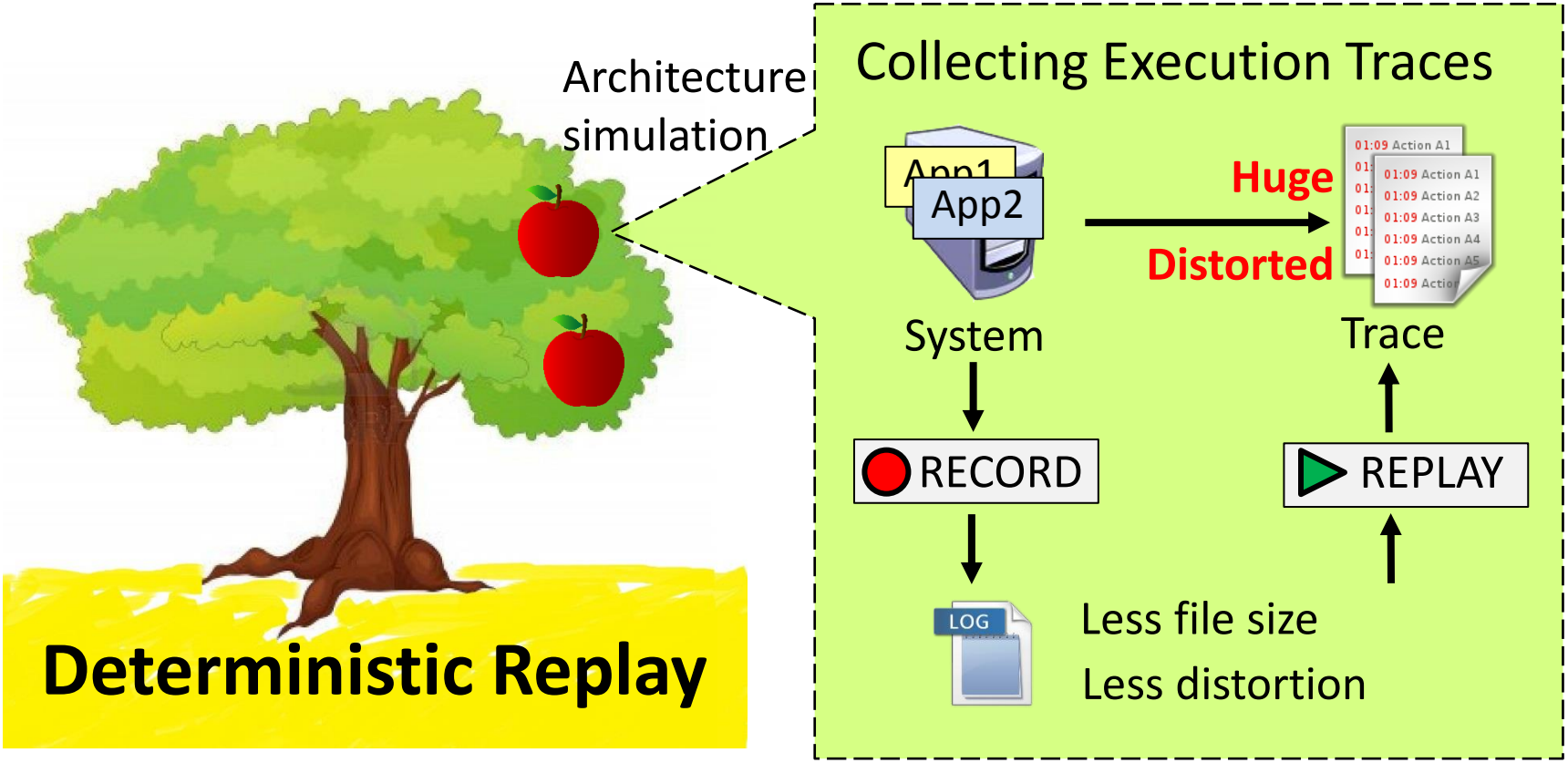


Deterministic Replay

- Record an execution
- Reproduce the same execution

≈

< DVR >



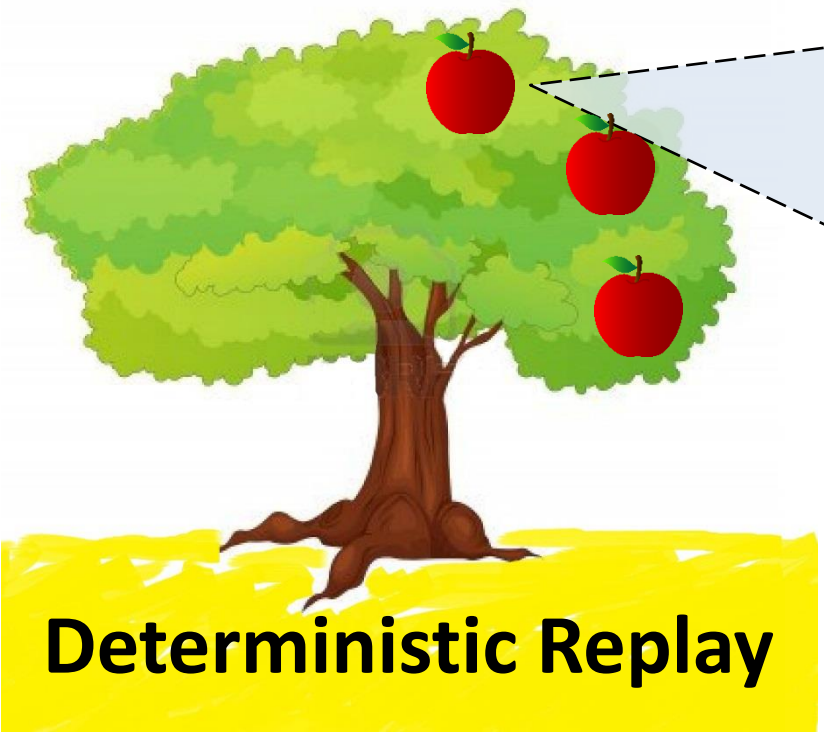
Deterministic Replay

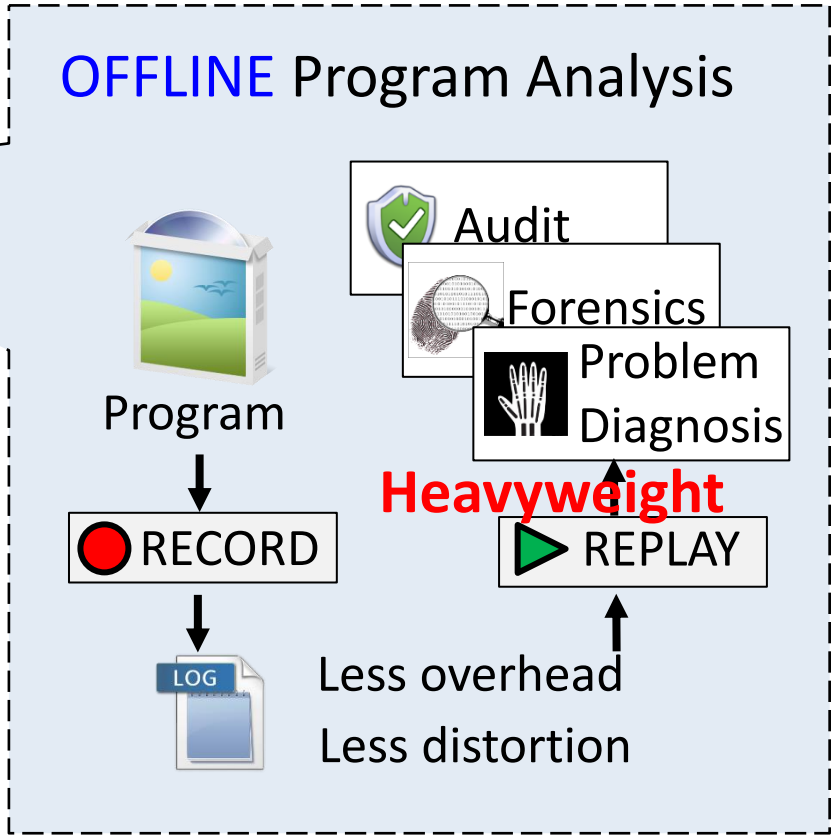
- Record an execution
- Reproduce the same execution

≈

< DVR >

Forensics, Auditing, …





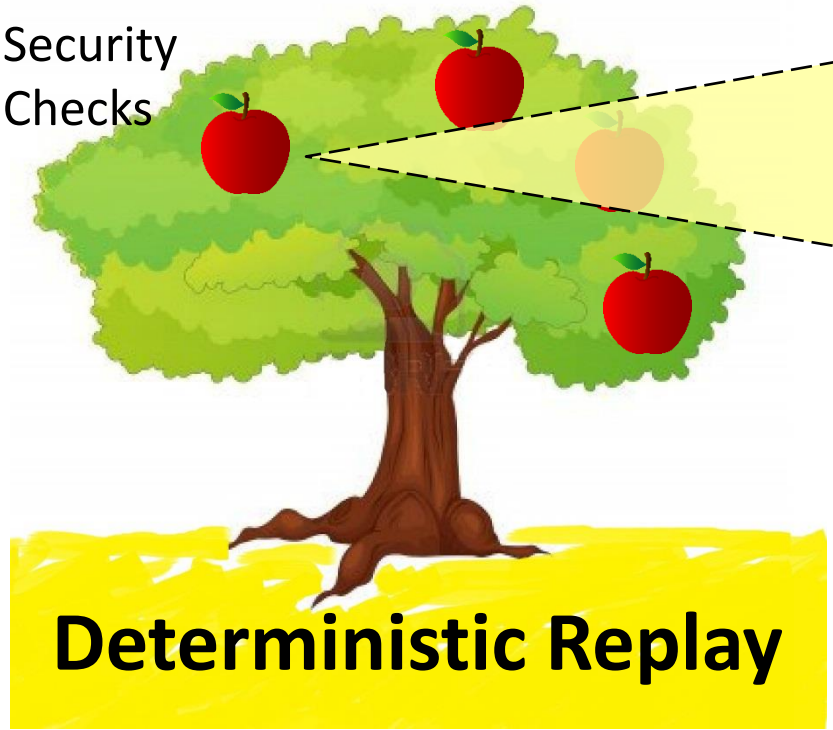
Deterministic Replay

- Record an execution
- Reproduce the same execution

≈

< DVR >

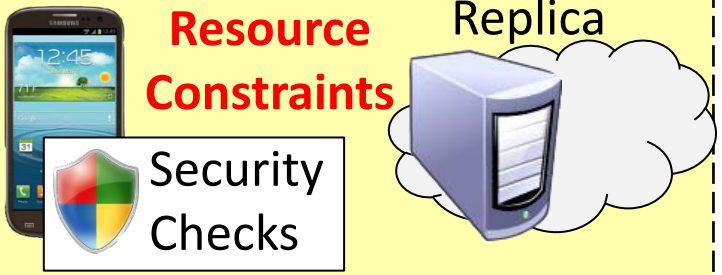
Offloaded Security Checks



ONLINE Program Analysis

Smartphone

Server-side Replica



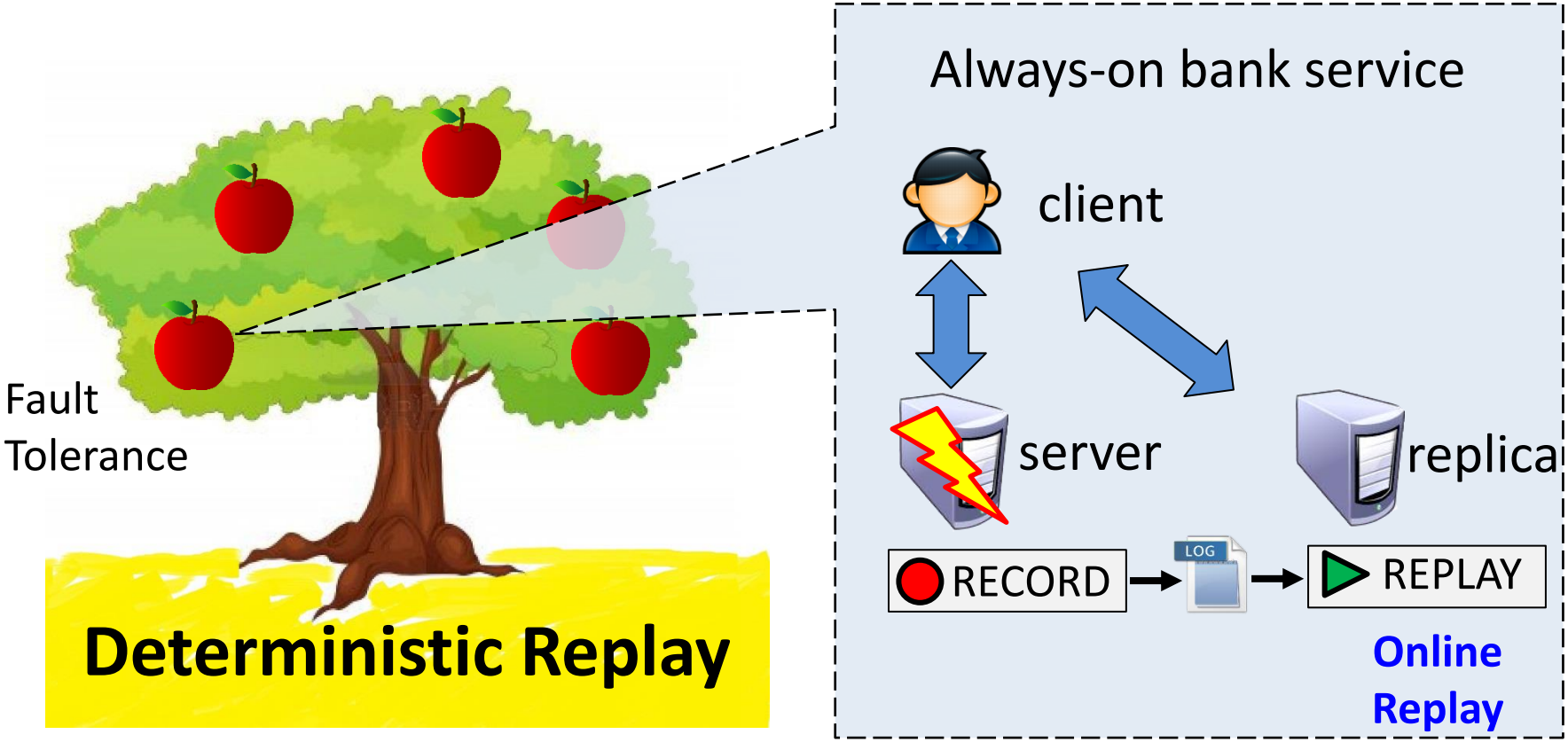
Online Replay

Deterministic Replay

- Record an execution
- Reproduce the same execution

≈

< DVR >

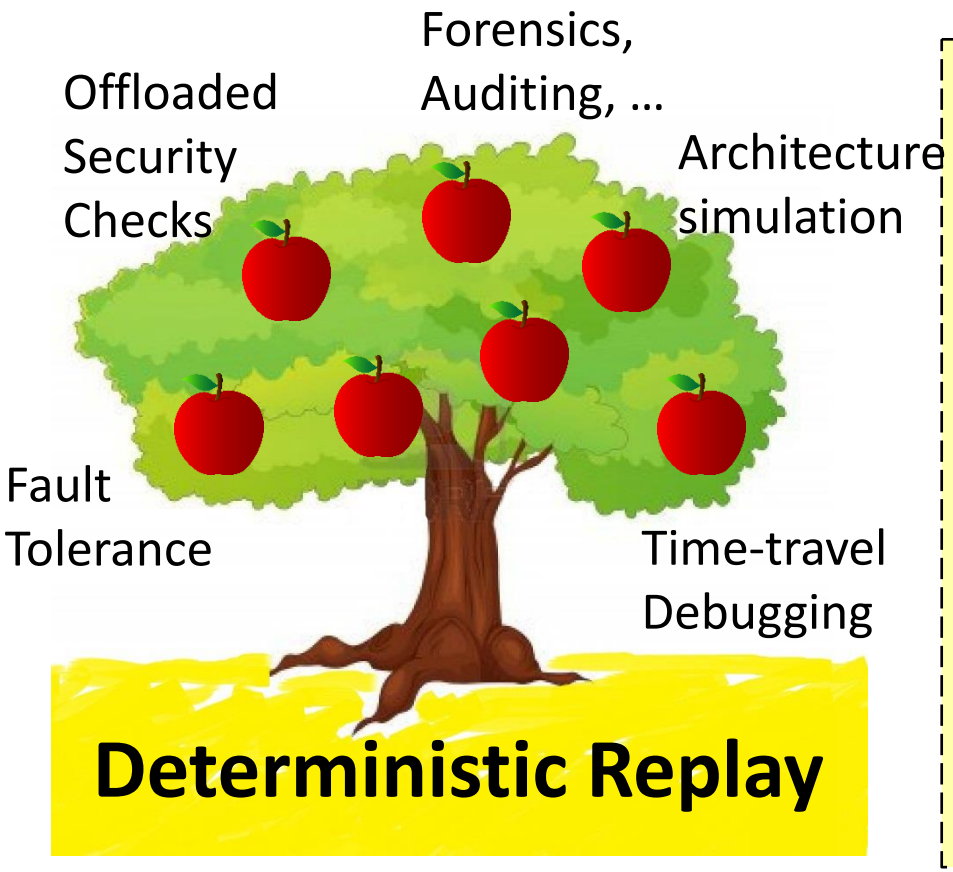


Deterministic Replay

- Record an execution
- Reproduce the same execution

≈

< DVR >



Uniprocessors

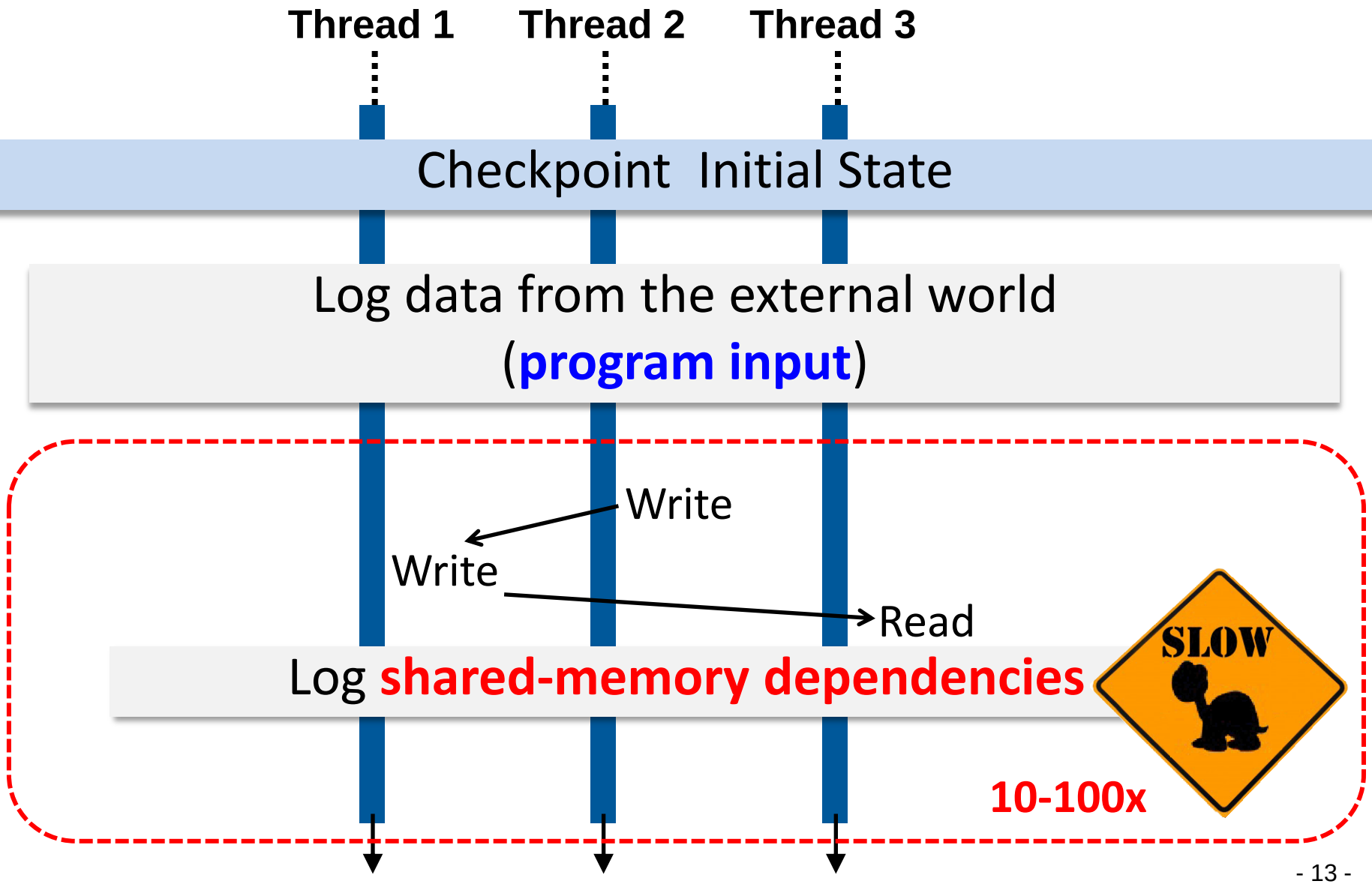
- At low overhead (~5%)
- Commercial products

SOLVED!

2004

Multiprocessors

Past Approach for Multiprocessor Replay



Contribution to Deterministic Replay

Log dependencies imprecisely,

but still provide useful replay guarantee

Replay Guarantee

Value Determinism

- Same sequence of instructions, reading/writing same value

Schedule Determinism

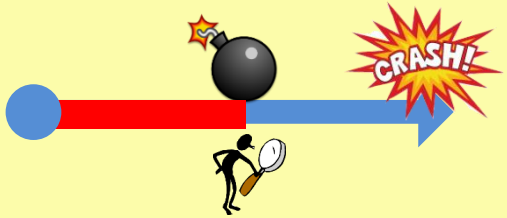
- Same thread interleaving

Past solutions log precise shared-memory dependencies

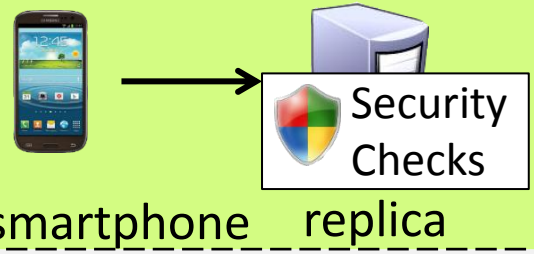
External Determinism

- Same output and final state

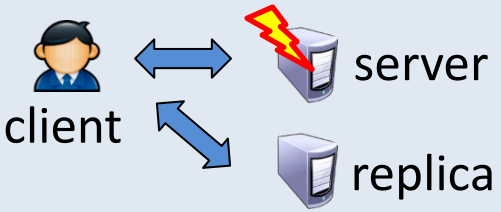
Debugging



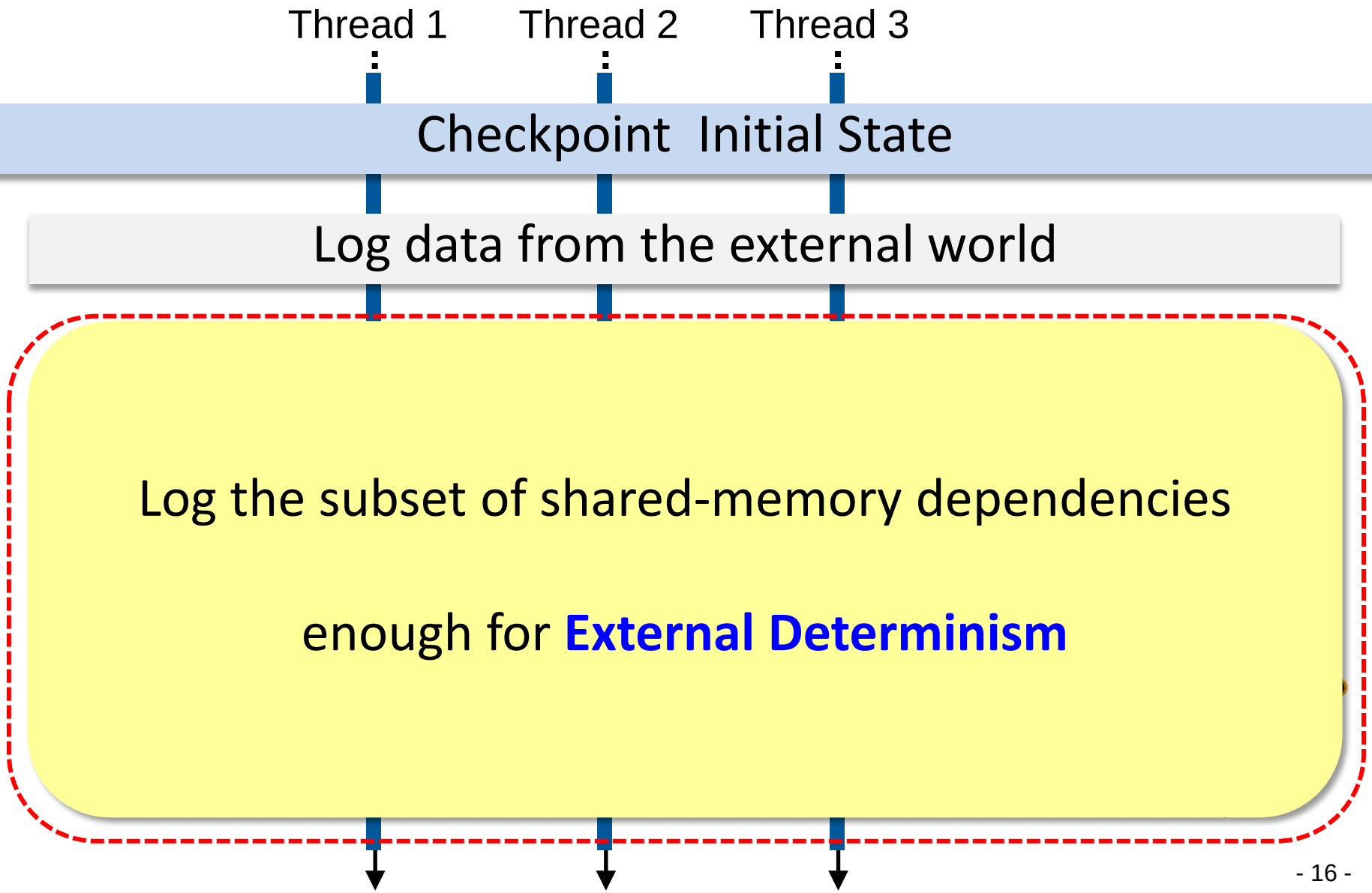
Offloaded analysis



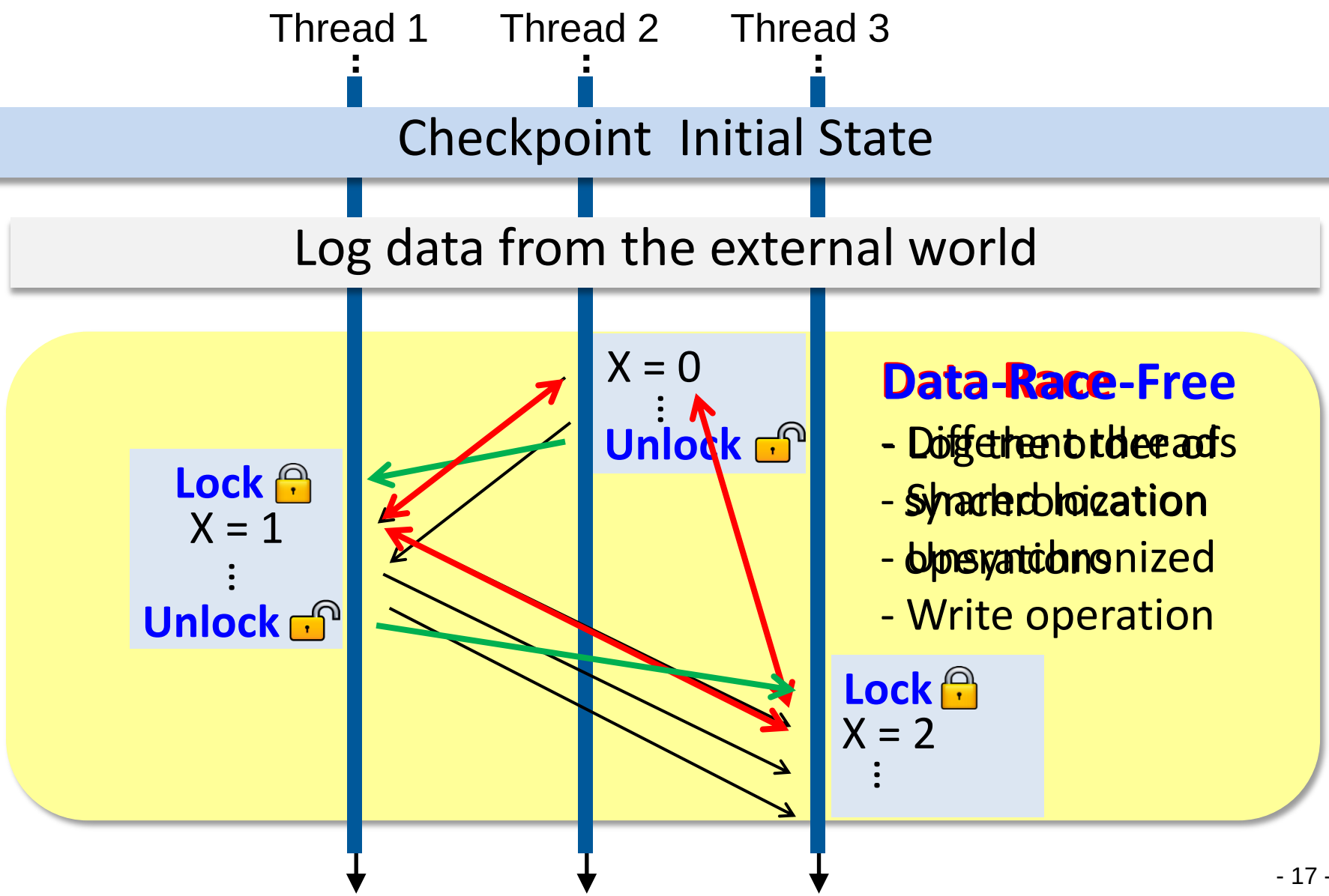
Fault tolerance



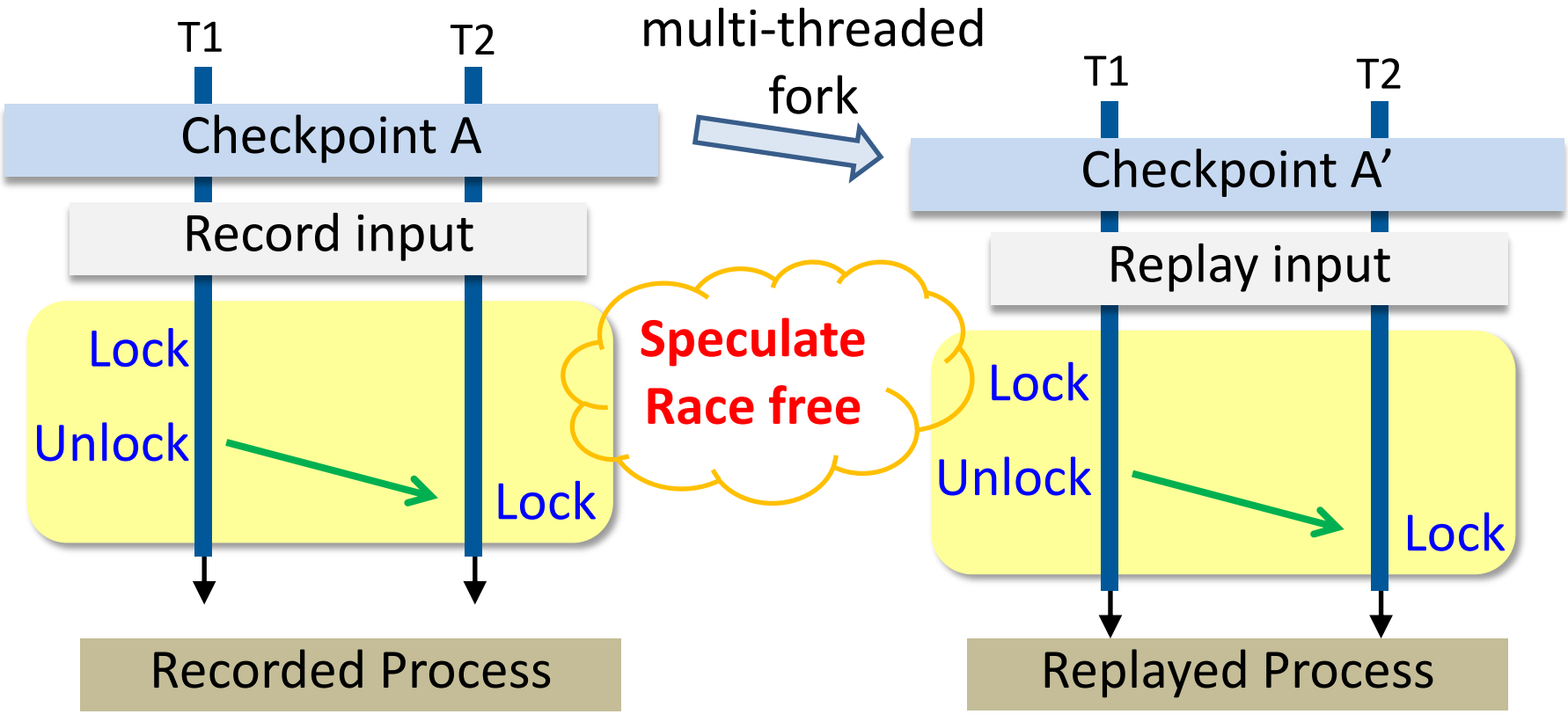
Overview of DoublePlay



Recording for Data-Race-Free Programs



Online Replay with Speculation

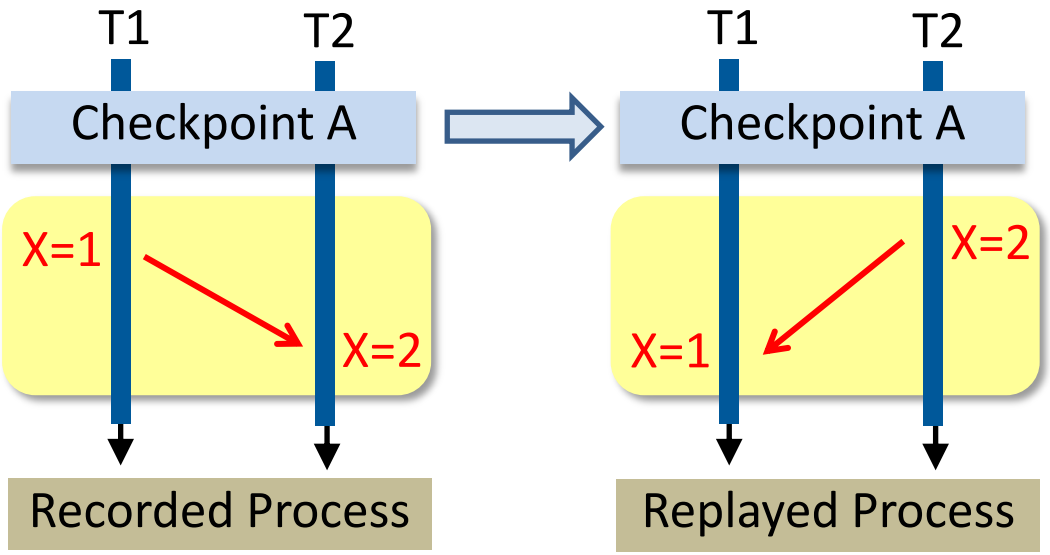


- Record and replay program input
- Record and replay the order of synchronizations
  - Speculate data-race-free programs

What if a program is NOT race free?

Problem

- Mis-speculation
- Data race detector?



Insight: External Determinism is sufficient

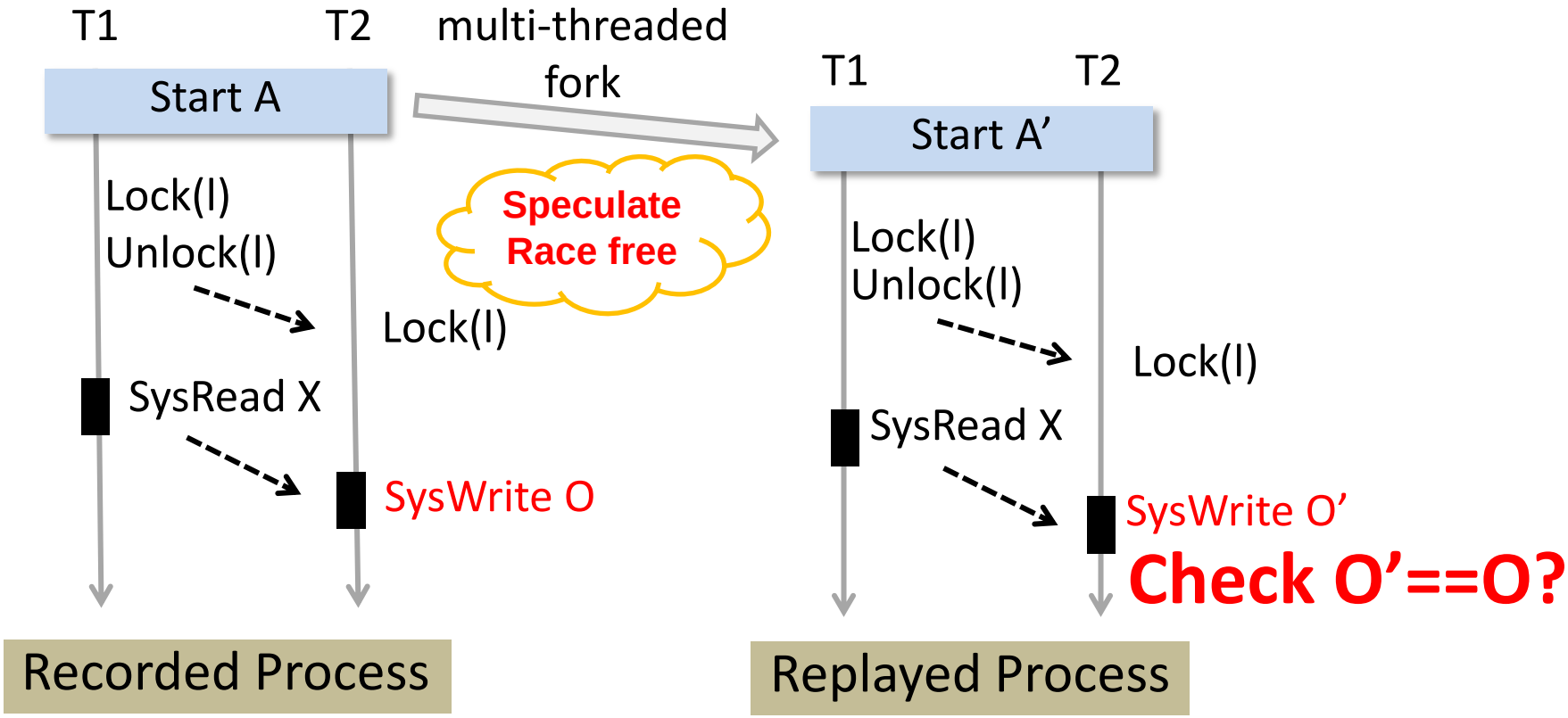
- Guarantee the same **output** and **final state**

Solution

- Detect mis-speculation when the replay is not externally deterministic

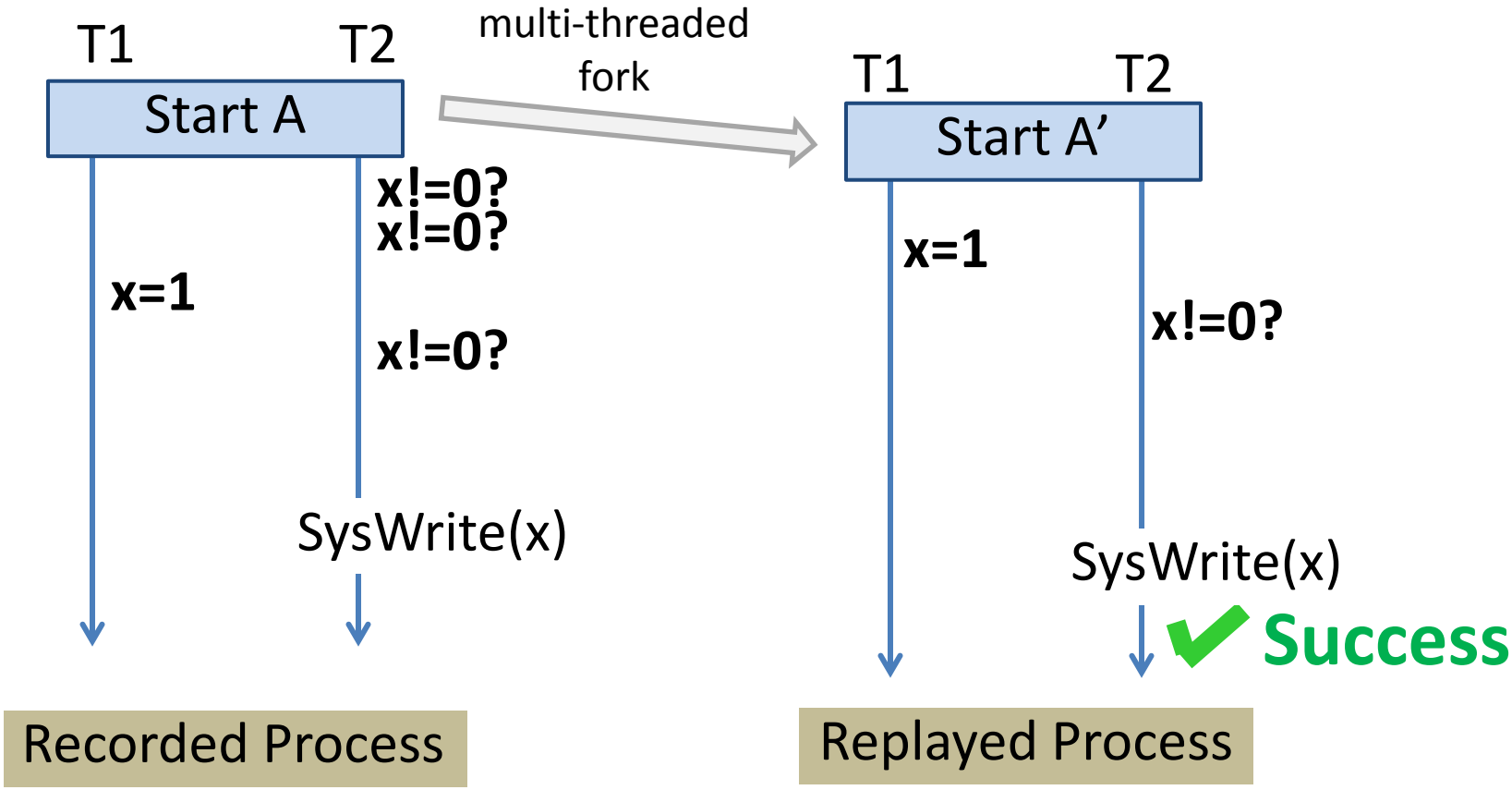
Check #1 – System Output

- Compare system call arguments
- Guarantee external determinism

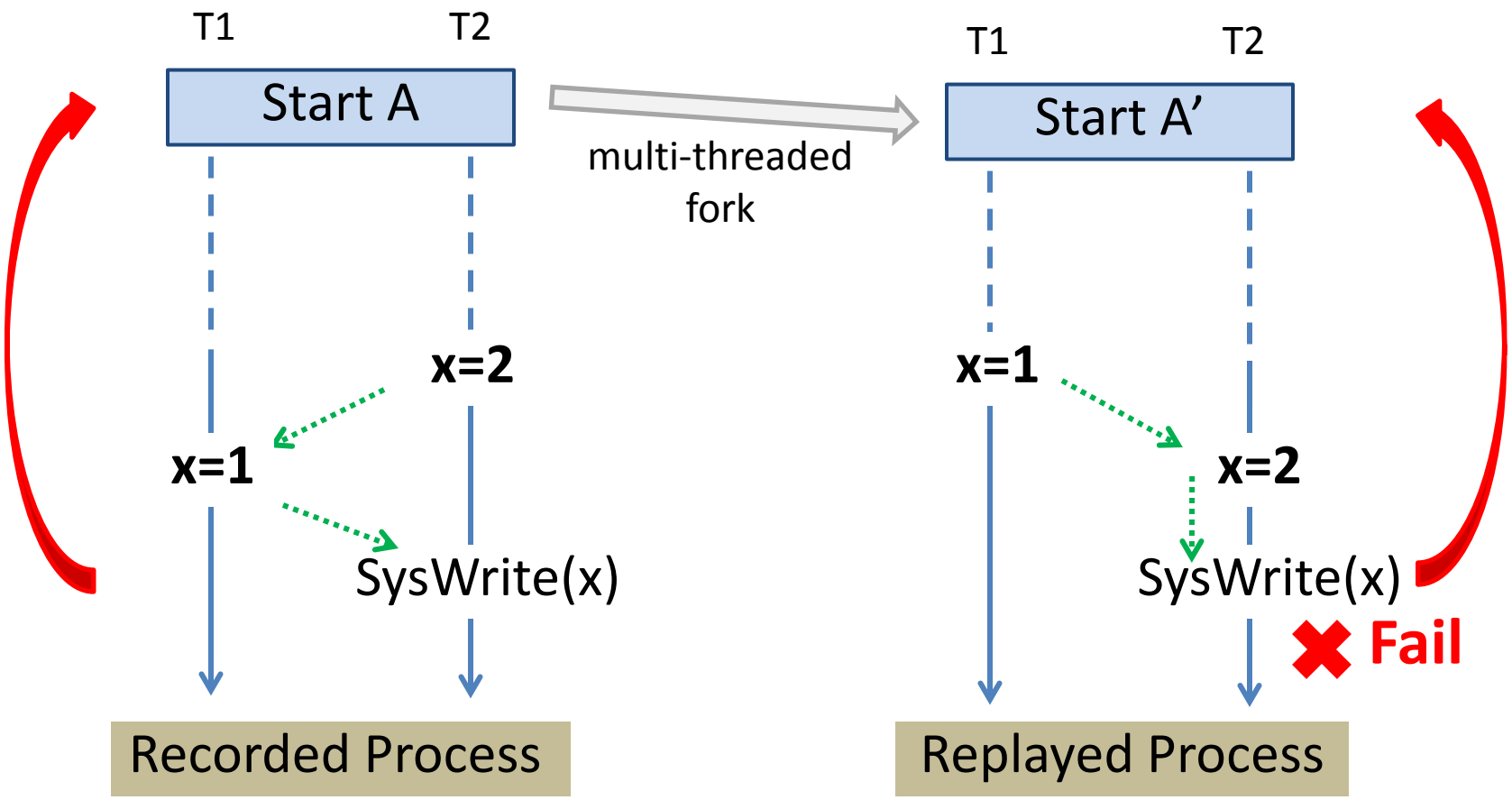


Inconsequential Data Races

- Not all races cause external determinism checks to fail



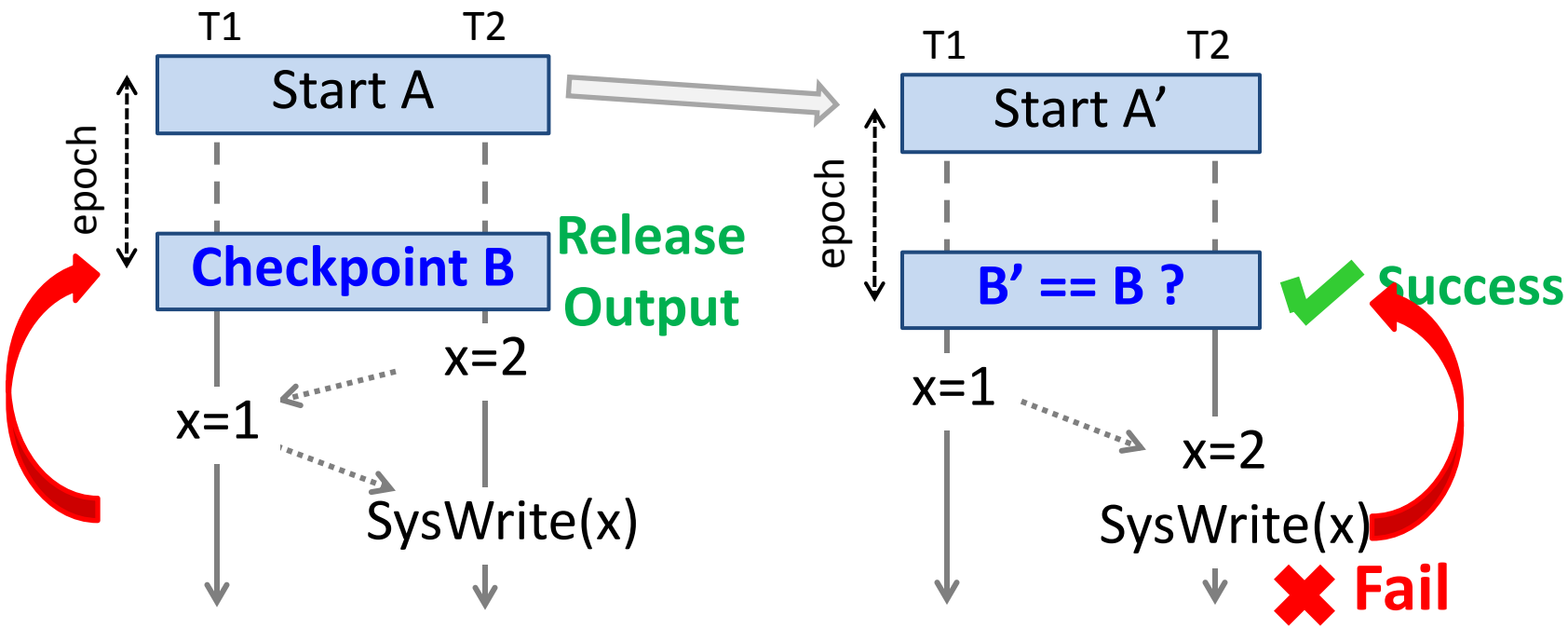
Divergence due to Data Races



1) Need to rollback to the beginning
2) Need to buffer system output till the end

Check #2 – Program State

- Create a checkpoint periodically
- Compare memory and register states

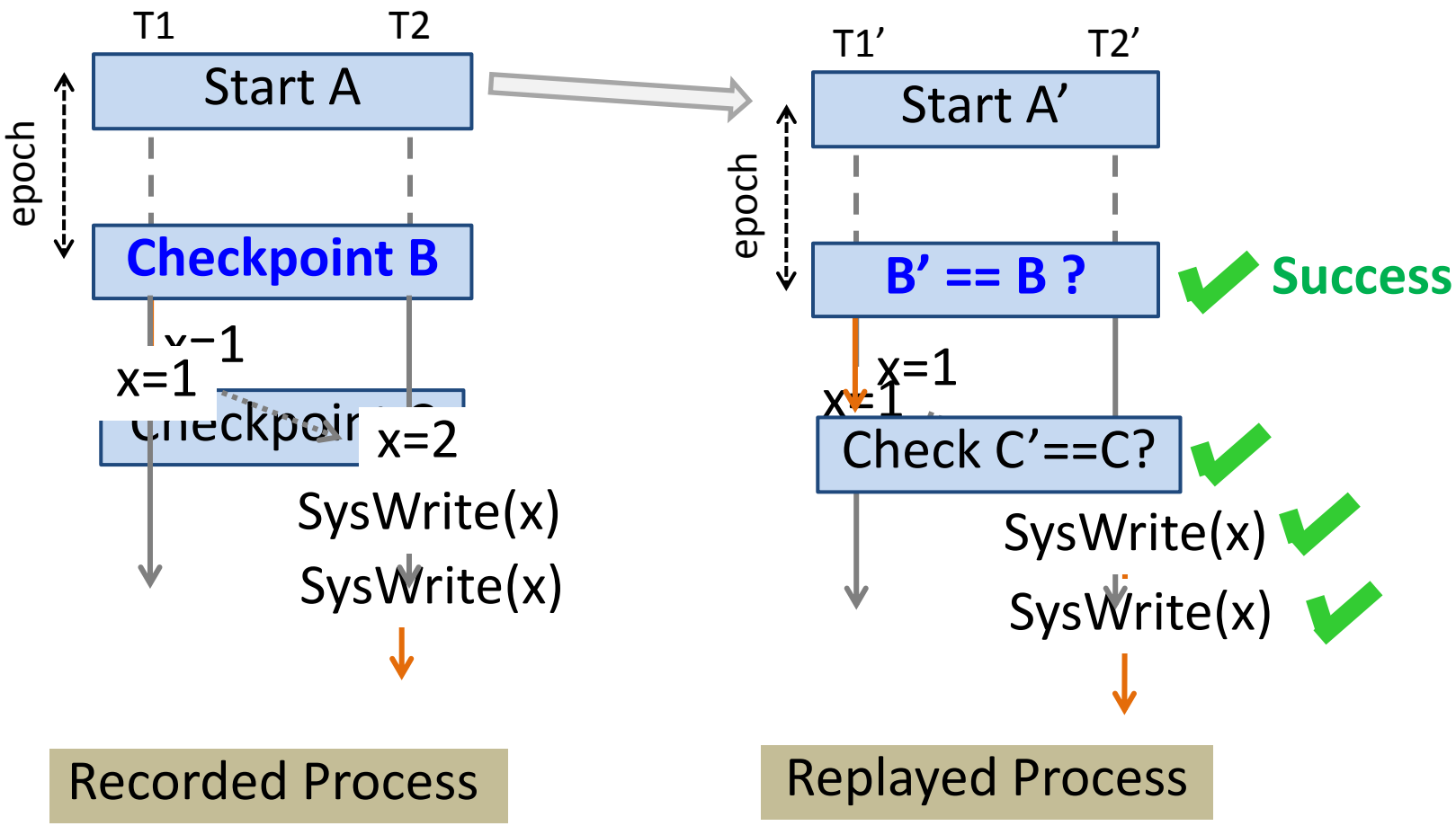


Recorded Process

Replayed Process

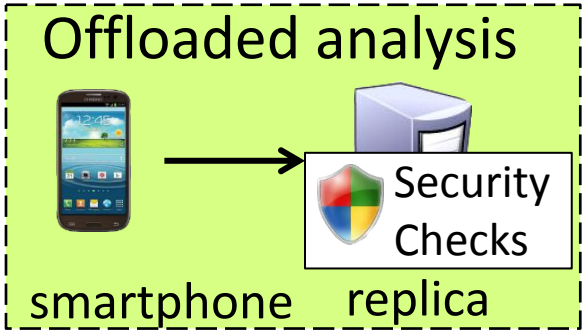
Recovery via Rollback and Re-execute

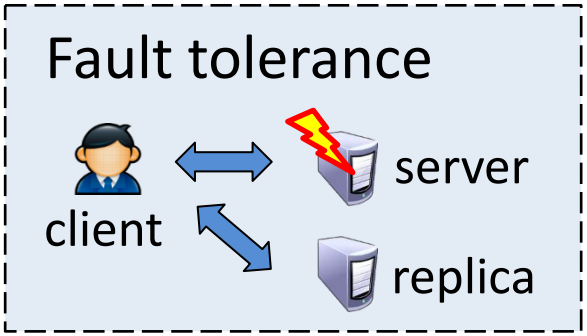
- Optimistically re-run the failed epoch
- On repeated failure, record and replay only one thread at a time



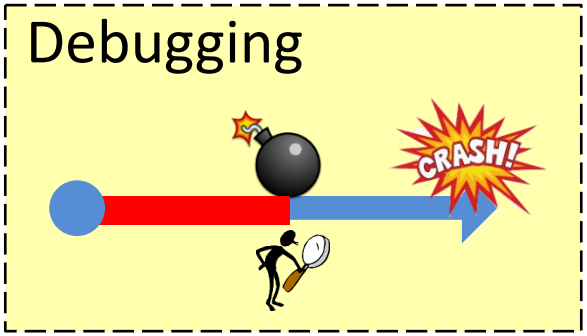
Online Replay vs. Offline Replay

- The system described so far supports online replay
  - The original run should be alive
  - This is sufficient for

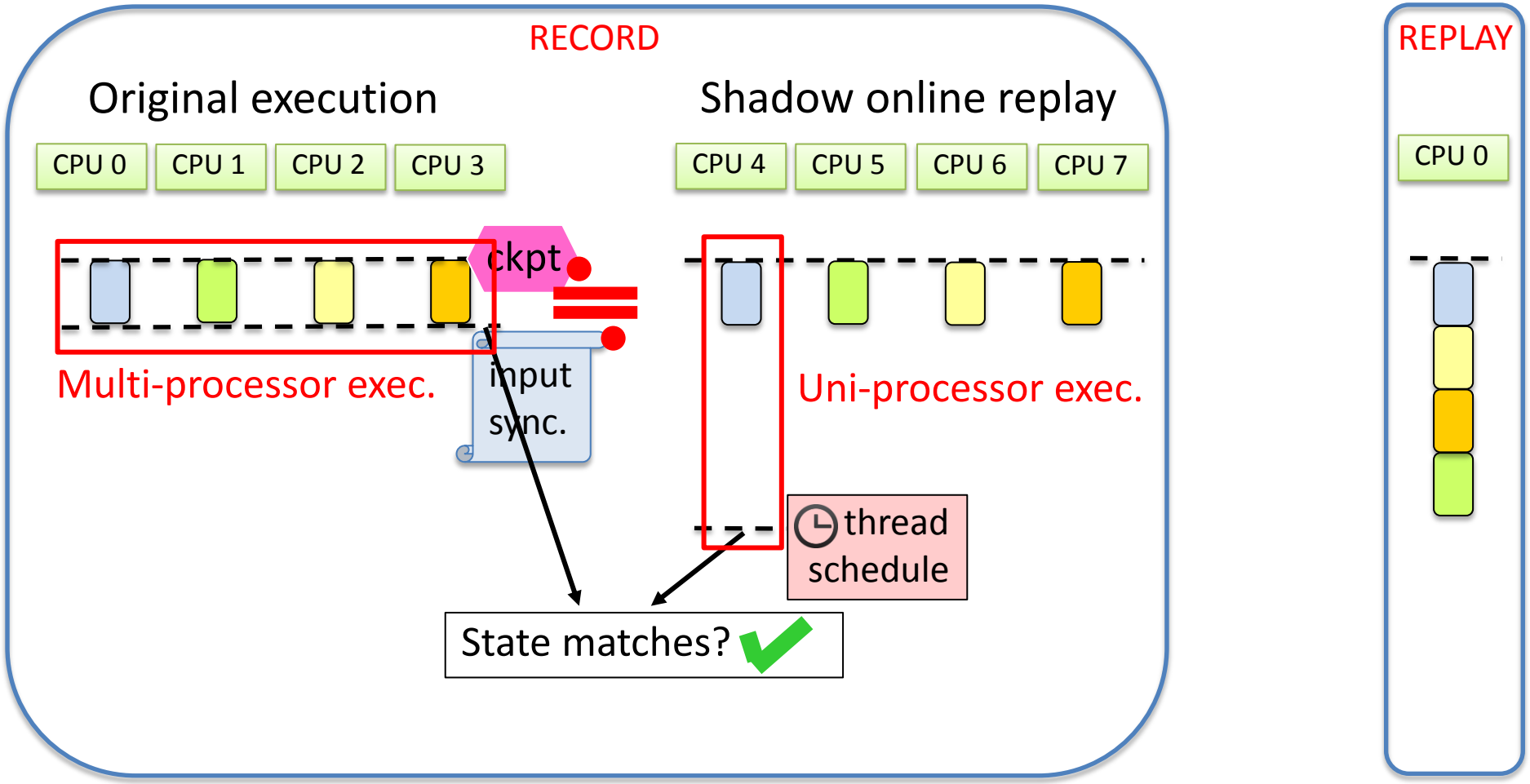




- Not for debugging which requires offline replay

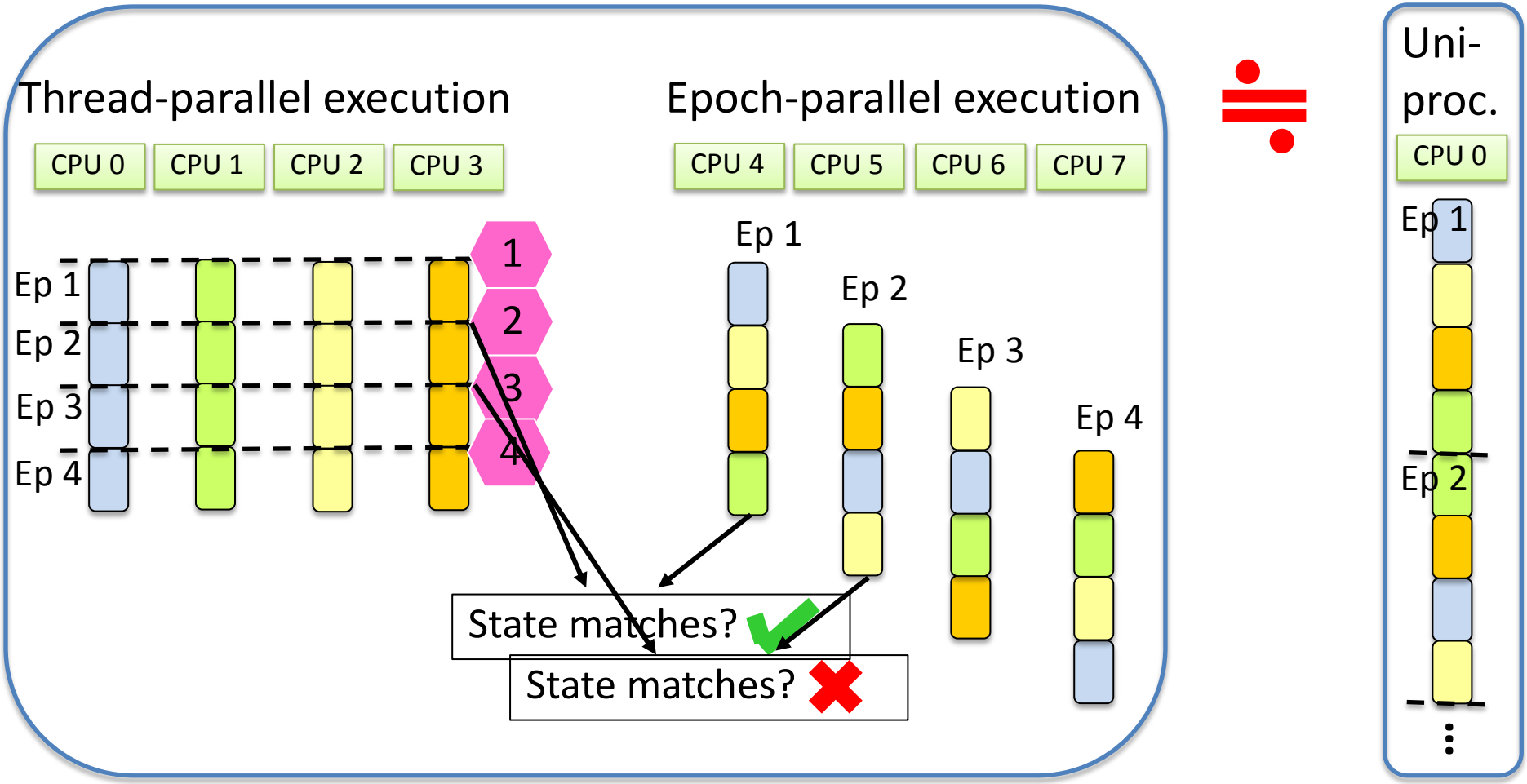


Recording Thread Schedule for Offline Replay



- Logging thread schedule is sufficient for deterministic replay
- Need to wait until the online replay finishes (4x slow)

DoublePlay: Uniparallel Execution



- Provide the benefits (guarantees) of uniprocessor executions
- Provide the performance of multiprocessor executions

Implementation and Evaluation

Implementation

- Modified Linux kernel and glibc library

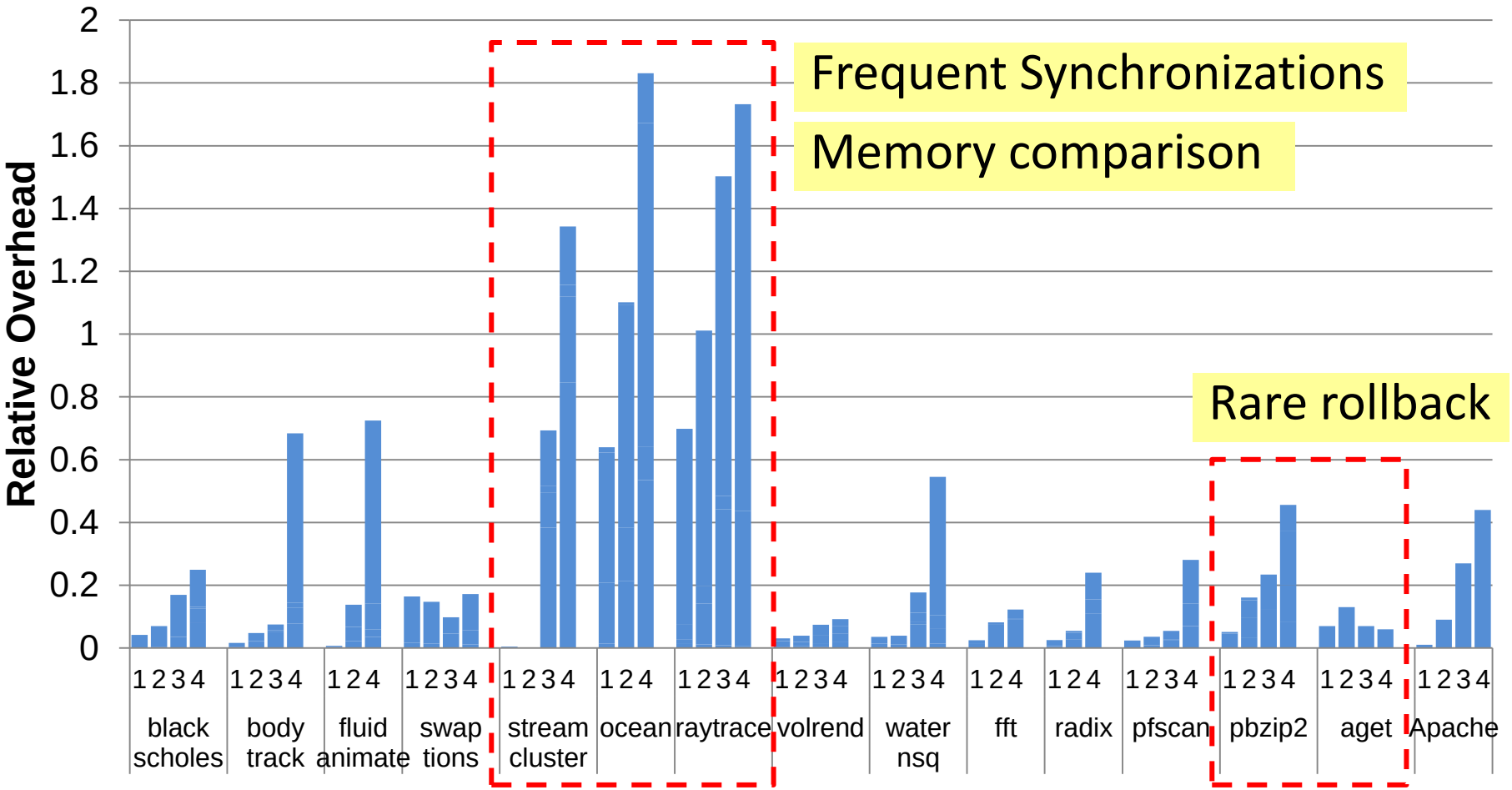
Test Environment

- 2 GHz 8 core Xeon processor with 3 GB of RAM
- Run 1~4 worker threads (excluding control threads)

Benchmarks

- PARSEC suite
  - blackscholes, bodytrack, fluidanimate, swaptions, streamcluster
- SPLASH-2 suite
  - ocean, raytrace, volrend, water-nsq, fft, and radix
- Desktop/Server applications
  - pbzip2, pfscan, aget, and Apache

Record and Replay Performance



- **18%** for 2 threads, **55%** for 4 threads

Roadmap

- Motivation
- DoublePlay: Deterministic Replay
- **TxRace**: Data Race Detector
- Conclusion

State-Of-The-Art Dynamic Data Race Detector

Software based solutions

- FastTrack [PLDI'09]
- Intel Inspector XE
- Google Thread Sanitizer
- …

✔ Sound (no false negatives)
✔ Complete (no false positives)
✗ High overhead (10-100x)

Hardware based solutions

- ReEnact [ISCA'03]
- CORD [HPCA'06]
- SigRace [ISCA'09]
- …

✔ Low overhead
✗ Custom hardware

Overview of TxRace

- Hybrid SW + (existing) HW solution

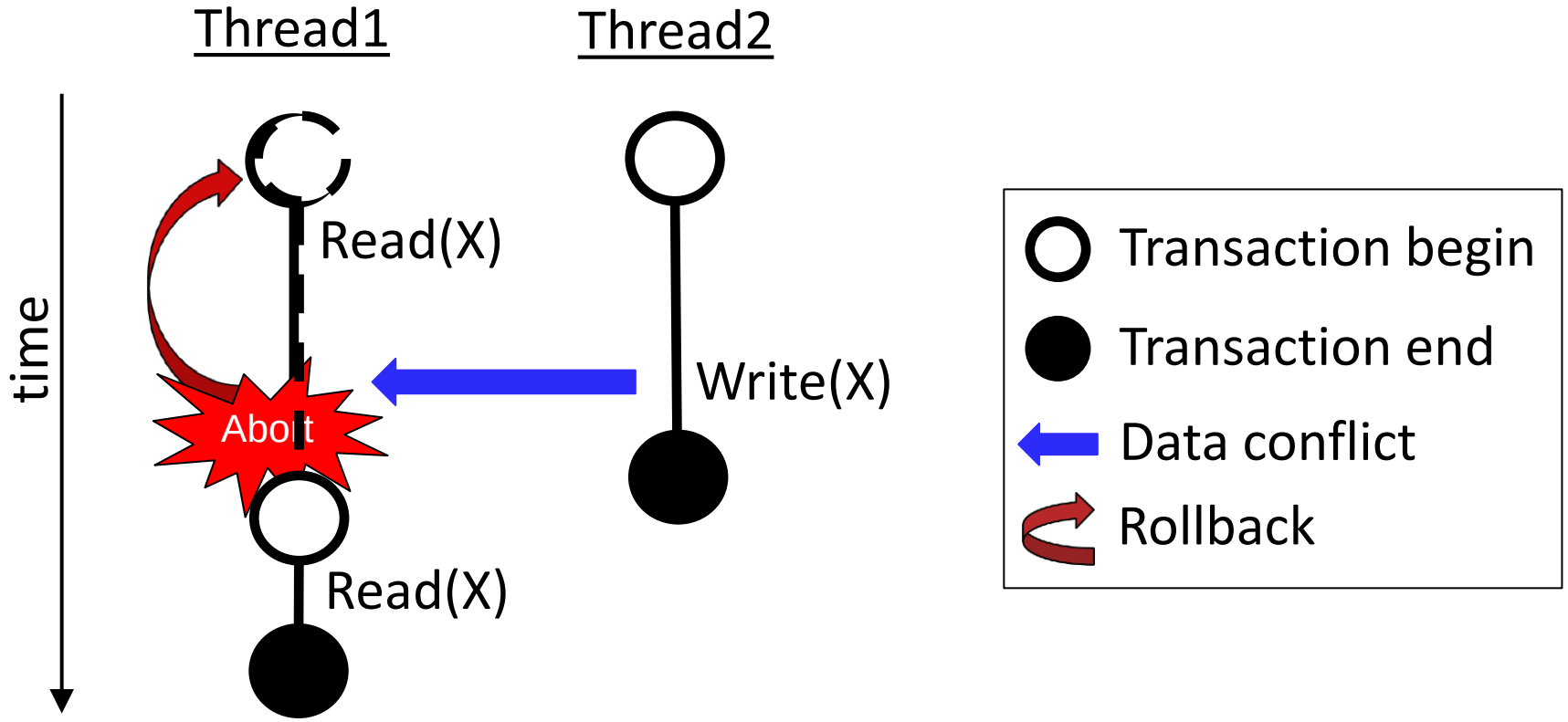
- Leverage the data conflict detection mechanism of Hardware Transactional Memory (HTM) in commodity processors for lightweight data race detection

  ✔ Low overhead

  ✔ No custom hardware

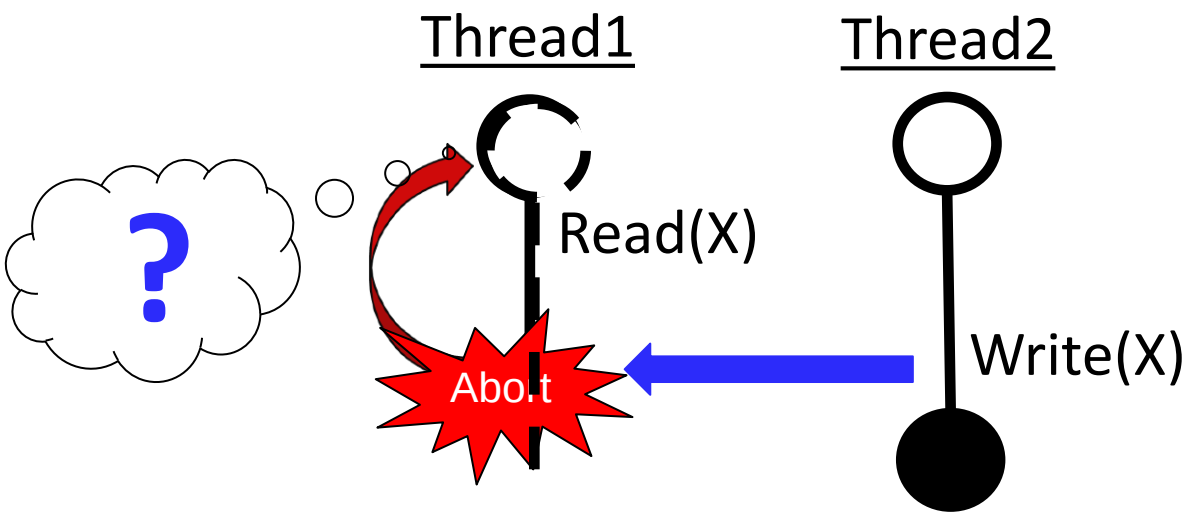
Background: Transactional Memory (TM)

- Allow a group of instructions (a transaction) to execute in an atomic manner



Challenge 1: Unable to Pinpoint Racy Instructions

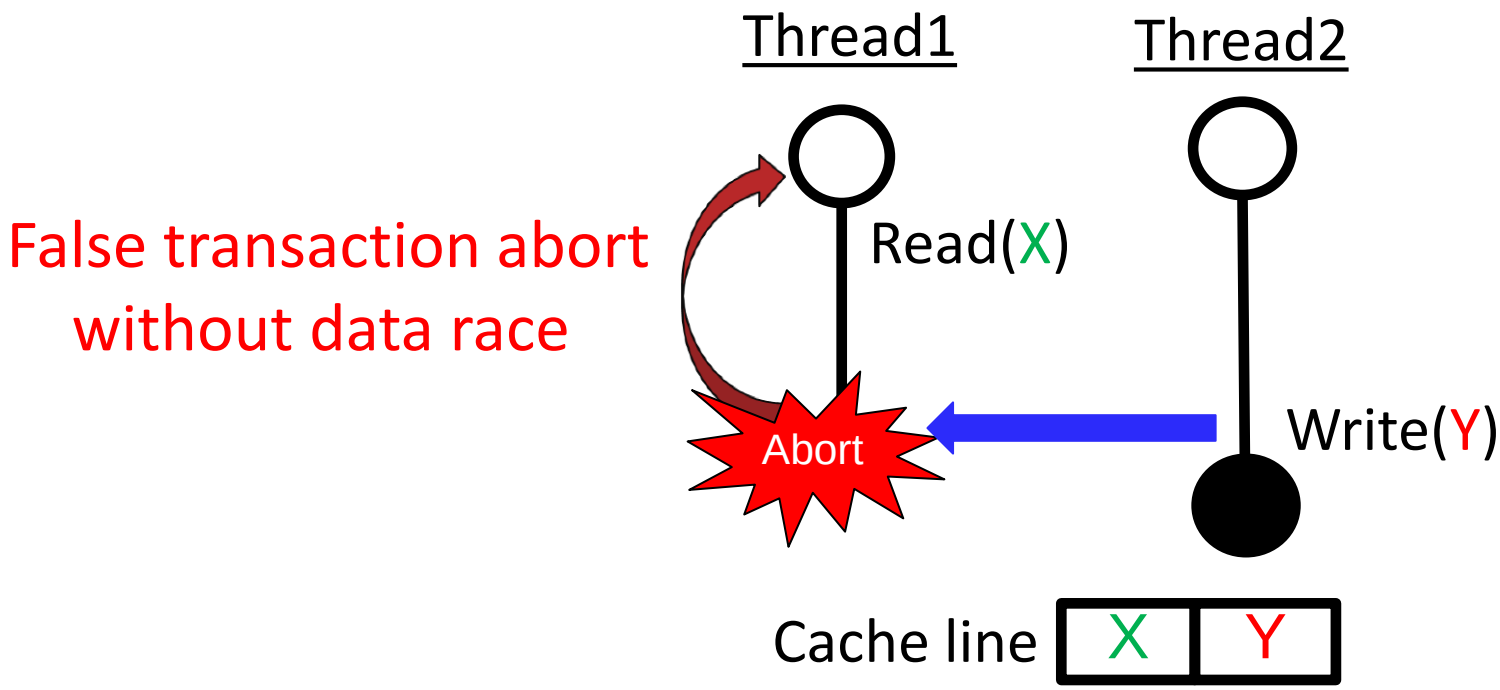
- When a transaction gets aborted, we know that there was a data conflict between transactions



- However, we DO NOT know WHY and WHERE
  - e.g. which instruction? at which address? Which transaction caused the conflict?

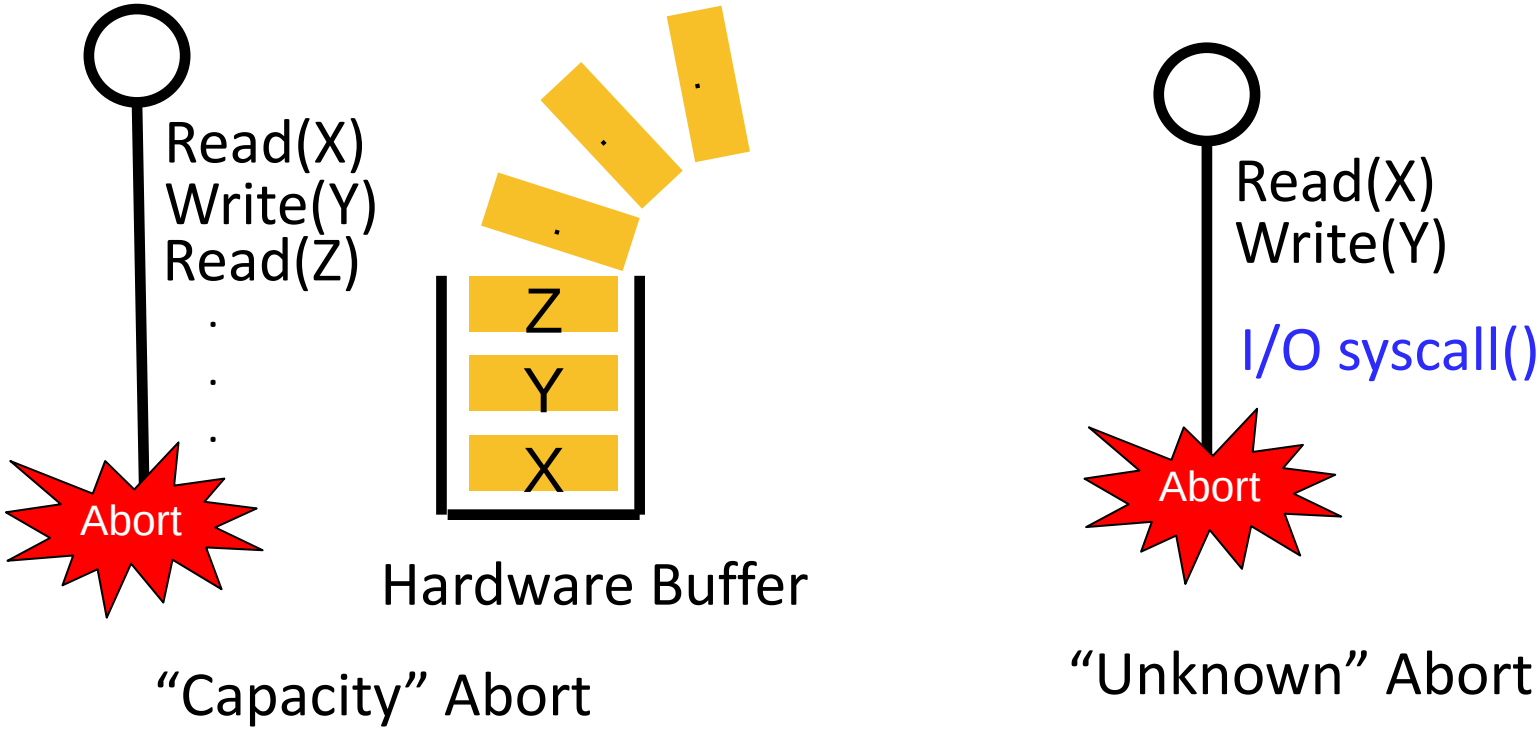
Challenge 2: False Conflicts → False Positives

- HTM detects data conflicts at the cache-line granularity → False positives

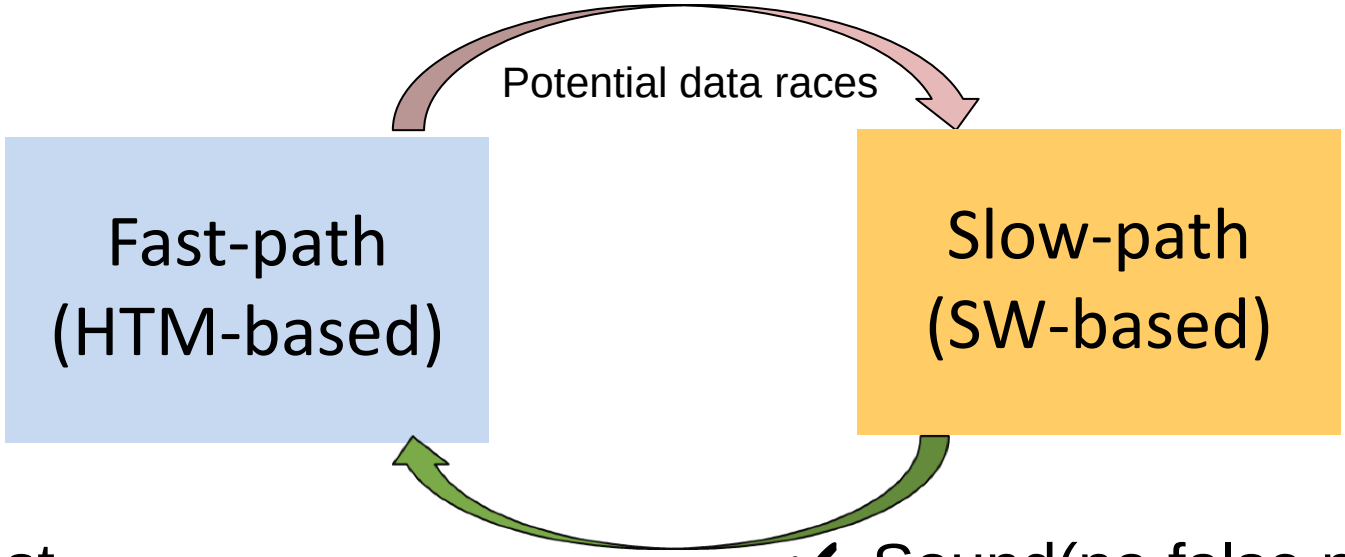


Challenge 3. Non-conflict Aborts

- Best-effort (non-ideal) HTM with limitations
  - → Transactions may get aborted without data conflicts
  - → False negatives (if ignored)



TxRace: Two-phase Data Race Detection



✔ Fast

✗ Unable to pinpoint races

✗ False sharing(false positive)

✗ Non-conflict aborts(false negative)

Intel Haswell
(RTM)

✔ Sound(no false negative)

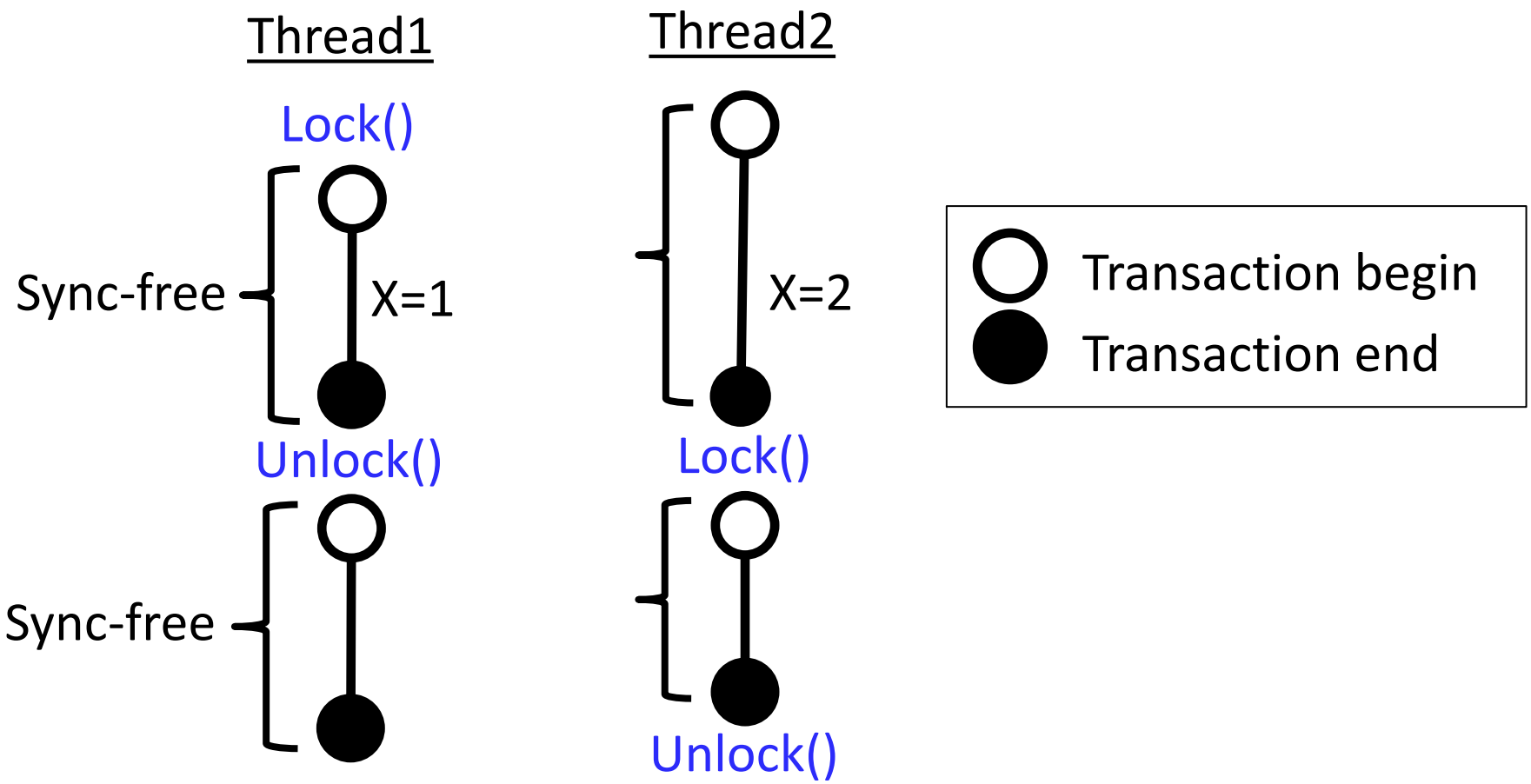
✔ Complete(no false positive)

✗ Slow

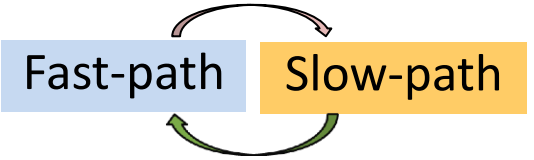
Google ThreadSanitizer
(TSan)

Compile-time Instrumentation

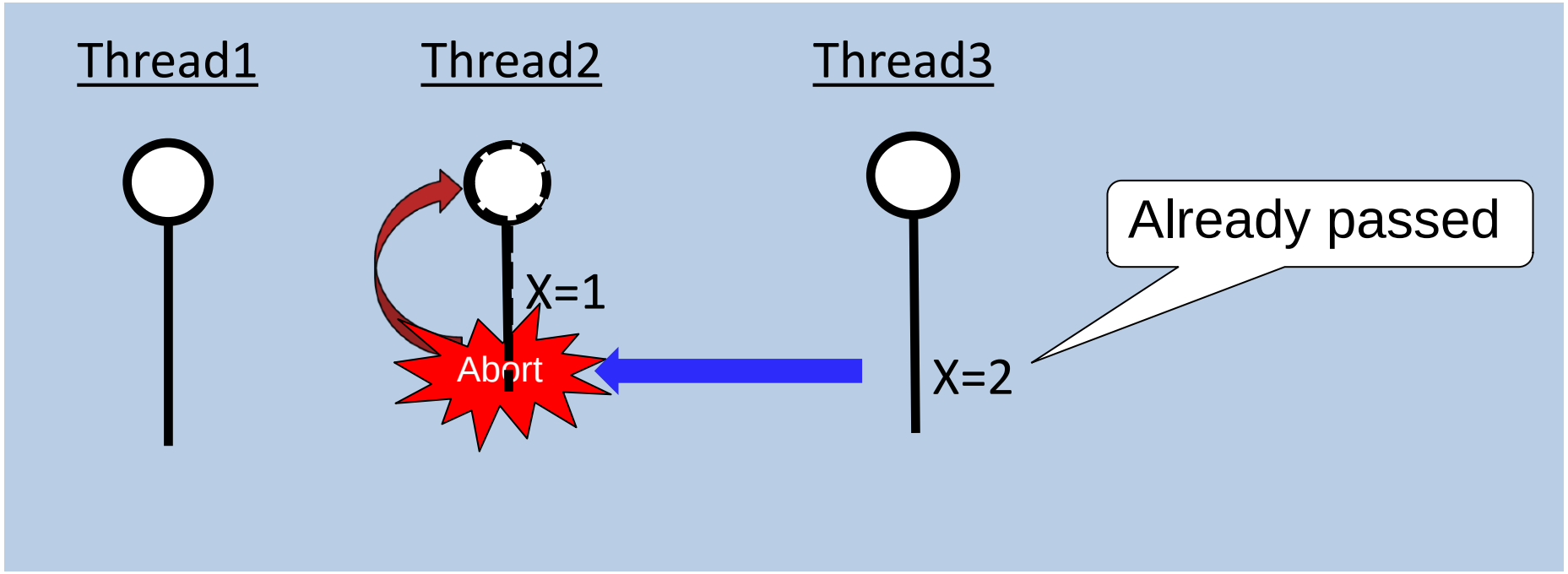
- Fast-path: convert sync-free regions into transactions
- Slow-path: add Google TSan checks



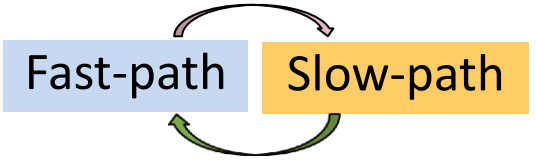
Fast-path HTM-based Detection



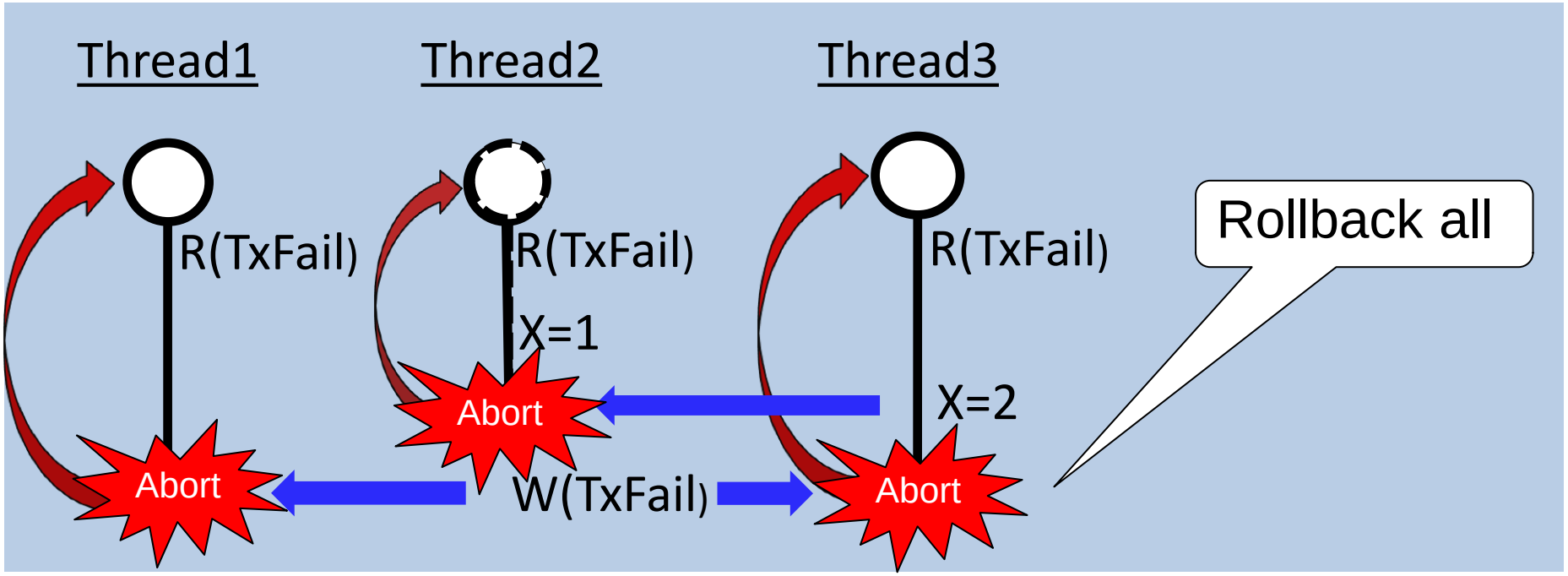
- Leverage HW-based data conflict detection in HTM
- **Problem**: On conflict, one transaction gets aborted, but all others just proceed → slow-path missed racy transactions



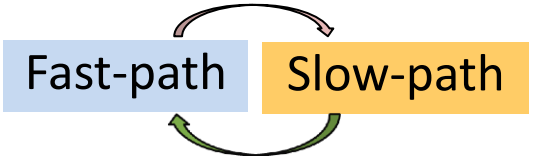
Fast-path HTM-based Detection



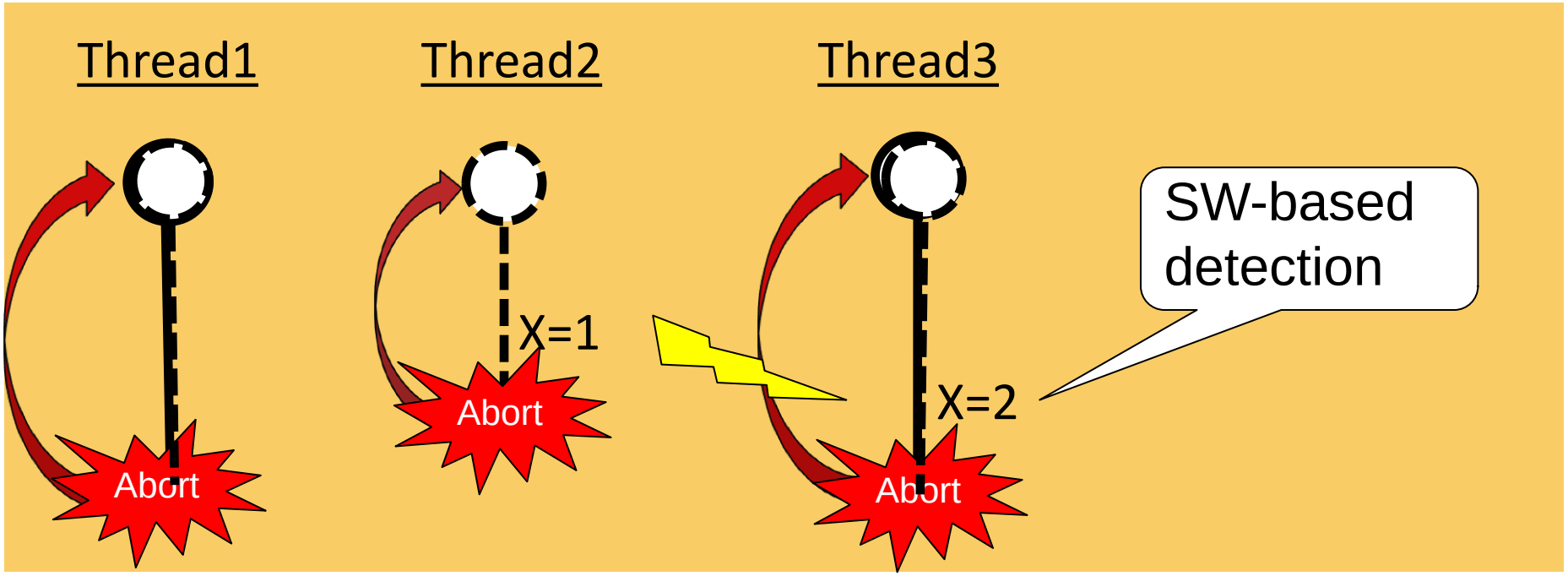
- Leverage HW-based data conflict detection in HTM
- **Problem**: On conflict, one transaction gets aborted, but all others just proceed → slow-path missed racy transactions
- **Solution**: Abort in-flight transactions artificially



Slow-path SW-based Detection



- Use SW-based sound and complete data race detection
  - Pinpoint racy instructions
  - Filter out false positives (due to false sharing)
  - Handle non-conflict (e.g., capacity) aborts conservatively



Implementation

Two-phase data race detection

- Fast-path: Intel's Haswell Processor
- Slow-path: Google's Thread Sanitizer

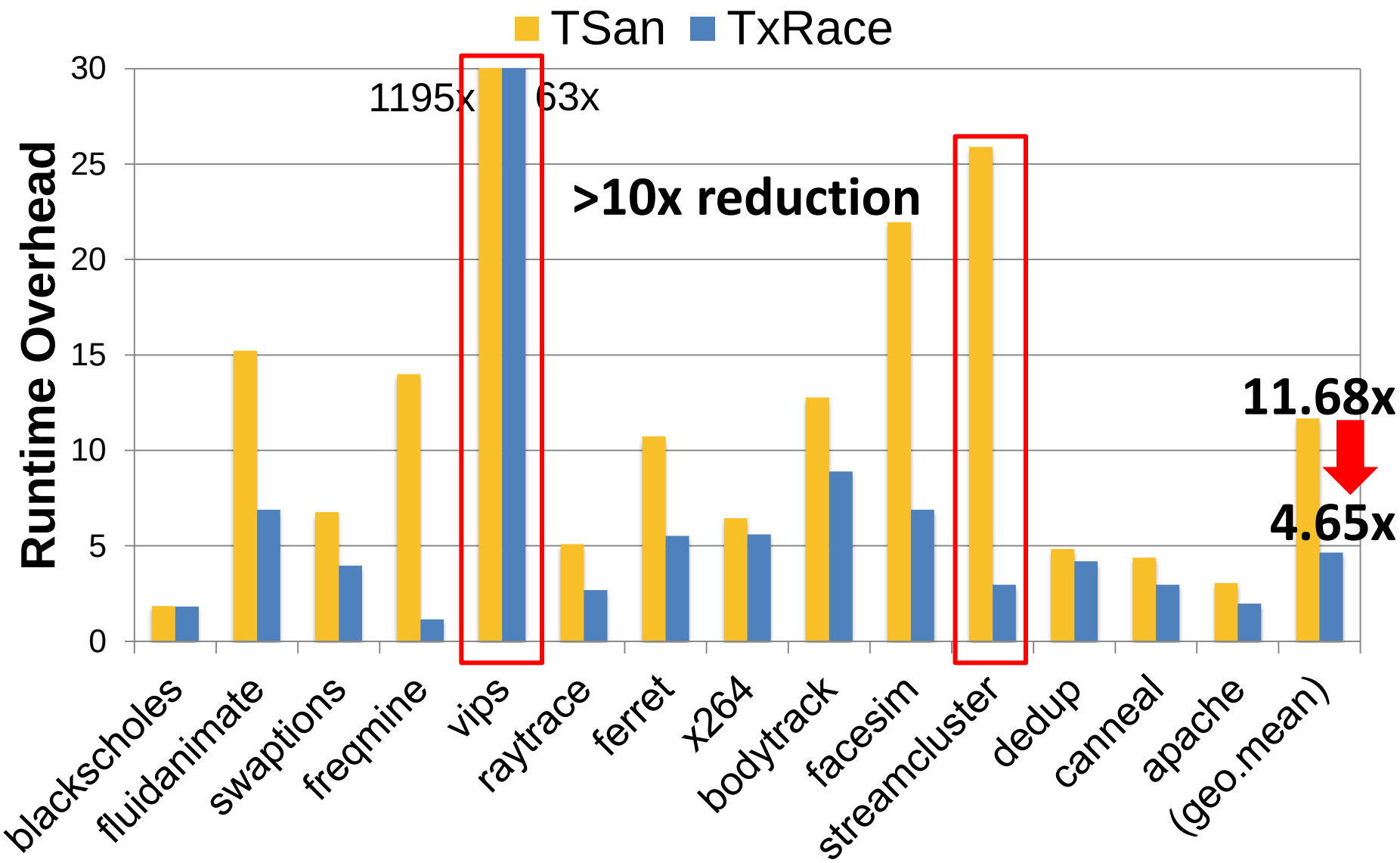
Instrumentation

- LLVM compiler framework
- Compile-time & profile-guided optimizations

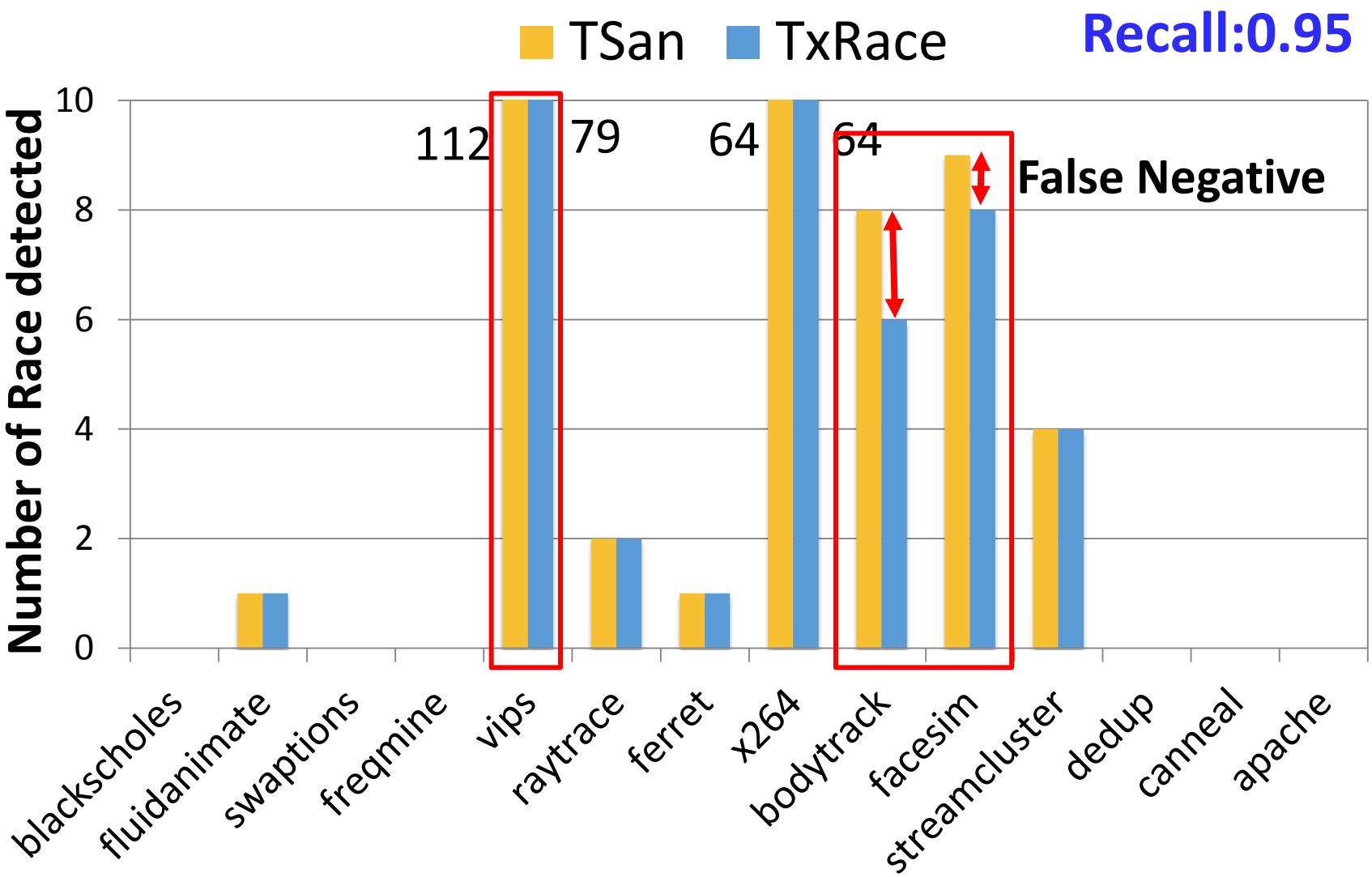
Evaluation

- PARSEC benchmark suites with simlarge input
- Apache web server with 300K requests from 20 clients
- 4 worker threads (4 hardware transactions)

Performance Overhead

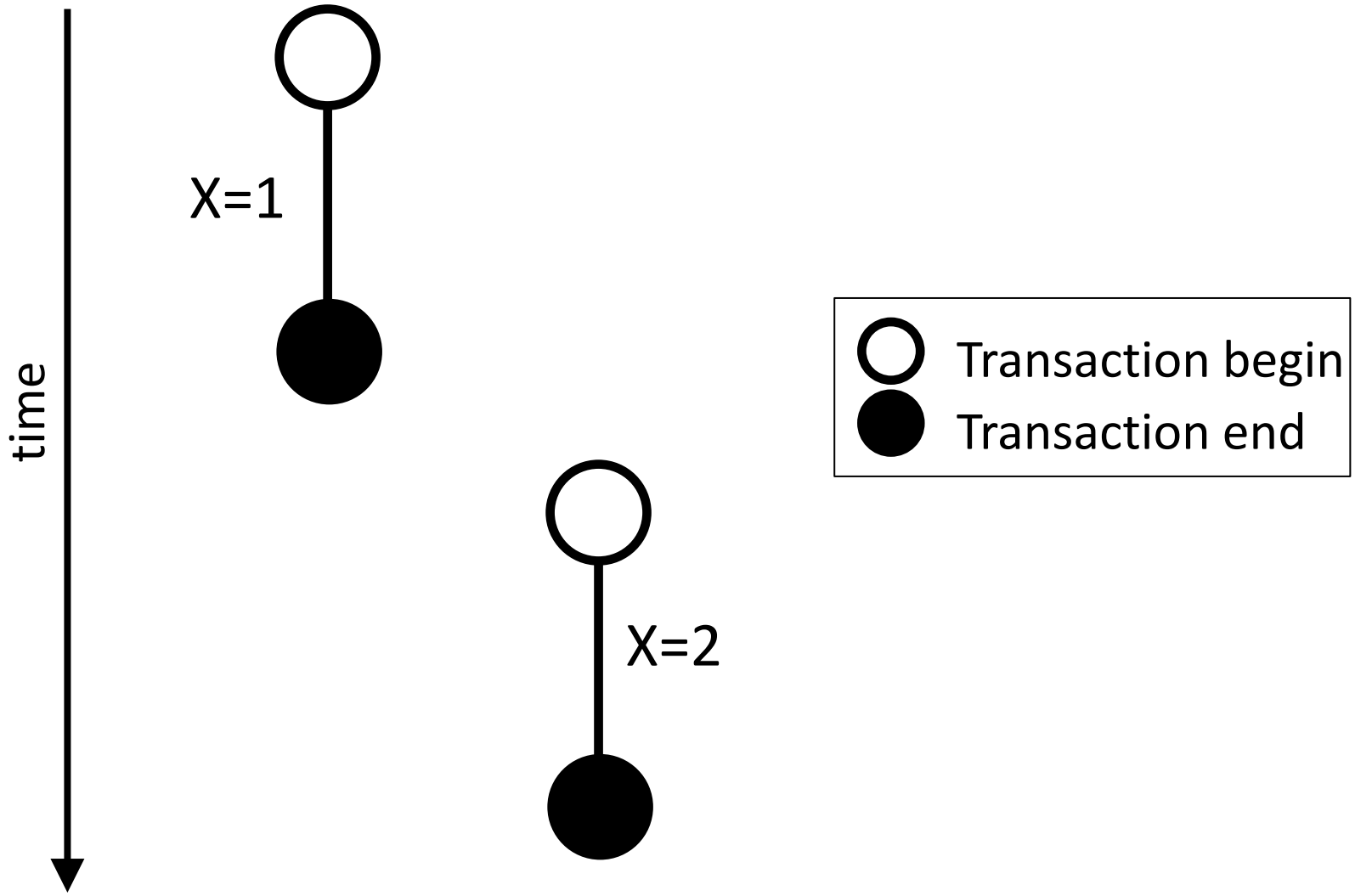


Soundness (Detection Capability)



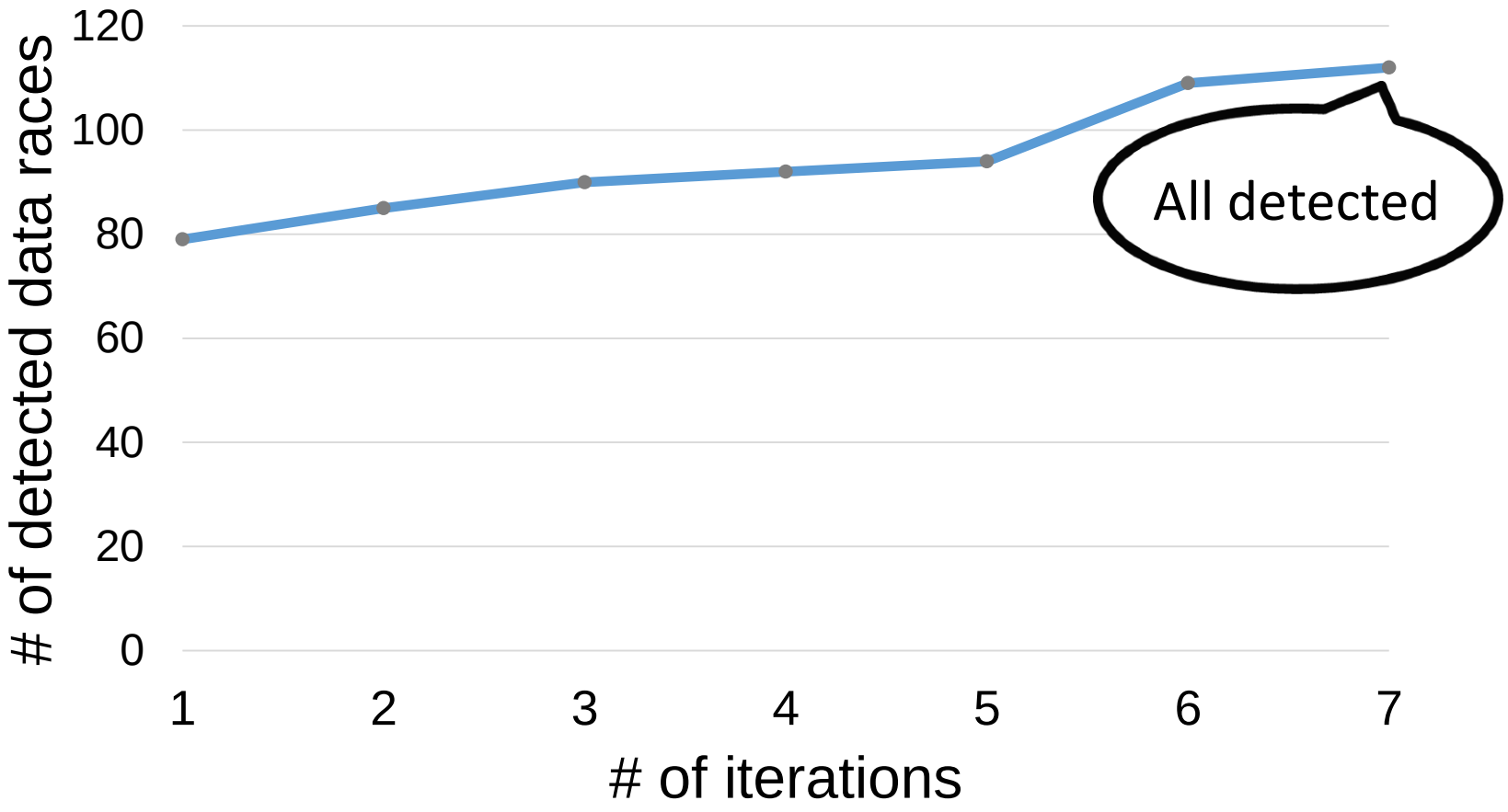
False Negatives

- Due to non-overlapped transactions



False Negatives Case Study in *vips*

- Repeat the experiment to exploit different interleaving



Conclusion

- Goal: to help programmers to develop reliable and secure concurrent systems
- **DoublePlay**: Deterministic multiprocessor replay with external determinism and uni-parallelism
- **TxRace**: Data race detection using hardware transactional memory in commodity processors
- On-going research: Concurrency analysis for event-driven programs (e.g., mobile, Node.js, IoT, etc.)

Roadmap

- Motivation
- DoublePlay: Deterministic Replay
- TxRace: Data Race Detector
- On-going work
- Conclusion

Support for Next-Generation Concurrent Systems

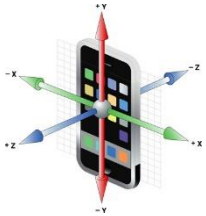
100+ cores
Desktop/Server

Event-Driven Programs

8+ cores
Smartphones

2+ cores
Smart Things

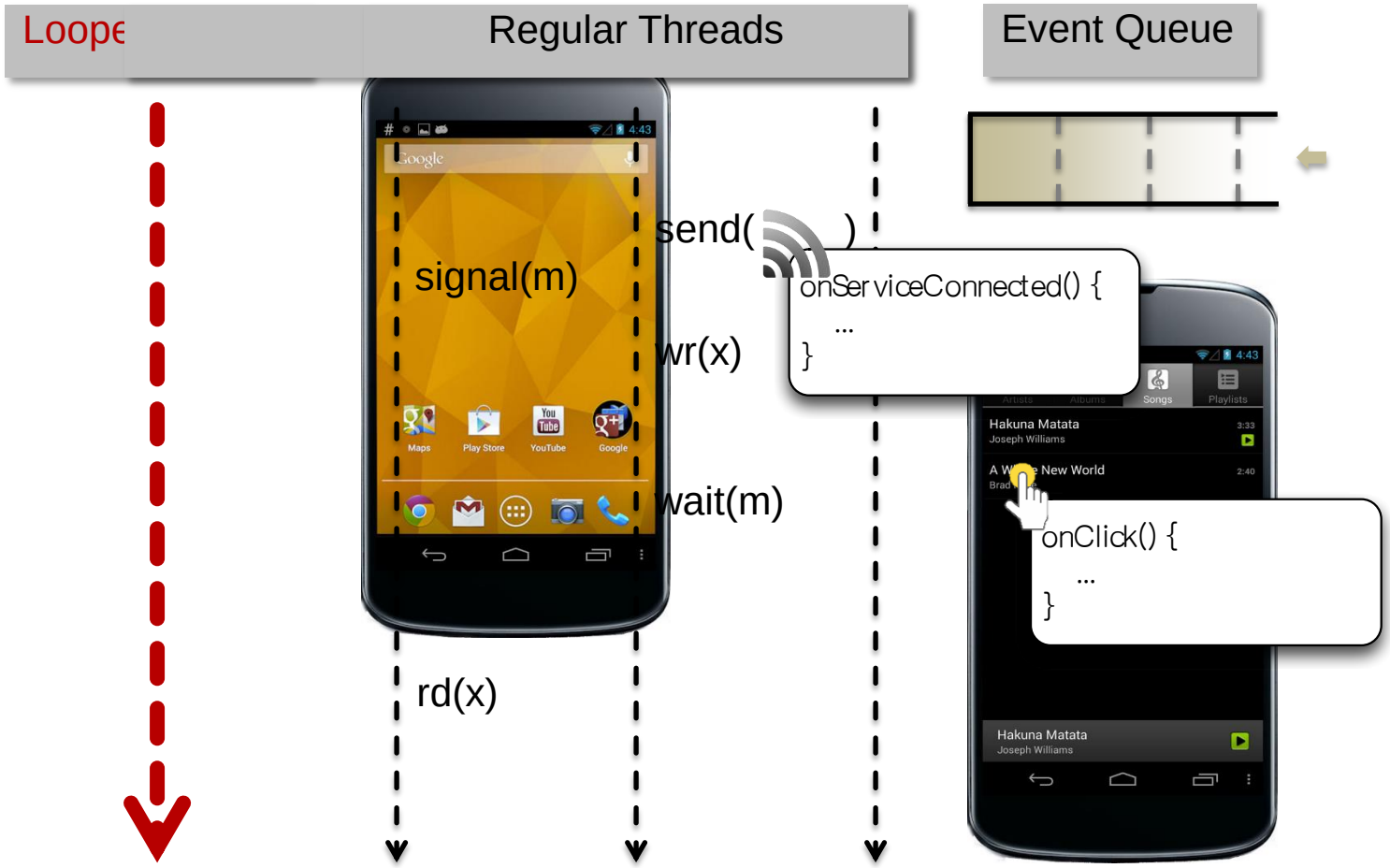
Why Event-Driven Programming Model?



Need to process asynchronous input
from a rich set of sources

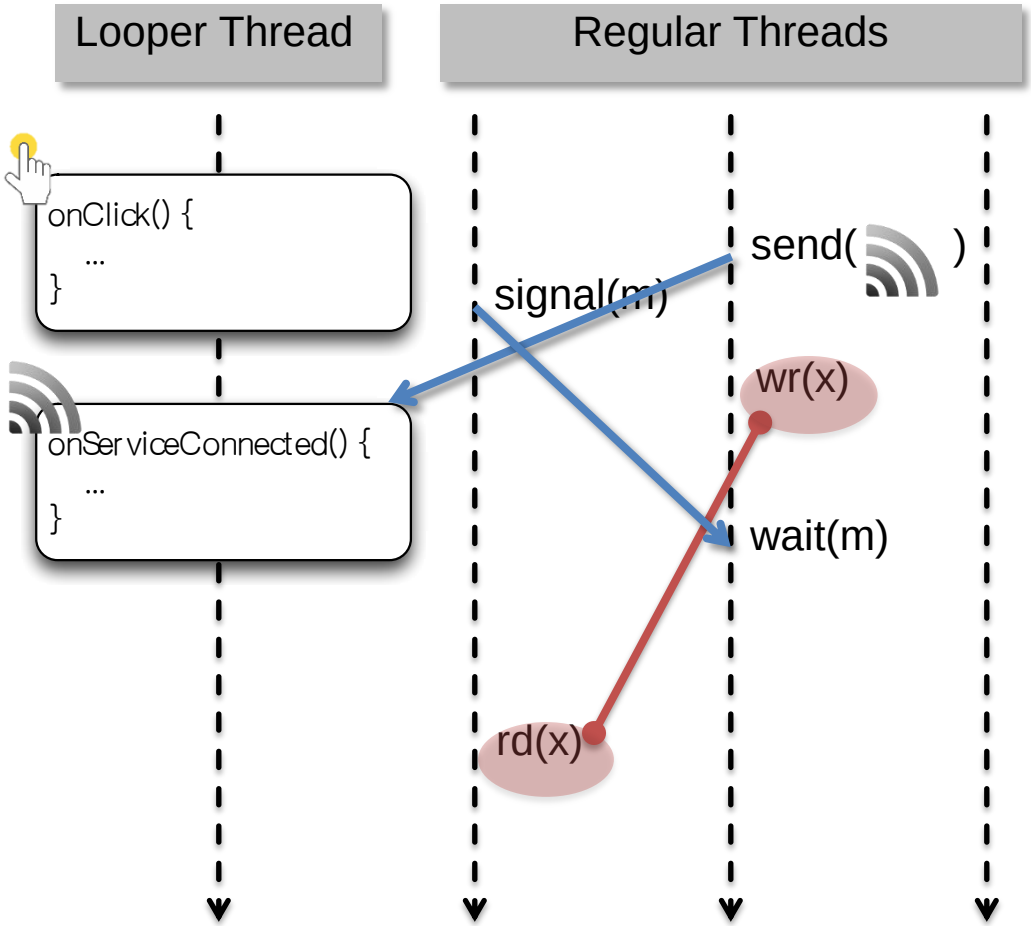
Slides extracted from “Hsiao et al’s PLDI 2014 presentation

Events and Threads in Android



Slides extracted from “Hsiao et al’s PLDI 2014 presentation

Non-determinism: Multi-thread Data Race



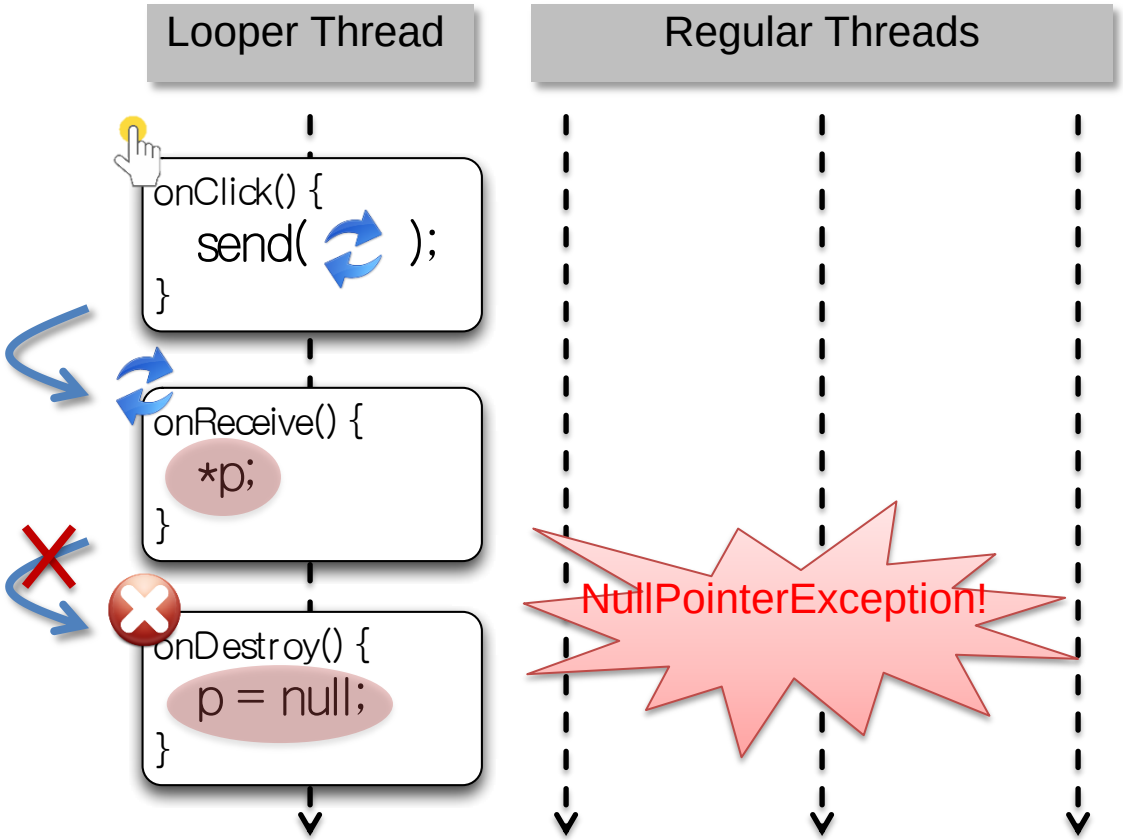
Causal order: happens-before (→) defined by synchronization operations

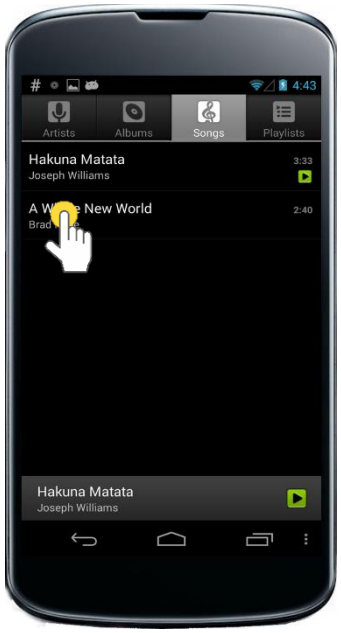
Conflict: Read-Write or Write-Write data accesses to same location

Race ( ): Conflicts that are not causally ordered

Slides extracted from "Hsiao et al's PLDI 2014 presentation

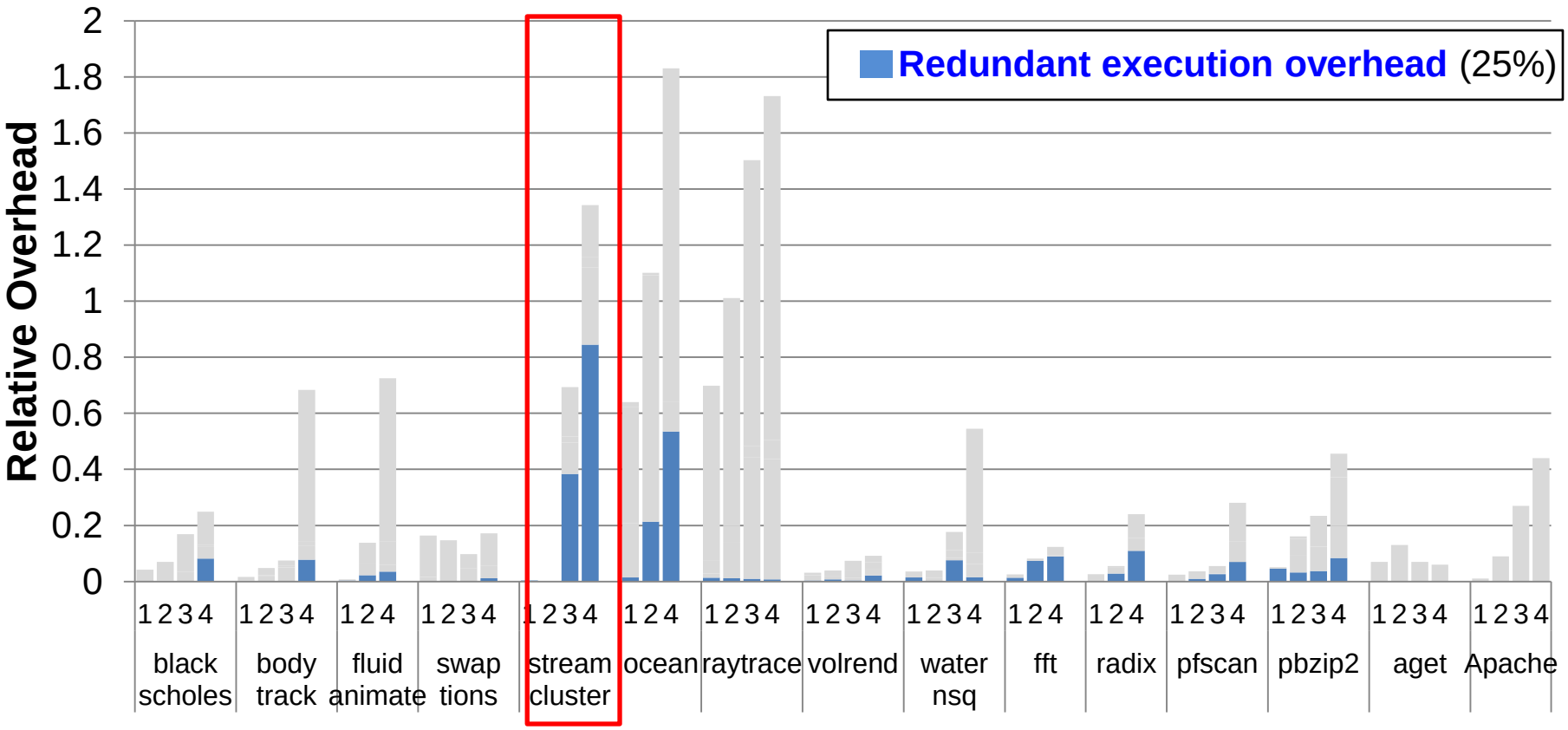
Non-determinism: Single-thread Data Race





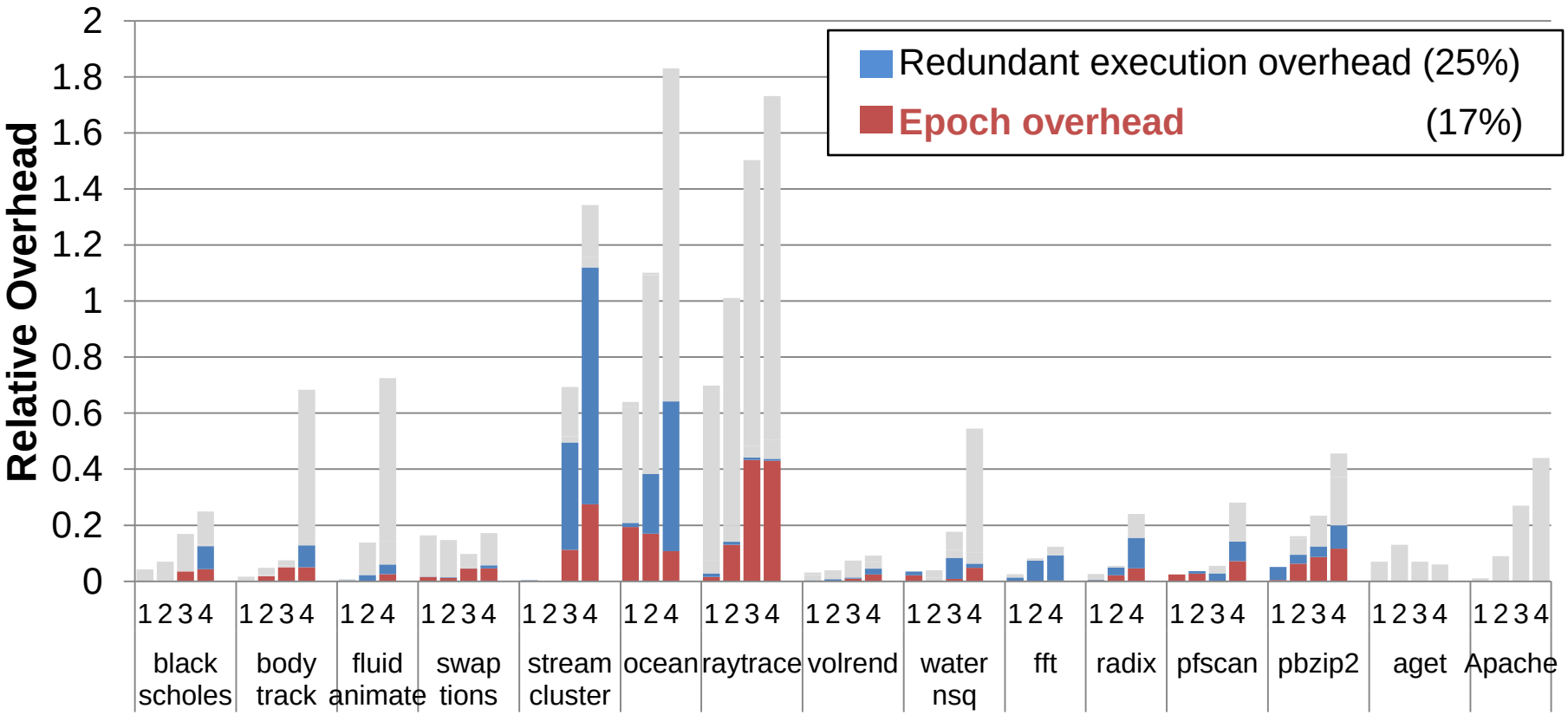
Slides extracted from “Hsiao et al’s PLDI 2014 presentation

1) Redundant Execution Overhead (25%)



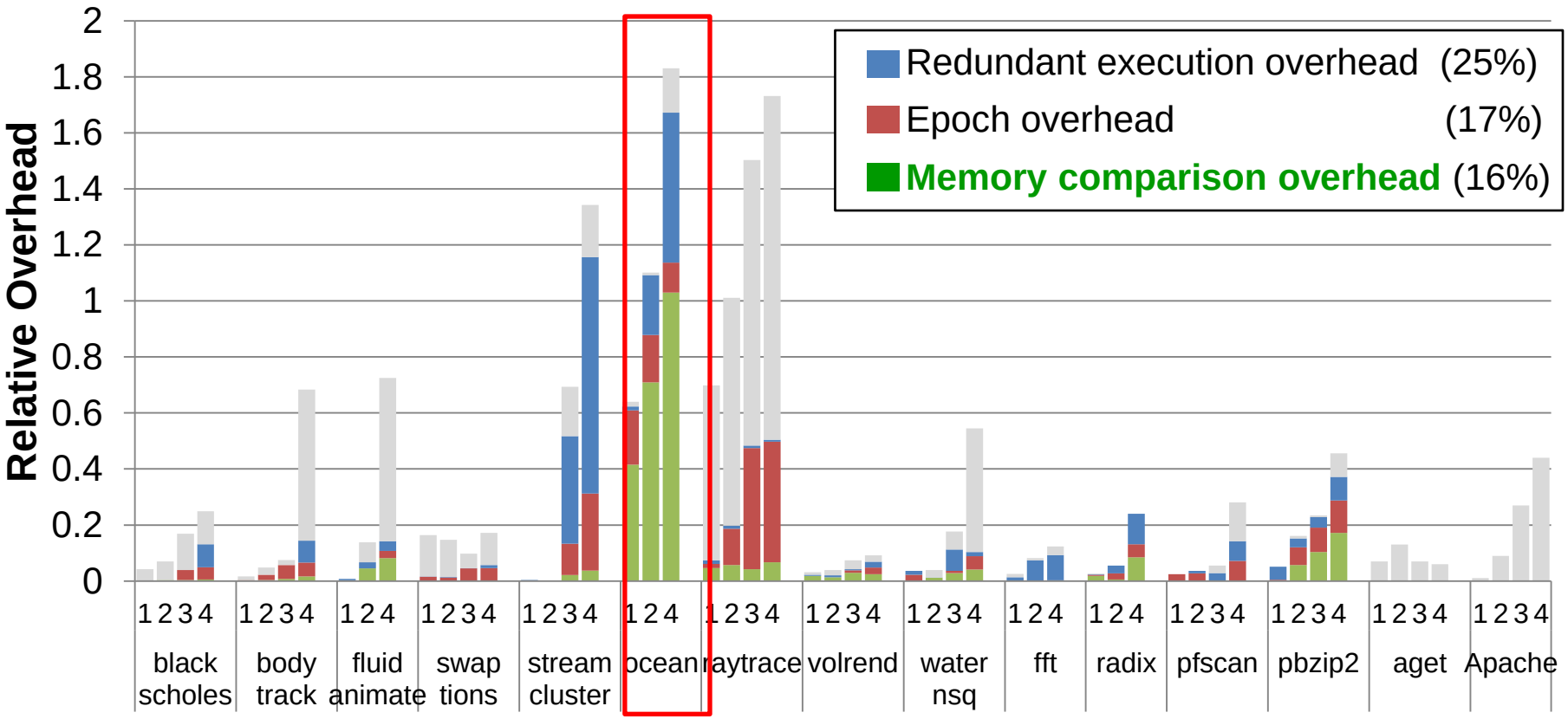
- Cost of running two executions (lower bound of online replay)
- Mainly due to sharing limited resources: memory system
- Contribute **25%** of total cost for 4 threads

2) Epoch overhead (17%)



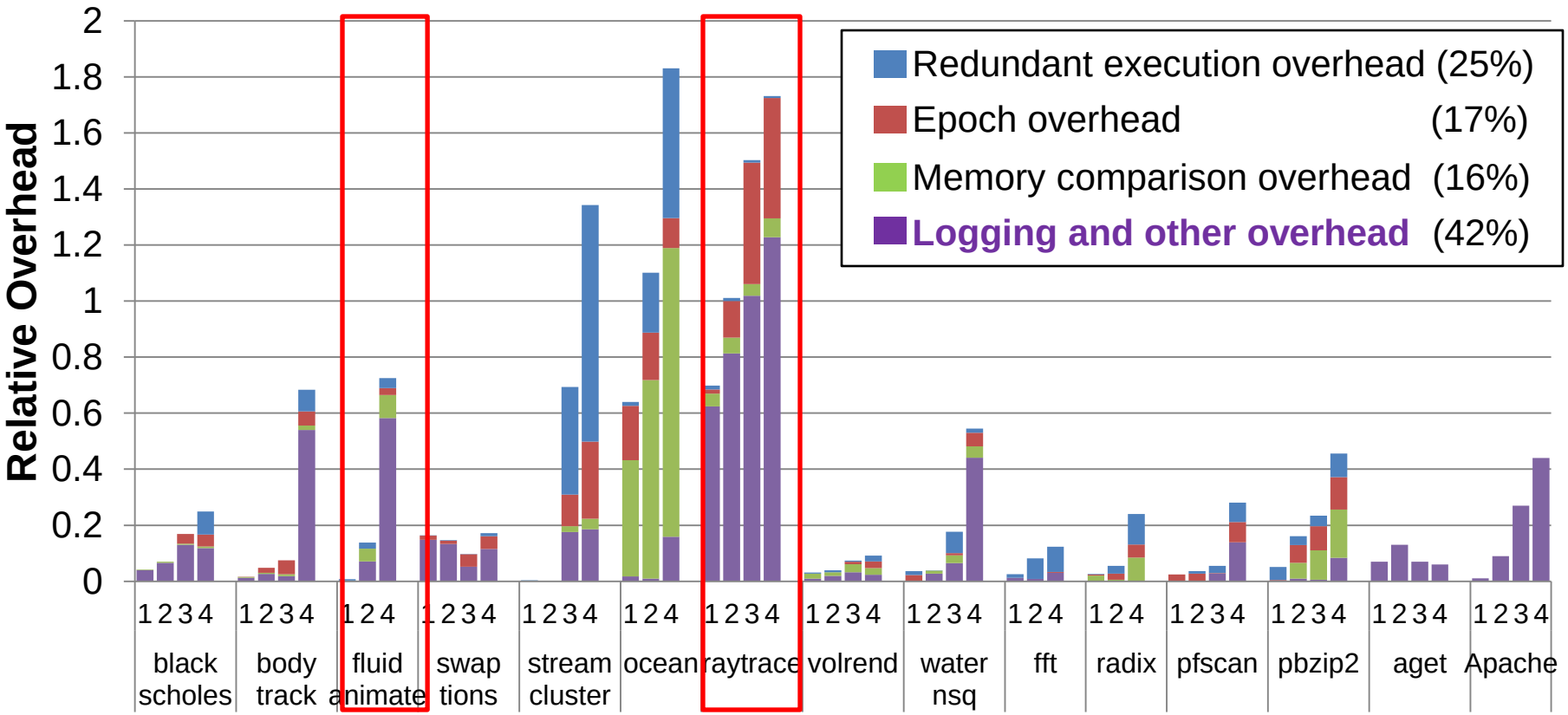
- Due to checkpoint cost
- Due to artificial epoch barrier cost
- Contribute **17%** of total cost for 4 threads

3) Memory Comparison Overhead (16%)



- Optimization 1. compare dirty pages only
- Optimization 2. parallelize comparison
- Contribute **16%** of total cost for 4 threads

4) Logging Overhead (42%)



- Logging synchronization operations and system calls overhead
- Main cost for applications with fine-grained synchronizations
- Contribute **42%** of total cost for 4 threads