RelaxReplay: Record and Replay for Relaxed-Consistency Multiprocessors

Nima Honarmand and Josep Torrellas

University of Illinois at Urbana-Champaign

http://iacoma.cs.uiuc.edu/

ILLINOIS

illinois.edu

i-acoma group

RnR: Record and Deterministic Replay

- Recreate execution of a (parallel) program
  - Record: capture non-deterministic events in a log
  - Replay: use the log to recreate the exact same execution
- Use cases: **Debugging**, **Security**, **Fault Tolerance**, …

Sources of non-determinism

- Program inputs
  - Capture using operating system support
- Memory access interleaving
  - **Memory Race Recording (MRR)**
  - Important in multiprocessor systems
  - Capture using hardware support

My Focus: Hardware-based memory race recording in relaxed-consistency multiprocessors

RelaxReplay Contributions

1) A hardware-based scheme for memory race recording in multi-processors with relaxed memory models

2) Design independent of memory model details
   - Only relies on conventional cache coherence

3) Compact log format
   - Log size comparable to previous, less general, schemes

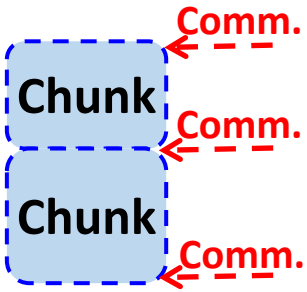
4) Log format enables efficient replay

Memory Race Recording (**MRR**) in Hardware

Observation: Shared-memory communications manifest as cache coherence transactions

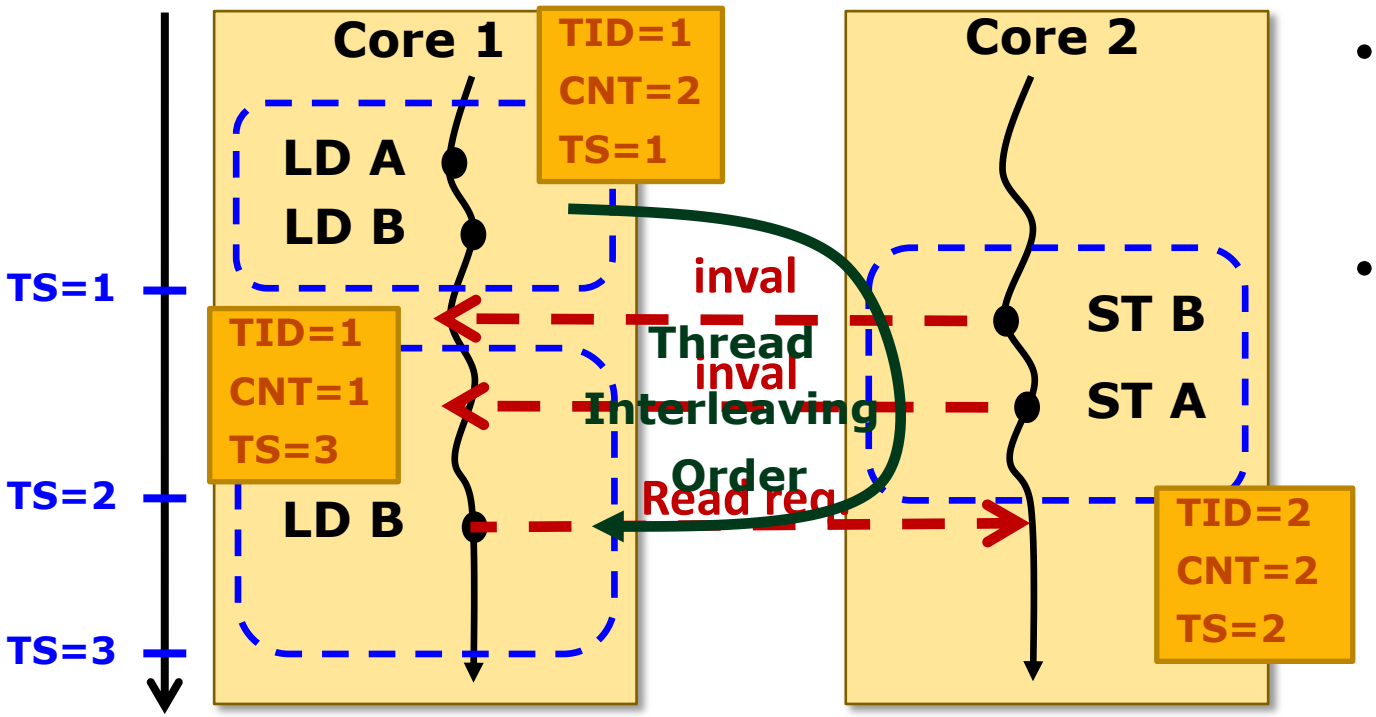
- Can do MRR by monitoring transactions

Chunk-Based Recording



1) Dynamically, break processor's execution into **chunks** of instructions
   - Chunk: the work done between consecutive communications with other processors
2) Order chunks across processors such that
   - Chunk containing *source* of a dependence is ordered before chunk containing *destination*
3) Replay chunks in recorded order to reproduce dependences

Chunk-Based Recording Example



- Each processor keeps a Read Set and a Write Set
- Read and Write Sets are cleared after chunk creation

- # of Chunks << # of dependences
- Chunk content logged efficiently as <# of insts>

  → **Compact log**

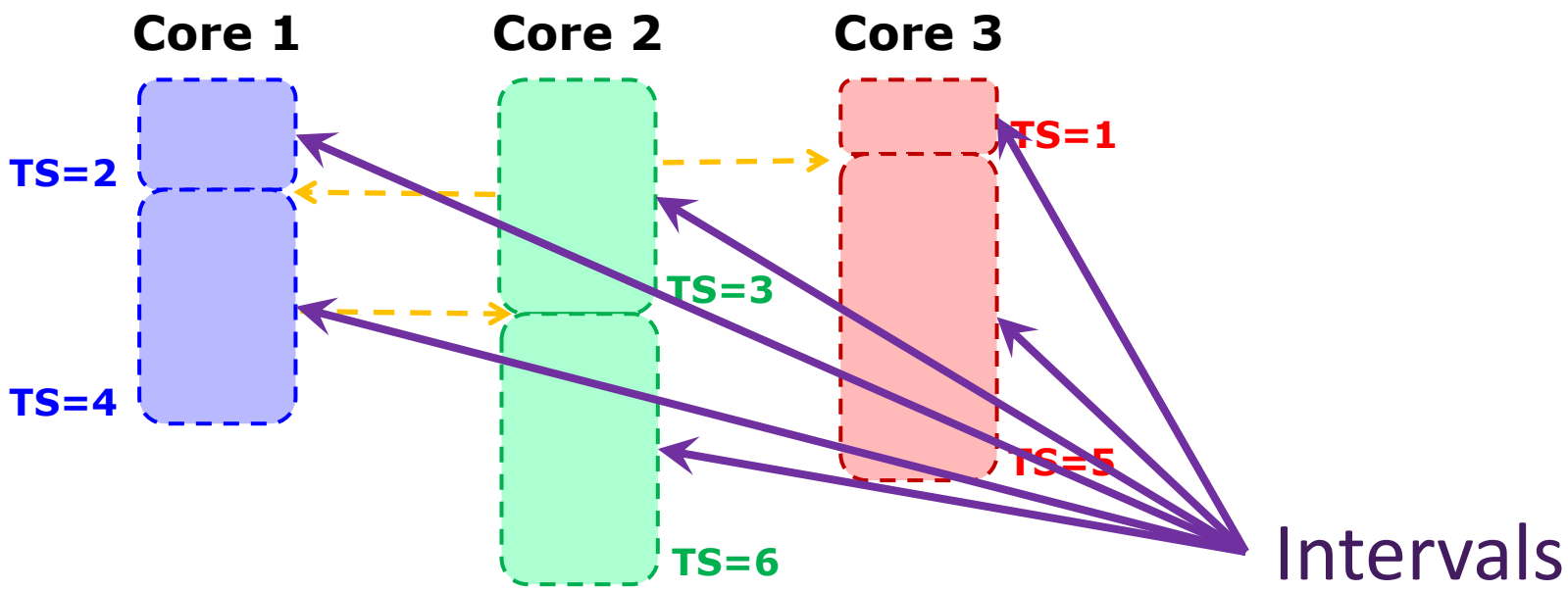
Problem: Access Re-ordering in Relaxed Models

- Chunk-based recording assumes in-program-order execution of memory accesses (Sequential Consistency)

- Real processors re-order memory accesses
  - Recording "Number of instructions" not enough to capture work done between consecutive communications

- Previous work support Total Store Ordering (TSO)
- Problem open for more aggressive relaxed model (ARM, IBM, …)
  - How to efficiently record heavily re-ordered accesses?

Notion of Interval in RelaxReplay

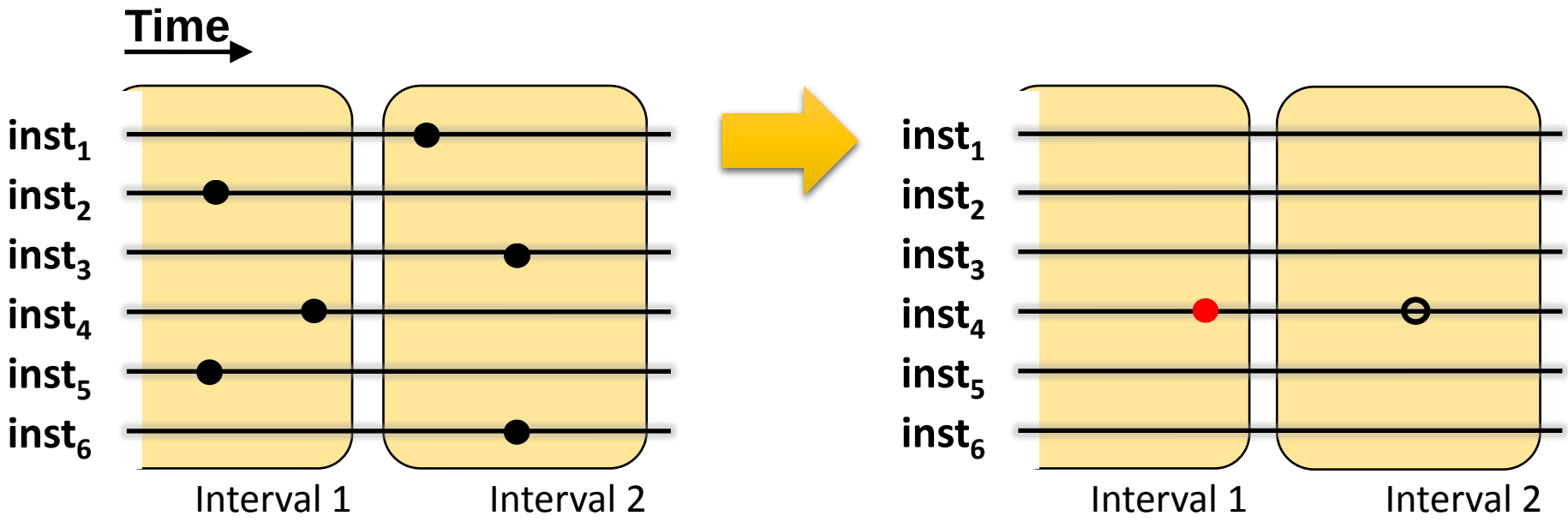


Interval: Period of time between consecutive communications with other processors

- Replaces notion of chunk

Instructions in an interval are not in program order
How to compactly log content of an interval?

Key Idea: Putting Instructions Back in Program Order



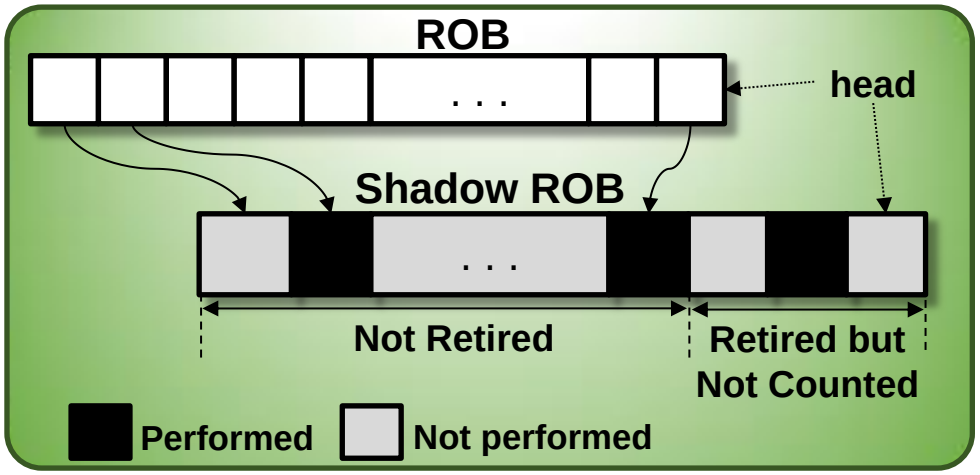
- Record an alternative order of execution with equivalent behavior and same set of inter-thread dependences
- Vast majority of instructions in program order → Compact log
- Some instructions cannot be put in program order → augment the log with additional information for correct play

Design: Decouple Instruction Execution and Logging

- Deferred logging: instructions go through a logging step, in program order
  - Detects instructions that can be put back in program order

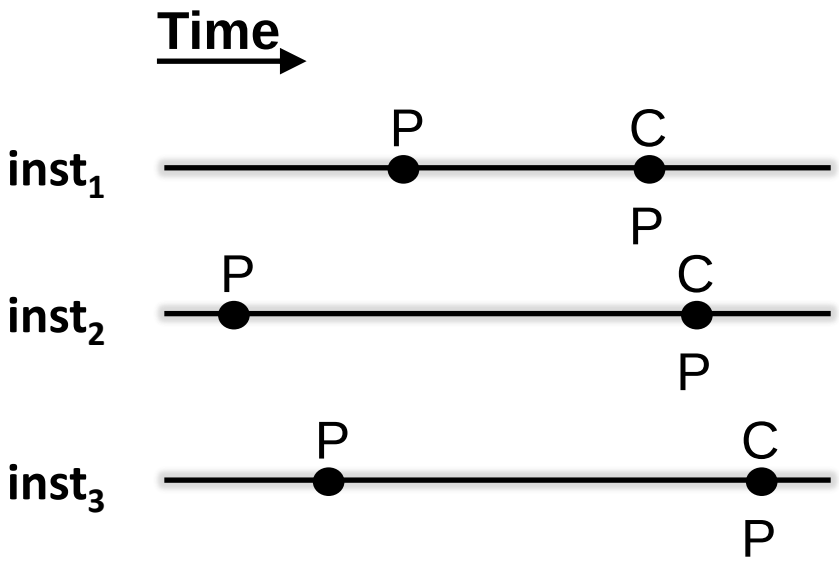
- Perform (**P**) Event: When instruction performs in memory
- Counting (**C**) Event: When instruction is added to the log
  - At the head of Shadow ROB
  - Inst should be performed

  → Happens in program order



- For each inst, try to **move P to C**
- Log an interval in terms of **its Counting (C) events**

Moving P Events to C Events



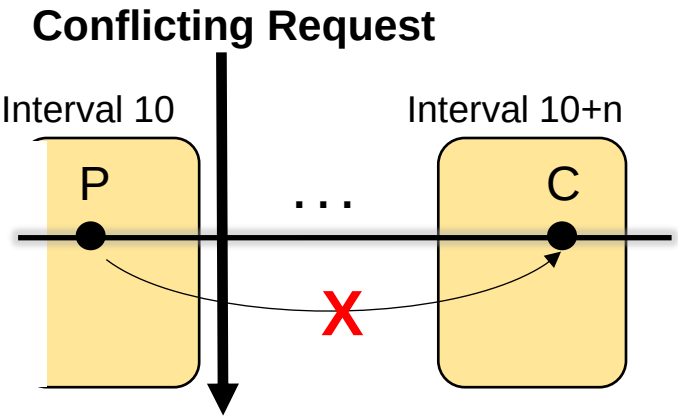
- Case 1 (**very common**):
  No conflicting coherence request between P and C

  → **safe to move P to C**
- We put P's back in program order → in-order inst
- Log a sequence of such insts as <# of insts>

Moving P Events to C Events

- Case 2 (**very rare**):

  Conflicting request between P and C → **can't move P to C**
  - Reordered inst
  - Special log entry



- Reordered loads
  - Log **the value** returned by load
  - During replay, read the value from the log (instead of memory)
- Reordered store
  - Log **the value, address** and **ID of P interval**

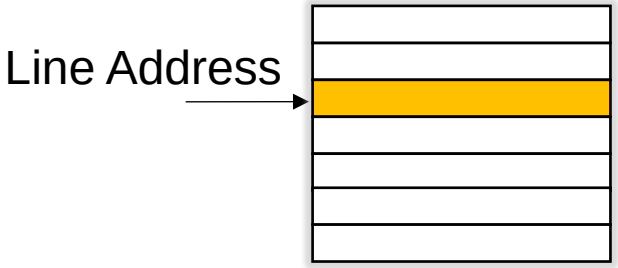
Detecting Conflicting Requests

- Naïve solution: Associatively search in-flight instructions
  - ✗ High hardware overhead

- **Basic Solution (BASE)**: Reordered if P and C in different intervals, otherwise in-order

- **Optimized Solution (OPT)**: Snoop Table
  - Table of counters
  - When snoop a request, index its address into the table and increment the counter

  → Reordered if different snoop counts at P and C

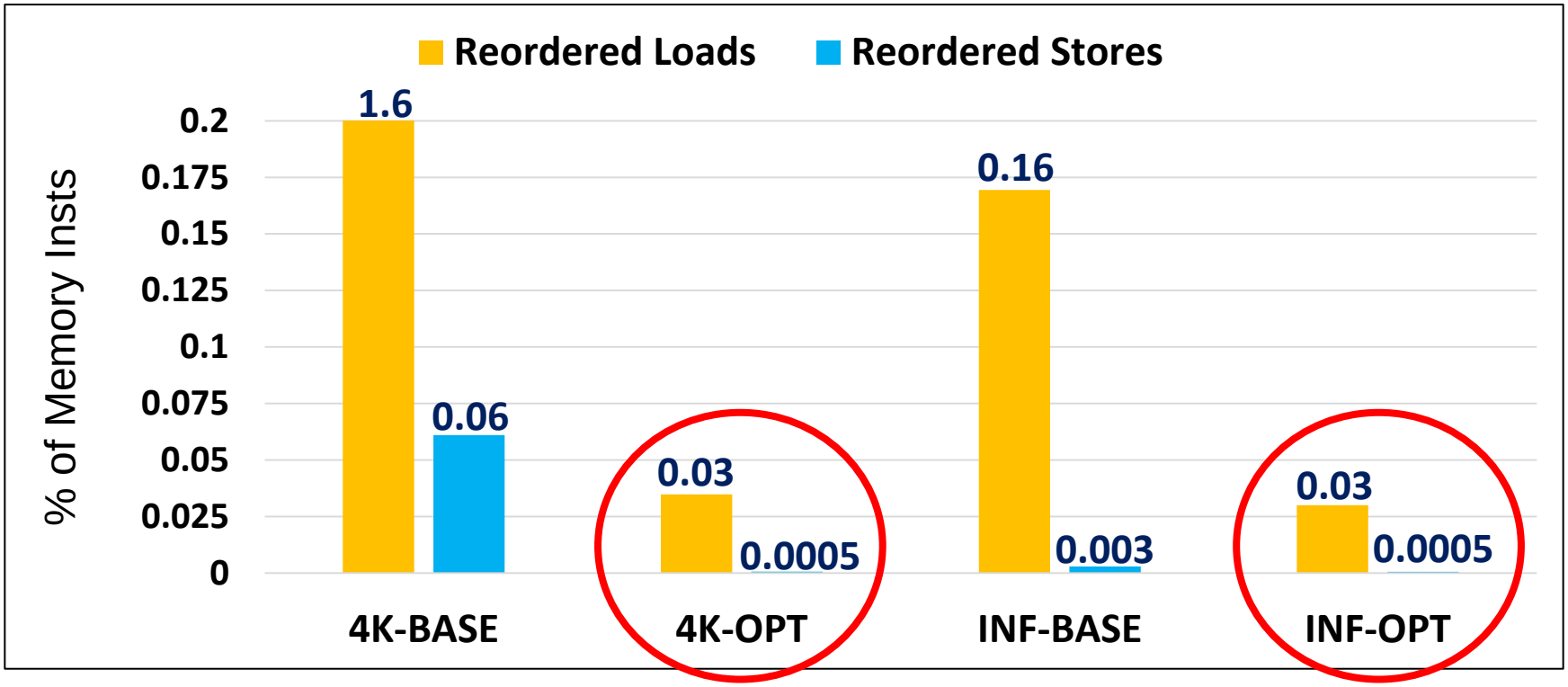


Evaluation Setup

- Processor: 8 cores, 4-way out-of-order, 2 GHz
- Memory subsystem: private L1, shared L2, Ring interconnect
- Memory model: Release Consistency
- Benchmarks: 10 SPLASH-2 programs

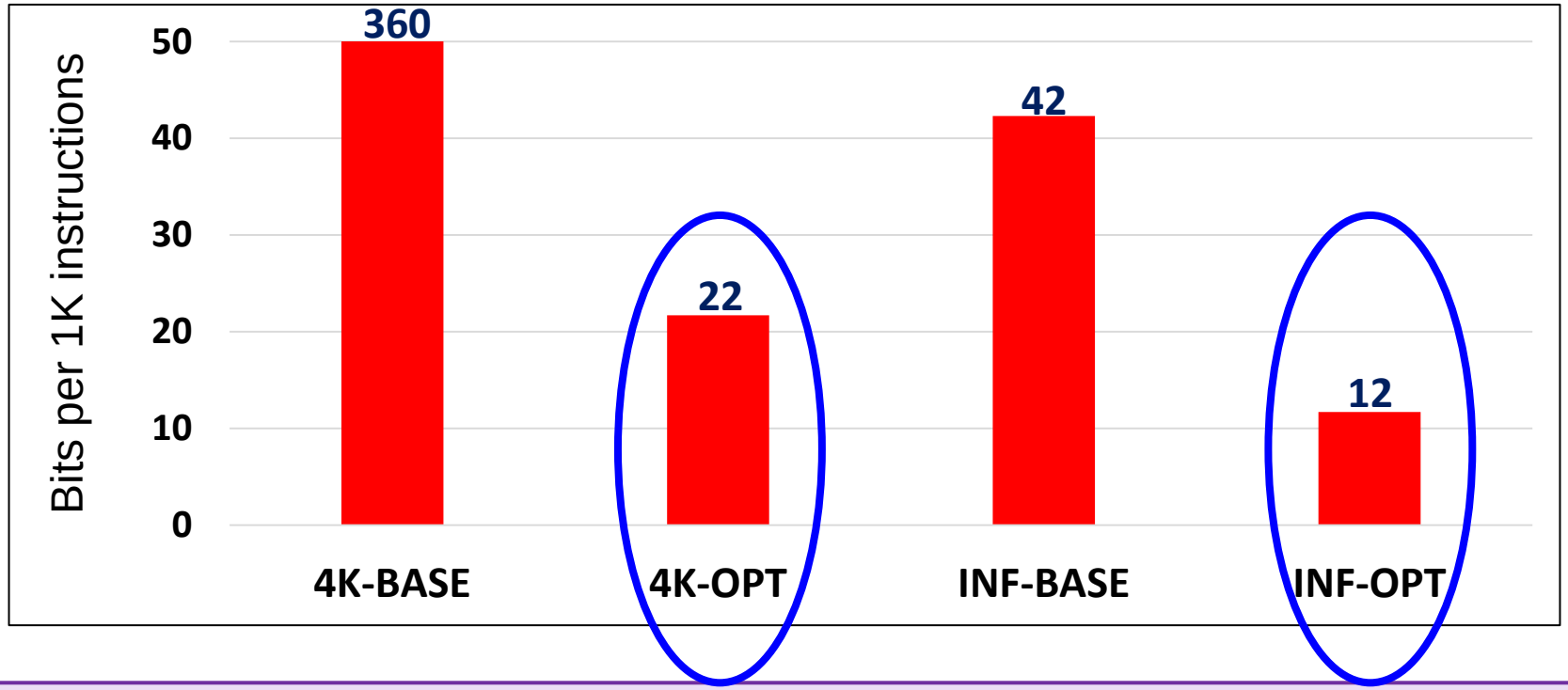
- Record:
  - Max interval sizes: 4K insts and INF
  - 4 configs: **4K-BASE**, **4K-OPT**, **INF-BASE** and **INF-OPT**

Number of Reordered Accesses



- **Less than 0.04% of mem. insts** re-ordered with OPT
  - Regardless of Max Interval Size
  - → Compact log

Log Generation Rate



- **Small log size:** 48 MB/sec (4K-OPT) and 25 MB/sec (INF-OPT) on 8 processors
  - Comparable to (1-4x) previous techniques
  - Very small compared to existing memory BW

Also in the Paper…

- Can combine RelaxReplay with any existing chunk-based recording scheme
  - In fact, I presented RelaxReplay on top of QuickRec [ISCA'13]
- Details of the replay algorithm
- More performance numbers
  - Replay performance
  - Scalability analysis (different numbers of processors)

Conclusions

- RelaxReplay: Memory race recording for relaxed memory models
- Design independent of memory model
  - Supports Intel, ARM, IBM, Tile, etc.
- Can be combined with existing chunk-based recorders
- Compact log format that enables efficient replay
- In practice, log size comparable to the ideal case of no re-ordering

THANK YOU!

Replaying RelaxReplay Logs

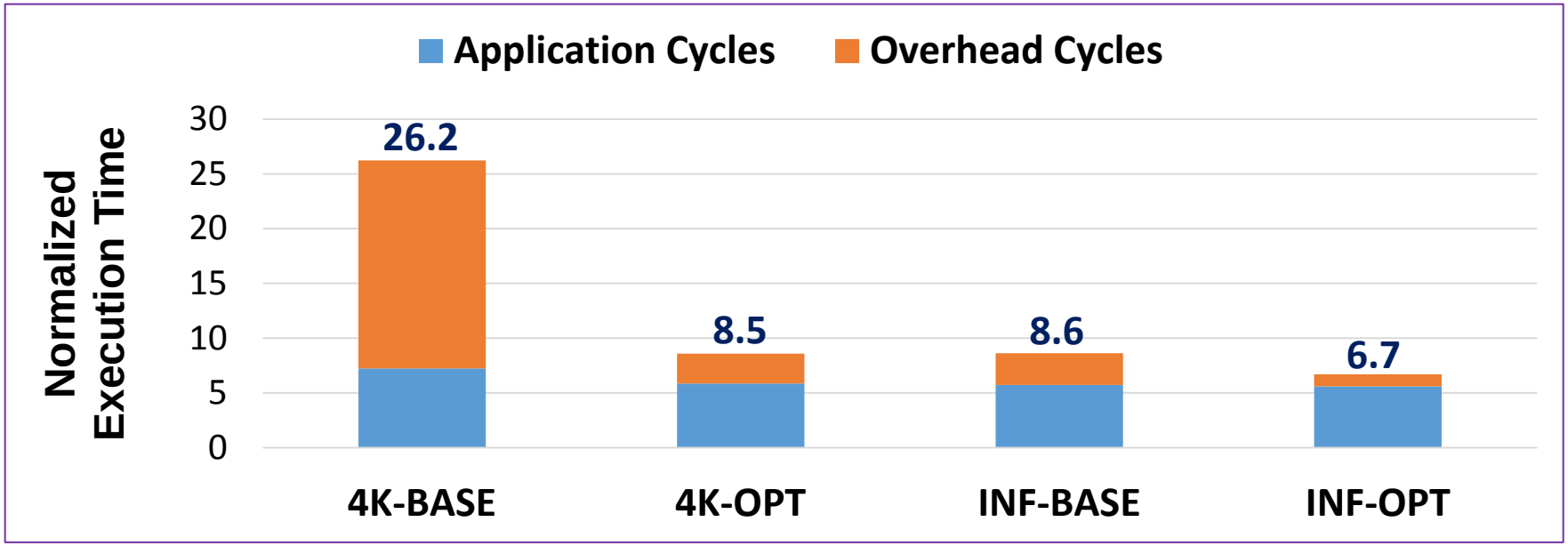
Before replaying an *interval* wait for its predecessors

```
While (interval has entries) {
        ent ← next entry of interval
        if (ent is Block) {
                execute next ent.size instructions
        } else if (ent is ReorderedLoad) {
                Read the load value from the log
        } else if (ent is ReorderedStore) {
                Read value and address from the log
                Execute the store
        }
}
```

Signal the successors of *interval*

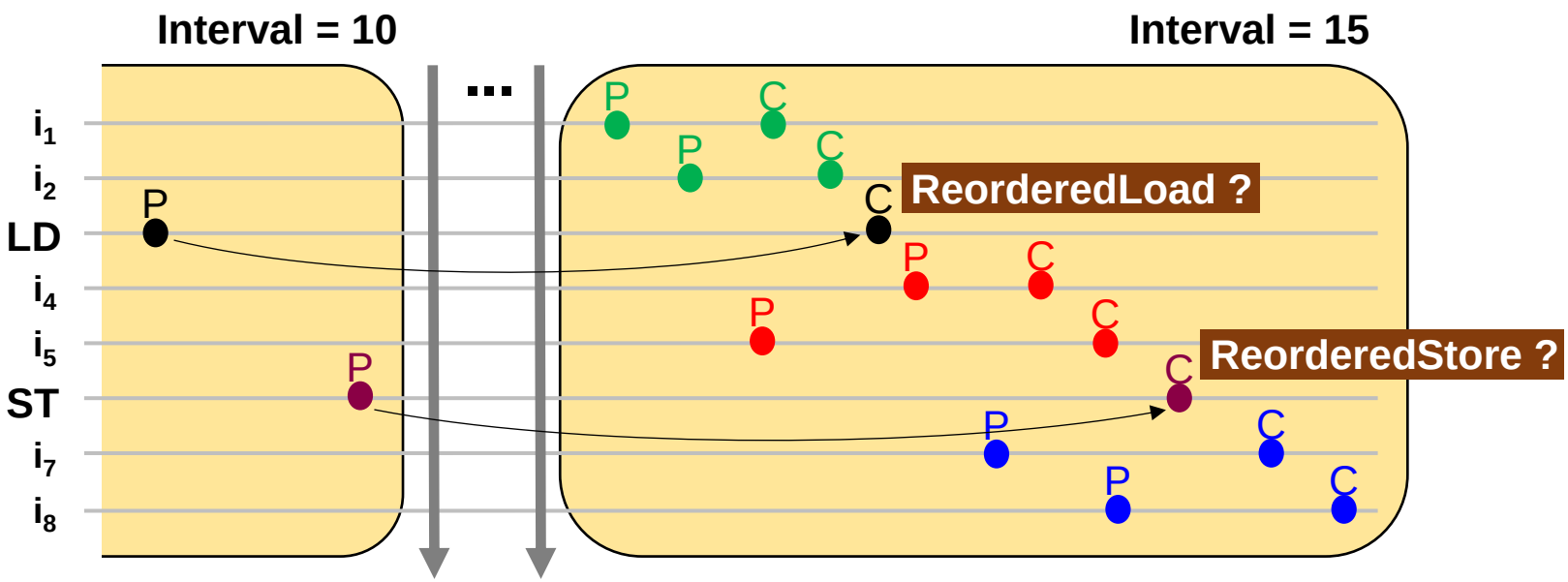
Replay Overhead

Sequential replay using OS to enforce interval ordering and emulate re-ordered instructions



- **Efficient replay: 46% (4K-OPT) and 18% (INF-OPT) overhead cycles**

Example



Base Log for Interval 15:

Block(Size=2)
ReorderedLoad(Val)
Block(Size=2)
ReorderedStore(Addr,Val,Interval 10)
Block(Size=2)

Opt Log for Interval 15:

Block(Size=8)