# ReEnact: Using TLS Mechanisms to Debug Data Races in Multithreaded Codes

Milos Prvulovic* and Josep Torrellas

University of Illinois at Urbana-Champaign

*Soon to be at Georgia Tech

# Motivation

- Software bugs are expensive
- Bugs remain in production SW
  - Usually the most elusive ones - “natural selection”
  - Difficult to collect relevant debug info at user’s site
  - Difficult to repeat and debug at developer’s site
- Debugging multithreaded codes even more difficult
  - Heisenbugs

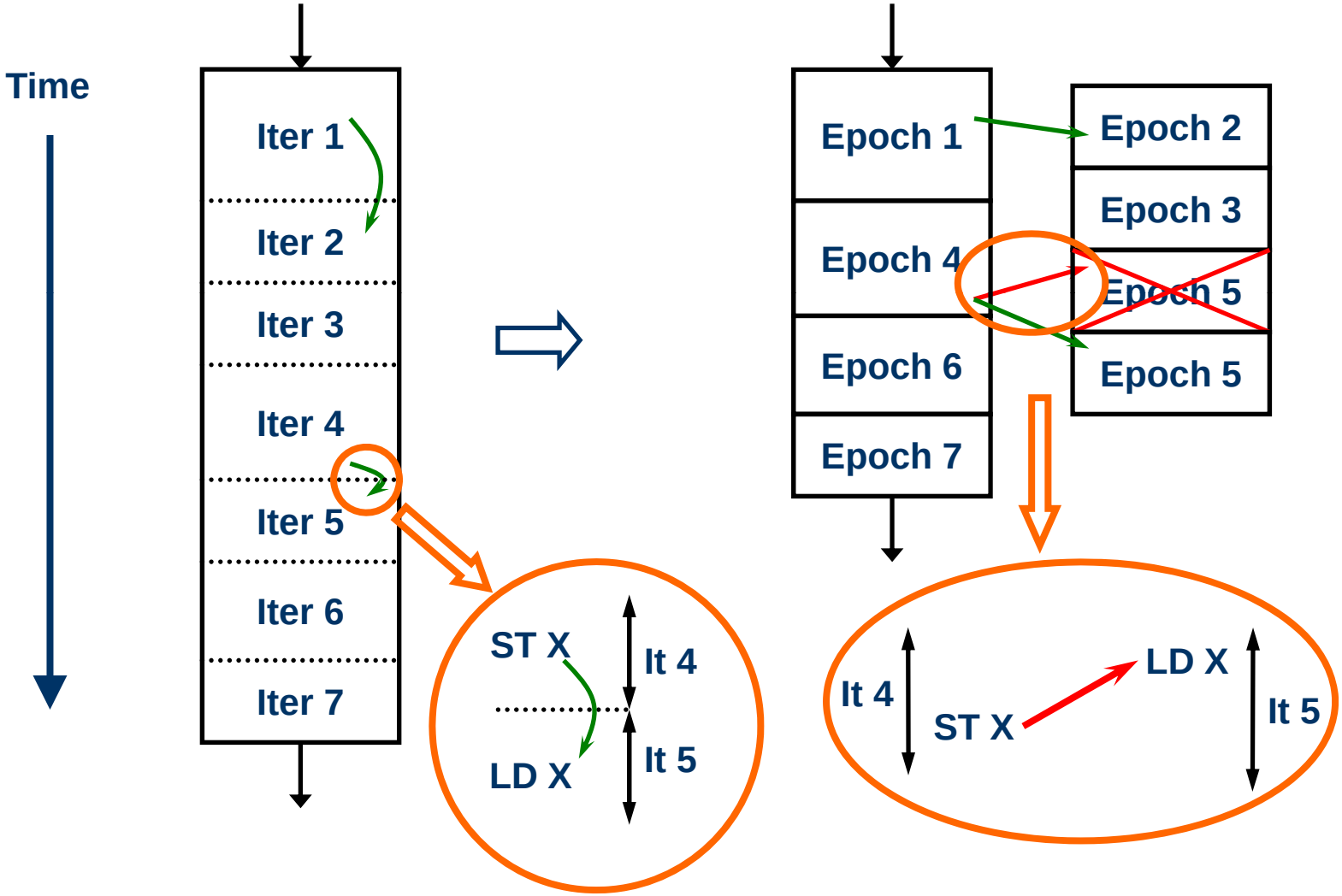
# Goal

- Automatic collection of relevant debugging information
  - On-The-Fly: when and where a bug occurs
    - Need to repeat recent past execution for analysis
  - Always-On: even in production runs
    - Must have little performance overhead
  - Deterministic: even for Heisenbugs
    - Eliminate non-determinism from re-executions
- Add as little HW as possible

# Contribution: ReEnact

- HW support for debugging of data races
- On-The-Fly Data Race Detection
  - Extends TLS memory dependence detection
- Deterministic Bug Analysis
  - Extends TLS In-Cache Buffering and Re-Execution
- Always-On operation
  - Only 6% performance overhead in race-free execution
- Works!

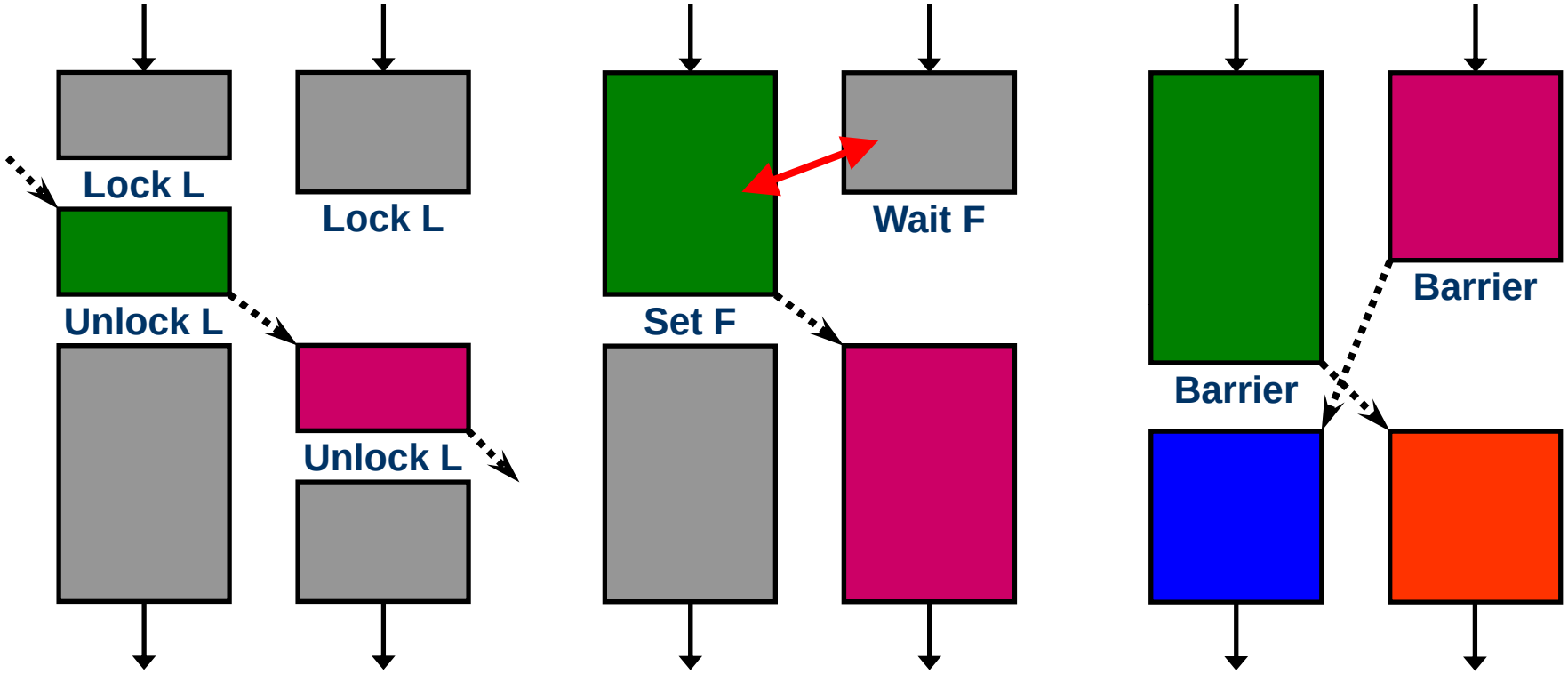
# Thread-Level Speculation (TLS)



# Overview of ReEnact

- Divide each thread of execution into epochs
- Buffer recently executed epochs
- Track epoch ordering
  - Within the same thread: sequential order
  - Across threads: synchronization creates order

# Epoch Ordering by Synchronization




- Race detection ⇔ unordered epochs communicate

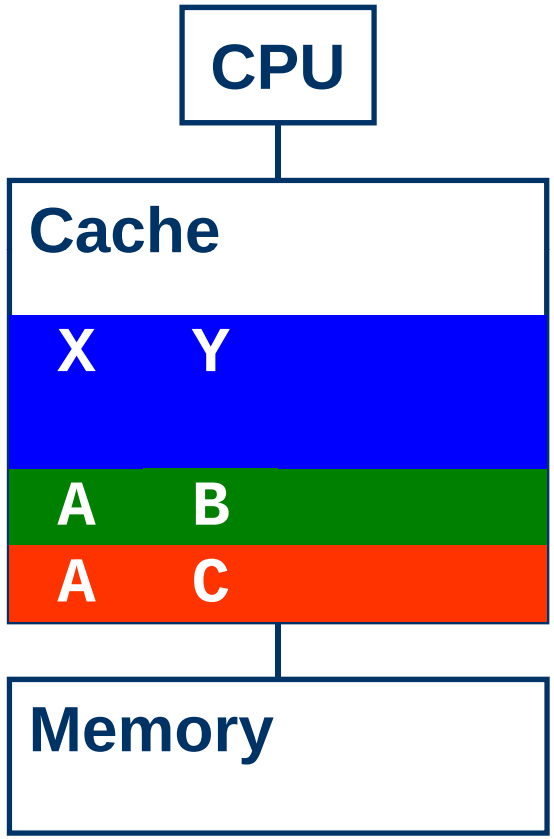
# Overview of ReEnact

- Analysis of data races
  - Re-execute buffered epochs using the same ordering
  - Gather information on data race accesses
- Repair of data races
  - Examine gathered information
  - Figure out the cause of data races
  - If possible, introduce the missing ordering and proceed

# ReEnact: Epoch Buffering

## Dynamic Instructions

```
ST X
ST Y
ADD
 ...
ST A
ST B
ST A
 ...
ST A
 ...
ST C
```

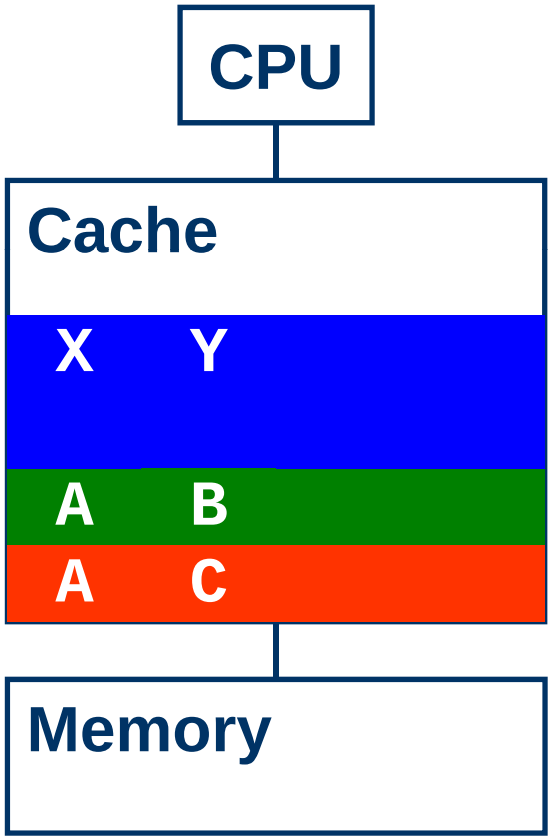



# ReEnact: Commit

## Dynamic Instructions

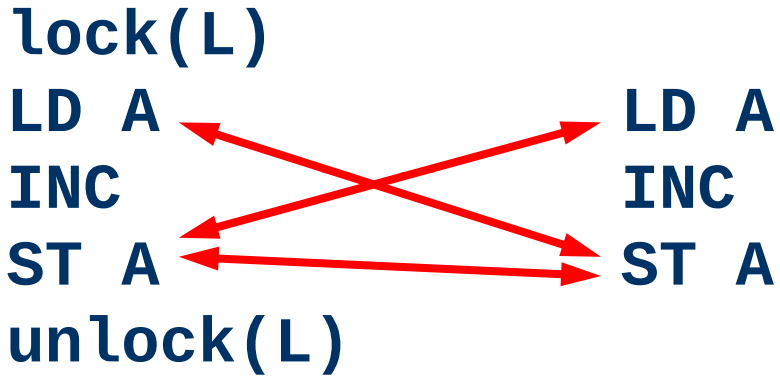
```
ST  X
ST  Y
ADD
 …
ST  A
ST  B
ST  A
 …
ST  A
 …
ST  C
```

Need to displace X from this epoch

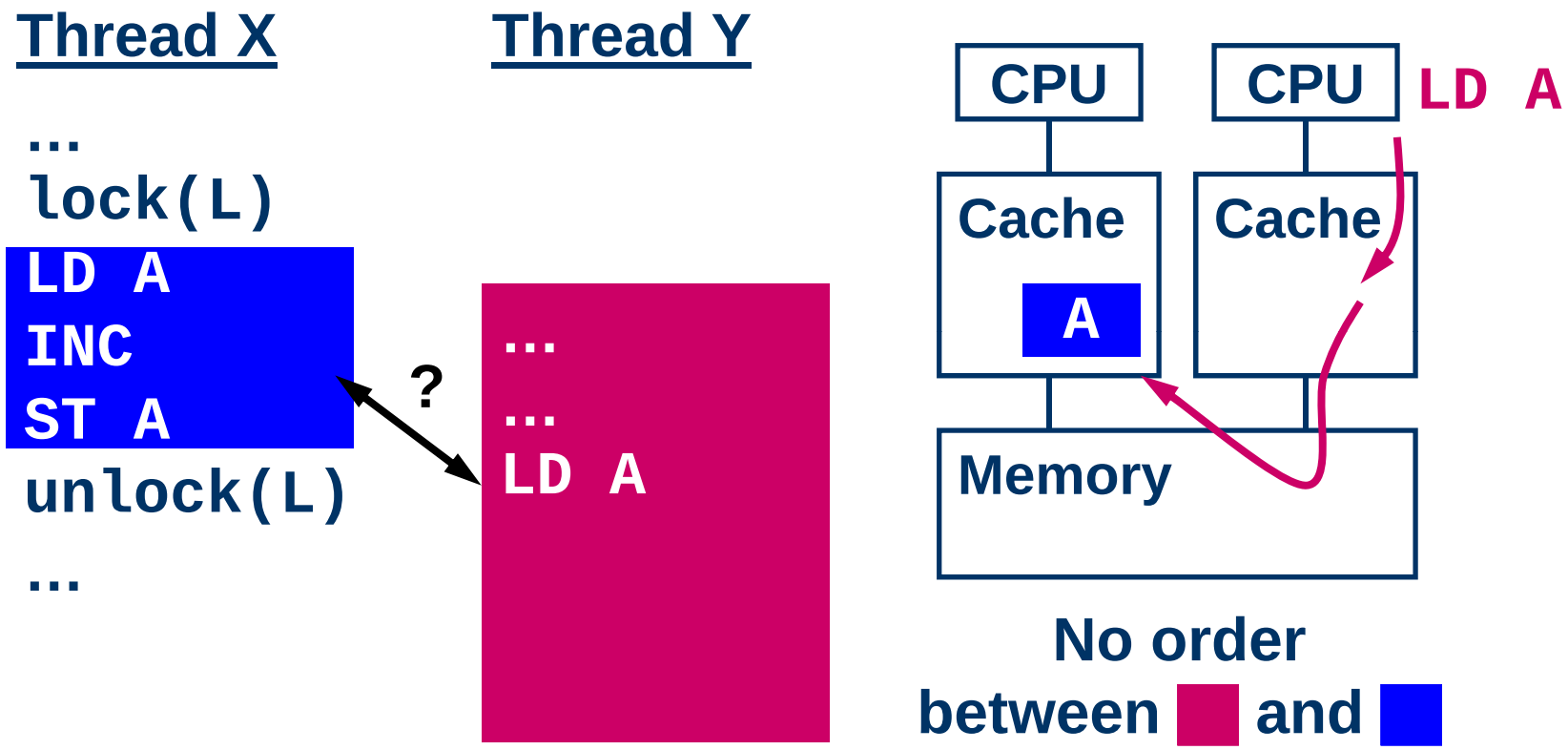



# Example: Missing Critical Section

**Thread X** **Thread Y**




# Detection: Data Race



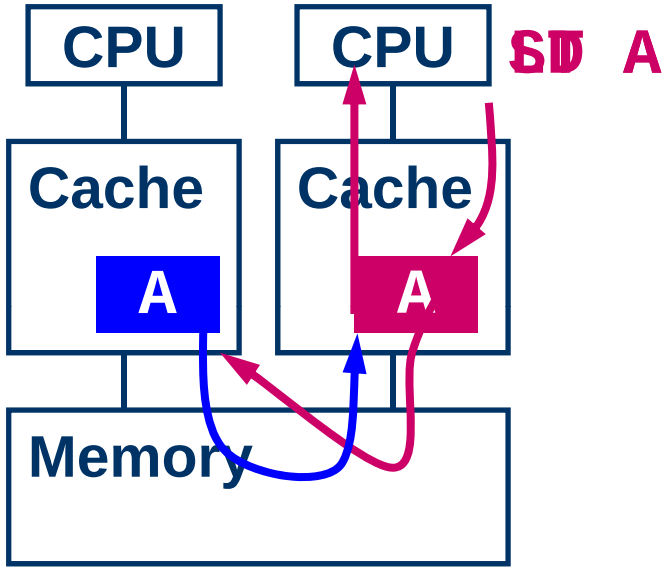
# Analysis: Find Race Signature

**Thread X**

```
...
lock(L)
LD A
INC
ST A
unlock(L)
...
```

**Thread Y**

```
...
...
LD A
INC
ST A
...
```




**No order between and**

**Assume order: after**

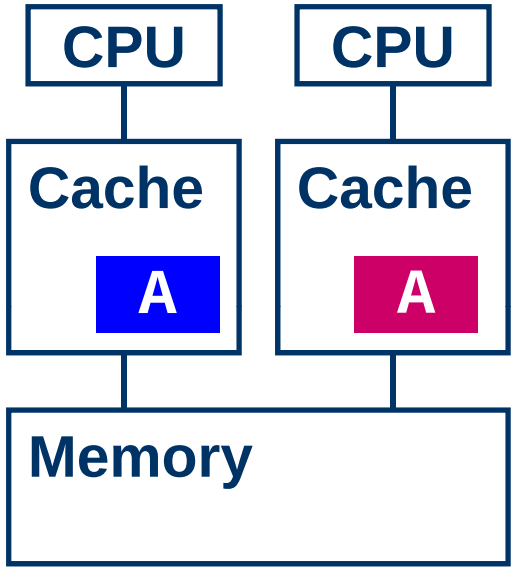
# Analysis: Refine Race Signature

**Thread X**

```
...
lock(L)
LD A
INC
ST A
unlock(L)
...
```

**Thread Y**

```
...
...
LD A
INC
ST A
...
```




**Assumed order: after**

**Put a watchpoint on accesses to data address A**

# Repair: Signature Matching

- Analysis resulted in a detailed signature
  - Instruction & data addresses, data values, timing, etc.
- Programmer uses the signature to fix the program, or
  - Usually a simple task in our experience
- Automated pattern-match w/ a library of errors to
  - Suggest repair to programmer, or
  - Try to automatically re-introduce missing ordering
- Re-execute with corrections

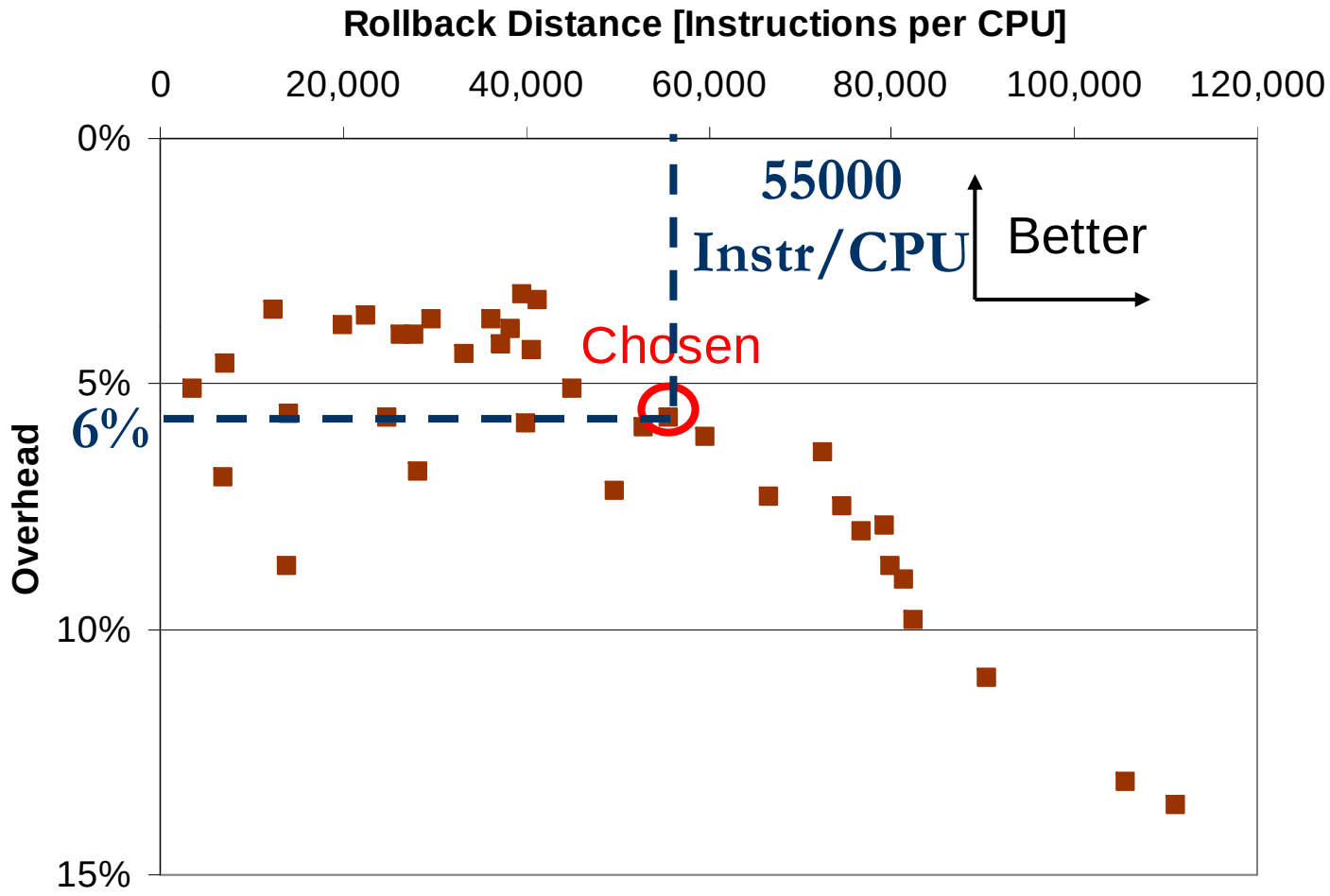
# ReEnact vs. TLS

- Correctness
  - TLS must detect and repair all dependence violations
  - ReEnact tries to detect and repair data races
- Commit
  - TLS: when epochs is safe and complete
  - ReEnact: when buffer space runs out
- Ordering
  - TLS: total ordering
  - ReEnact: partial ordering

# Evaluation Setup

- Splash-2 benchmarks
- Execution-driven simulation
- Chip-multiprocessor with 4 processors

# ReEnact: Overhead



# How Good ReEnact is to:

| | Detect | Rollback | Analyze | Match |
|---|---|---|---|---|
| Sync through plain variables | ✓ | ✓ | ✓ | ✓ |
| Missing Lock | ✓ | ✓ | ✓ | ~ |
| Missing Barrier | ✓ | ~ | ~ | ~ |

# ReEnact vs. FDR (next paper)

- Both address recording and deterministic re-play of recent multithreaded execution
- Different targets
  - ReEnact: data race detection, analysis, repair(?)
  - FDR: general-purpose recording for off-line analysis
- Both are useful and can be combined
  - ReEnact: detects data races, extracts relevant info
  - FDR: provides support for long-range replay

# Conclusions

- Architectural support for debugging data races
- Automatic Detection, Analysis
  and some Correction of synchronization bugs
- Low overhead in error-free execution (6%)
- Future work – other classes of bugs
  - Need a detection mechanism
  - Need heuristics for analysis and repair
  - Buffering and deterministic re-execution:
    ReEnact HW support largely unchanged

# ReEnact: Using TLS Mechanisms to Debug Data Races in Multithreaded Codes

Milos Prvulovic* and Josep Torrellas

University of Illinois at Urbana-Champaign

*Soon to be at Georgia Tech