THE GRAINGER COLLEGE OF ENGINEERING
SIEBEL SCHOOL OF COMPUTING AND DATA SCIENCE

CS533
Hardware Support for Reliability

Josep Torrellas
Presenting: Antonis Psistakis

03/26/2026

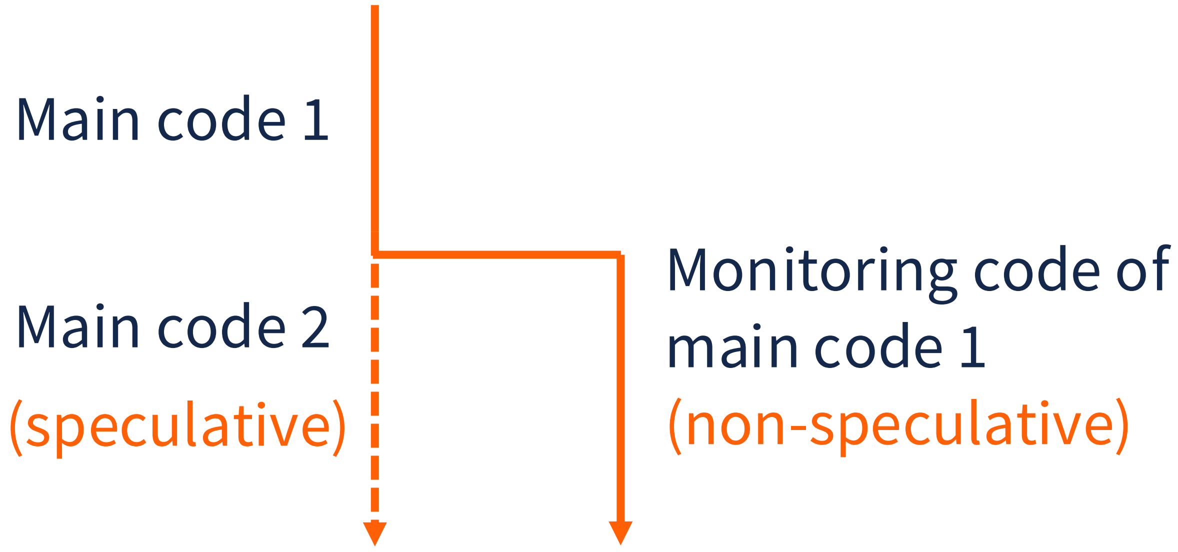
ILLINOIS

- Enhancing Software Reliability with Speculative Threads, Oplinger and Lam, ASPLOS 02
- ReEnact: Using Thread-Level Speculation Support to Debug Data Races in Multithreaded Codes, Prvulovic and Torrellas, ISCA03
- Detailed Design and Evaluation of Redundant Multithreading Alternatives, Mukherjee et al, ISCA02
- ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors, Prvulovic et al, ISCA02

Oplinger et al: Contributions

- Use speculative threads to support:
  - Efficient fine-grain code monitoring
  - Fine-grain transactional programming
- Goal: Enhance software reliability
- Rationale: We have plenty of transistors; no need to devote them all to performance

Fine-Grain Execution Monitoring



- Main code 2 assumes monitoring code will return “success”

Fine-Grain Execution Monitoring

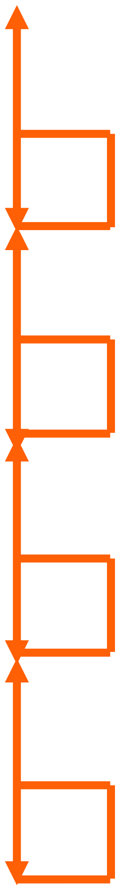
Current state of the art

- Execution monitoring has large overheads
- Not run on production codes

Proposed solution

- Low overhead
- If error caught, monitoring function communicates the failed check to the original program: return error code to the caller
- Hope to recover without terminating the program
- Typically: the speculative thread will not be rolled back

Fine-Grain Transactional Programming



All threads are speculative

All threads contain checking code at the end, which either aborts or okays the commit

Transactional model of execution

Fine-Grain Transactional Programming

- Envision that programs in the future be built as a composition of transactions
- Goal: error containment and recovery
- Current programming construct in Java:

```
TRY {
        … the original code…
        if (error-detected())
                ABORT;
} CATCH {
        return an error code;
}
return OK;
}
```

- If exception, go to CATCH, but side effects are not removed

Fine-Grain Transactional Programming

- What we want:

```
        TRY ;
        … the original code…
        if (error-detected())
                ABORT;
        COMMIT;
        return OK;
L1: return an error code;
```

- If exception, side effects are removed
- Architectural support needed to minimize overhead of transactions of finest grain

ReEnact: Using Thread-Level Speculation Support to Debug Data Races in Multithreaded Codes

Prvulovic and Torrellas, ISCA03

SM Provides Support to . . .

- Buffer state for squash or commit (caches)
- Maintain task ordering information (Task ID)
- Monitor communication across tasks to enforce ordering (cache coherence protocol)
- Cleanly undo side-effects of speculative task in a few cycles (flash-invalidate cache, restore regs)

Implementation Issues

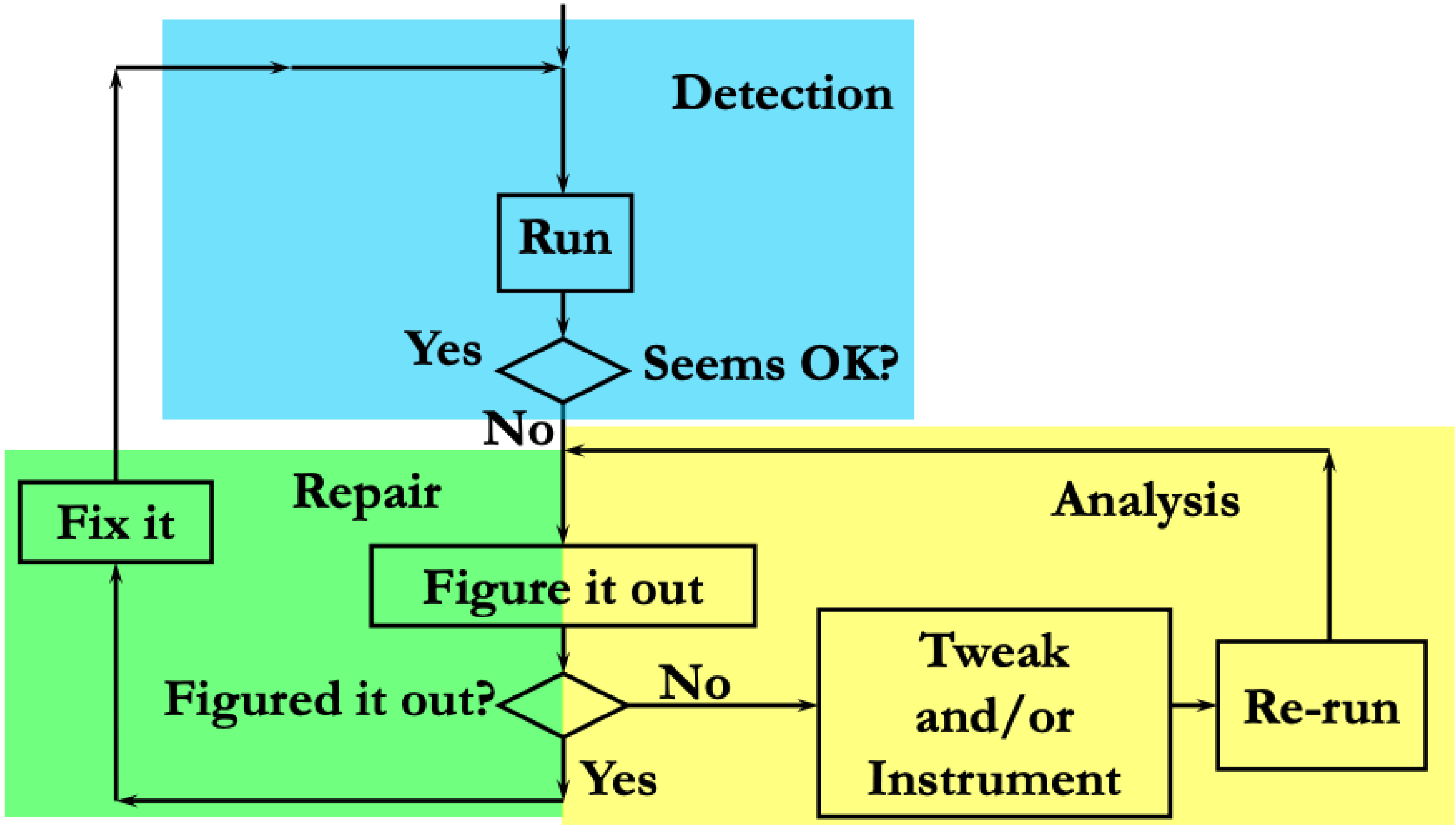
- When entering the speculative thread:
  - Fence-type instruction that creates a checkpoint of the register state
- While executing thread speculatively:
  - Buffer all memory updates in the cache --cannot update memory
  - Mark cache lines read and written
  - Monitor for errors or faults
- If an error or fault occurs:
  - Invalidate updated cache lines, reset marks, restore the register checkpoint
- Successful end of speculation:
  - Allow updated cache lines to be committed (displaced to memory)

ReEnact

- Enhance Software Debugging:

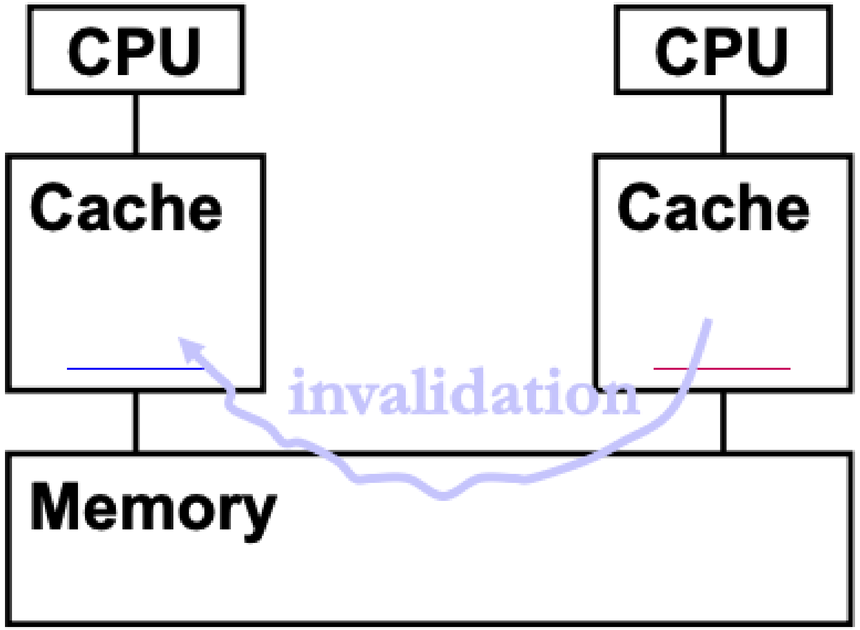
  Detect + Characterize + Correct software bugs in programs on the-fly automatically in production codes

Conventional debugging

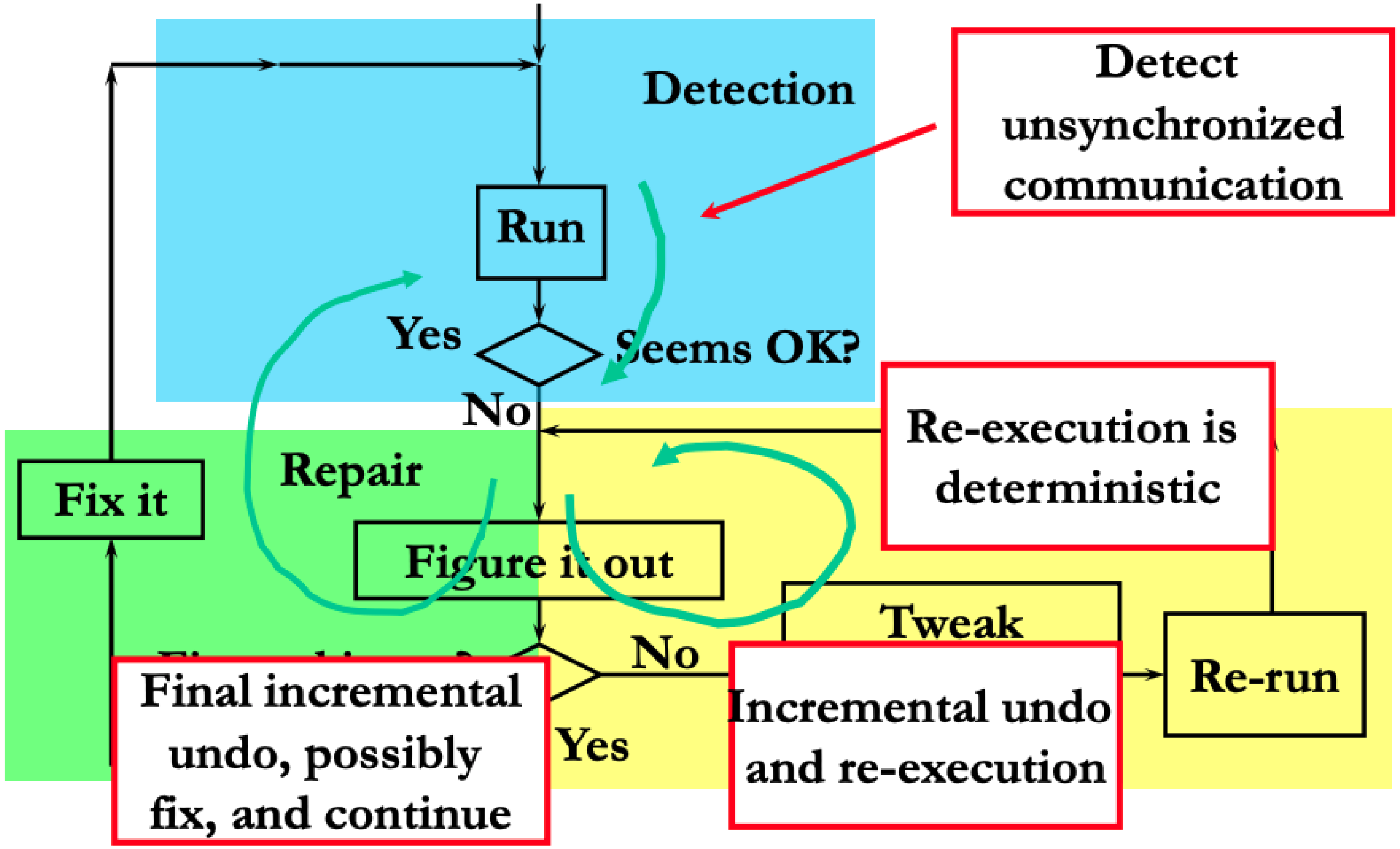


What ReEnact Provides

- Cleanly undo group of tasks (buggy code section, hopefully)
- Re-execute those tasks only
- Re-execution of tasks is deterministic even under multithreading
- Bonus: detect bugs that appear as communication (e.g., **Data Races**)



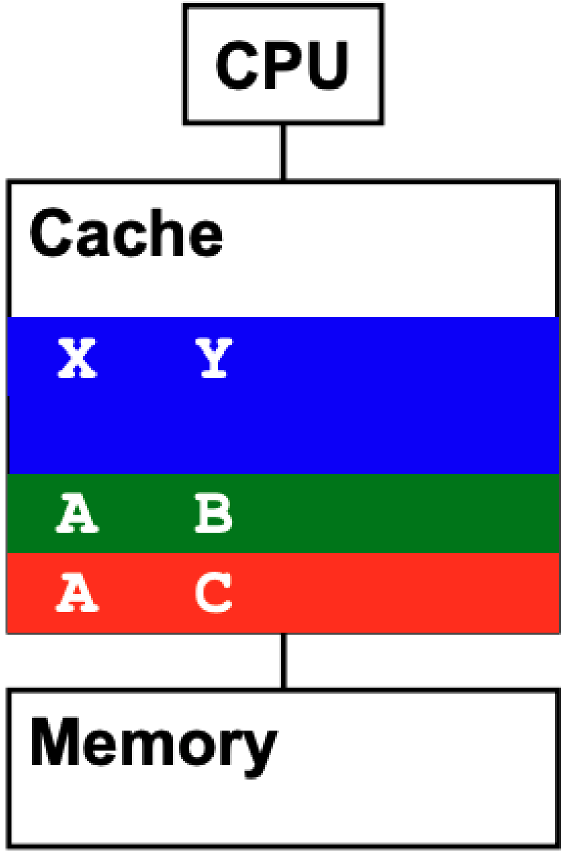
Enhancing Debugging



Breaking Code into Chunks

Dynamic Instructions

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



Breaking Code into Chunks

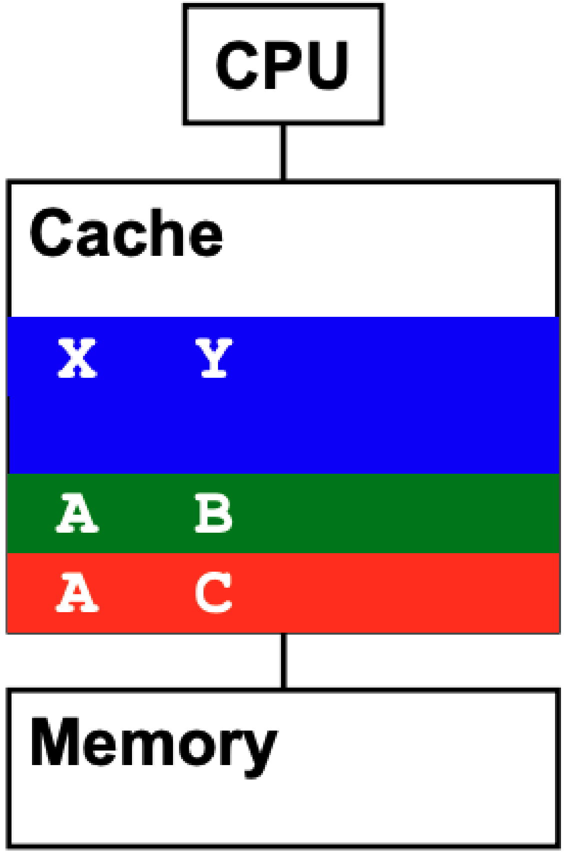
- New chunk begins → save CPU state
- Undo (squash) recent chunks if needed
  - Invalidate cache lines, restore saved CPU state
  - Enables re-execution for analysis and repair
- Commit old chunks
  - Allow displacement from cache, free saved CPU state
  - Makes room for buffering more recent chunks
  - Can not undo committed chunks

Undo Chunks

Dynamic Instructions

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

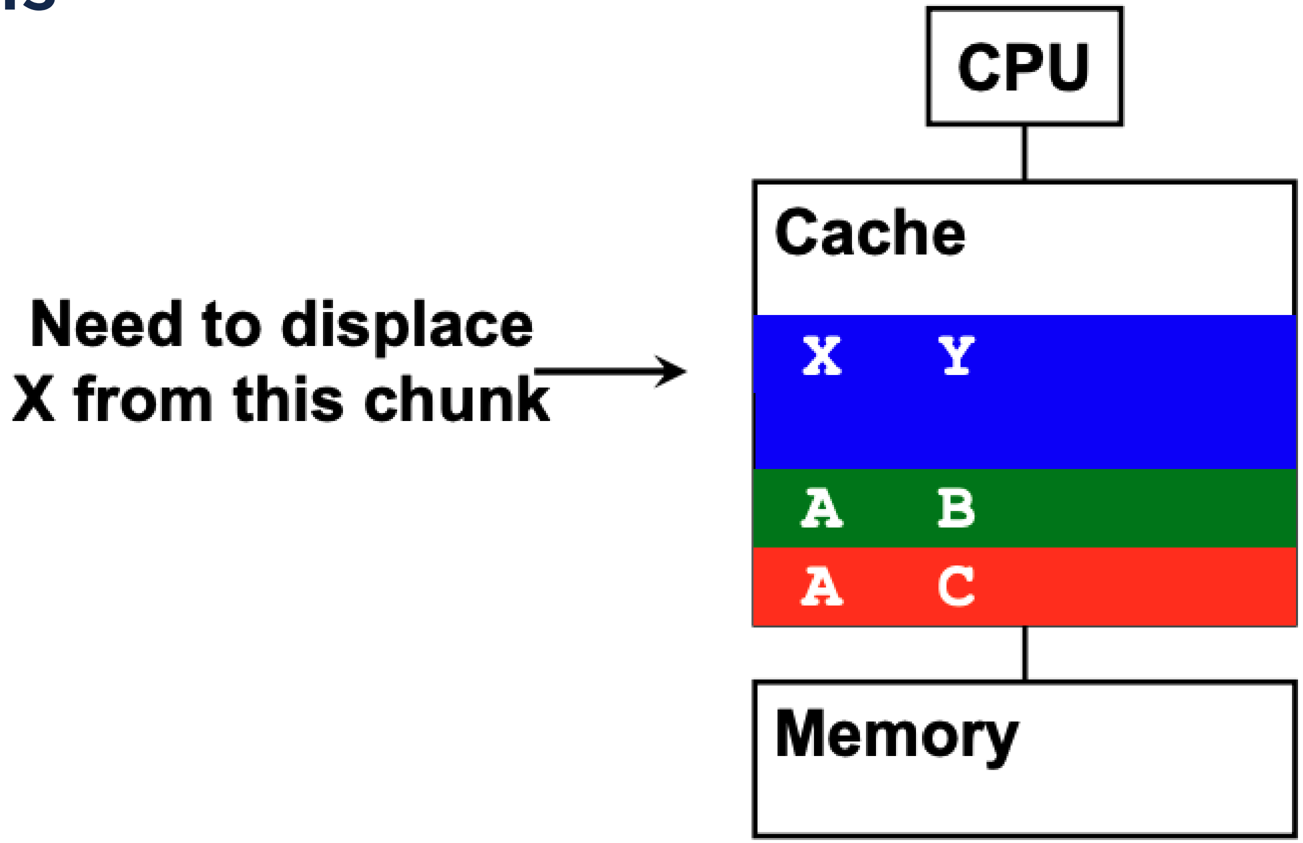
Analysis requires a rerun of these chunks



Chunk Commit

Dynamic Instructions

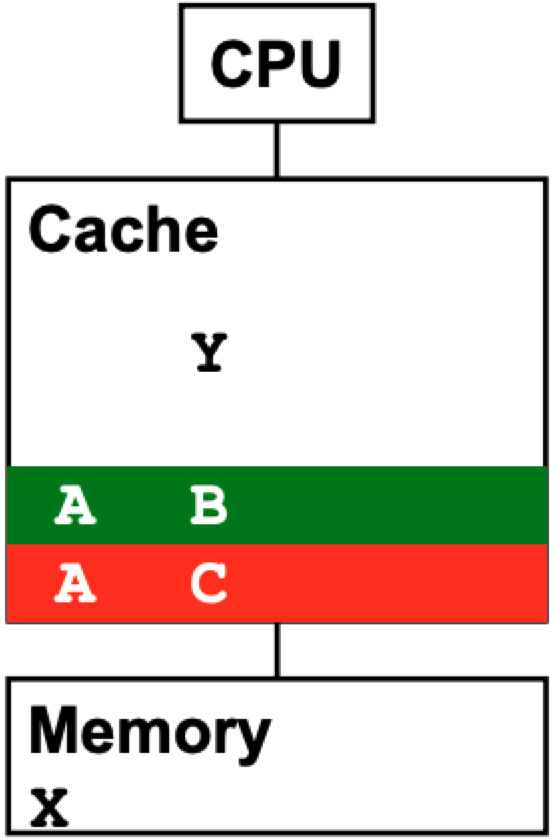
ST X
ST Y
ADD
...
ST A
ST B
ST A
...
ST A
ST C



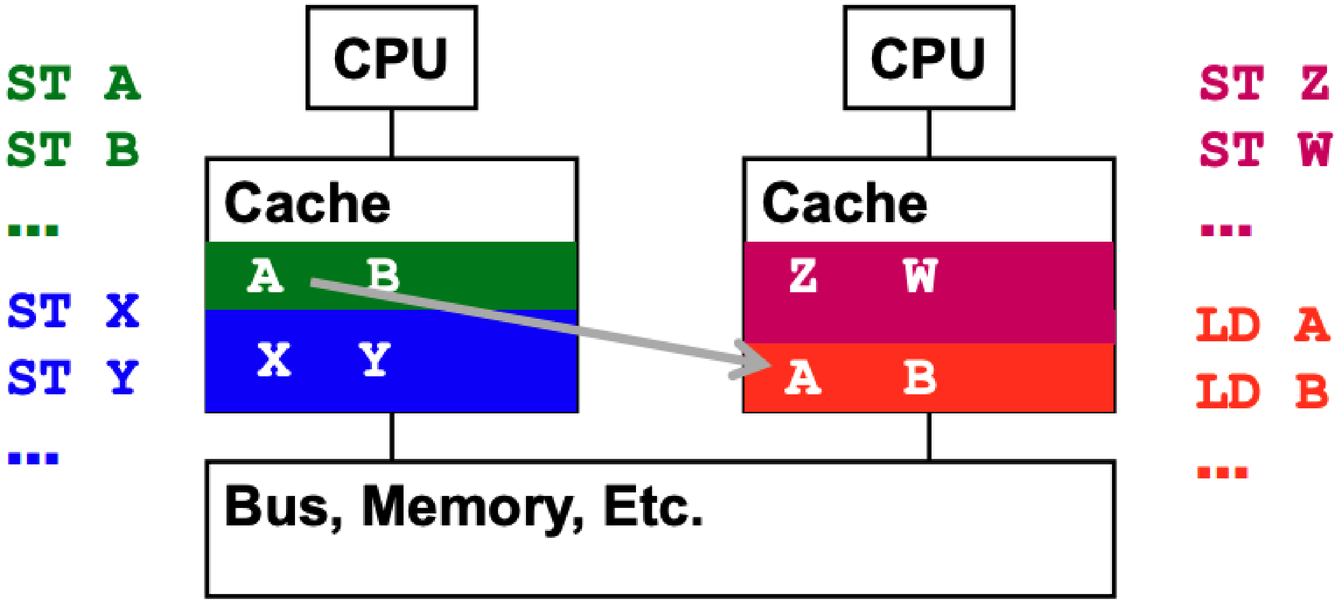
Chunk Commit

Dynamic Instructions

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



Ordering Chunks

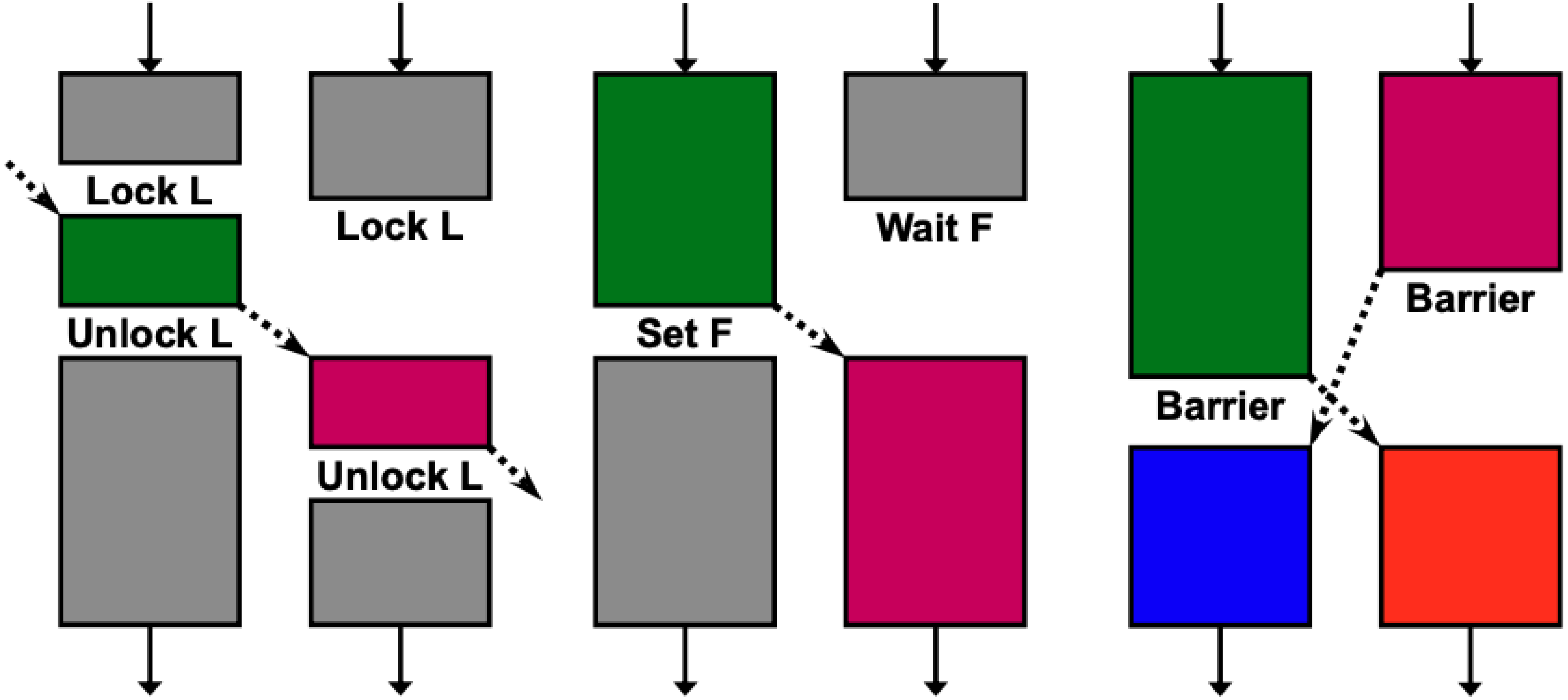


Chunk “happens before” chunk

Deterministic Re-execution

- Cross-thread communication → chunk order
  - Order chunks on first communication
  - Enforce order on subsequent communication (may squash)
- Entire chunk ordered before or after another
- Deterministic re-execution → repeat chunk order

Chunk Ordering by Synchronization



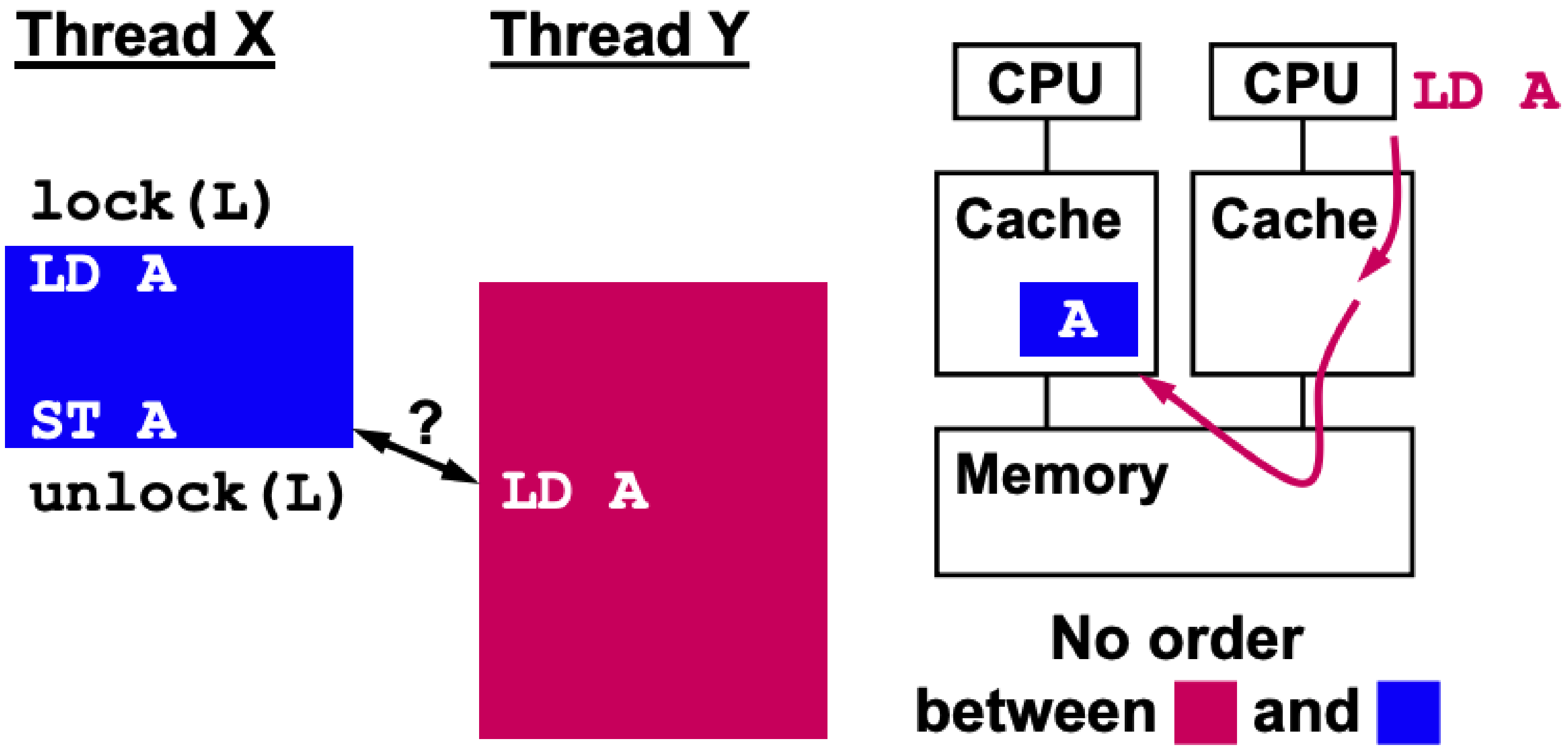
Data Race Detection

- Detect communication between…
  - **Ordered** chunks: synchronized access
  - **Unordered** chunks: data race detected

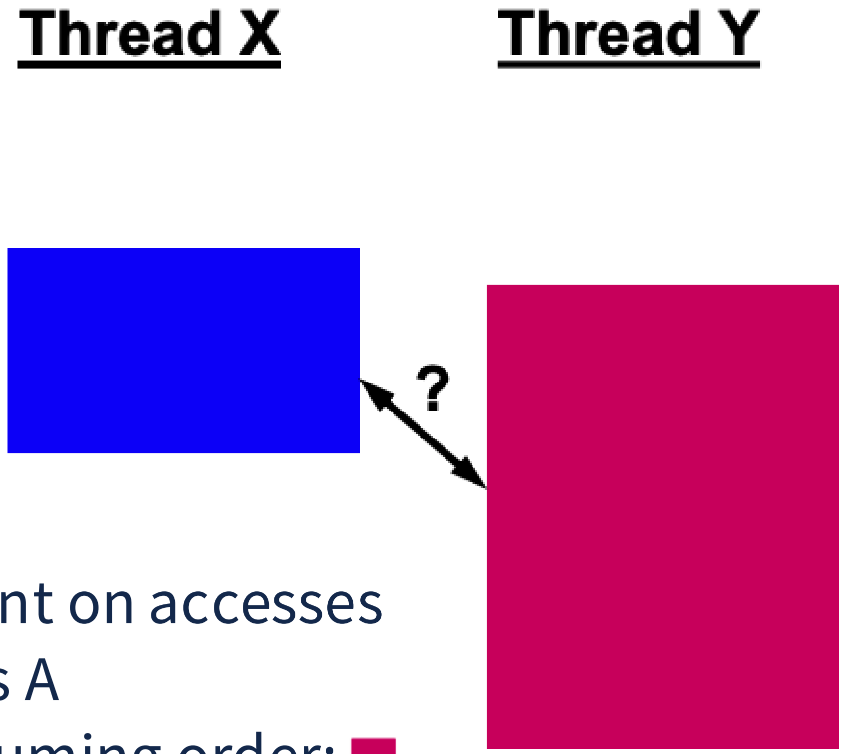
Example: Missing Critical Section

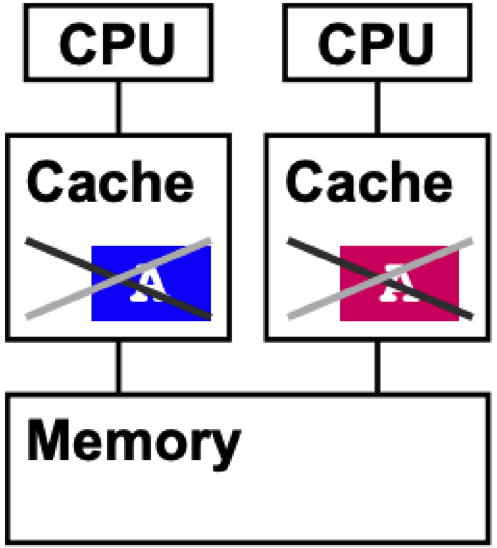
Thread X	Thread Y
`lock(L)`	
`LD A`	
`ST A`	
`unlock(L)`	
	`LD A`
	`ST A`

Detection: Data Race



Analysis: Refine Race Signature





1. Rollback
2. Put a watchpoint on accesses to data address A
3. Re-execute assuming order: after

Found Race Signature

<u>Thread X</u>	<u>Thread Y</u>
LD A	
ST A	
	LD A
	ST A

Repair: Pattern Matching

- Analysis resulted in a detailed signature
  - Instruction & data addresses, data values, timing, etc.
- Pattern-match it with a library of common races:
  - Suggest repair to programmer, or
  - Download bug-specific patch, or
  - Try to automatically re-introduce missing ordering
- Squash chunks, re-execute with corrections

ReEnact Pros

- On the fly – debug each problem as it is found
  - Low-latency re-execution of surrounding code
  - Low-latency detection of bug symptoms
- Always on – even in production code
  - Low overhead in bug-free execution
- Debug multi-threaded code
  - Must address non-determinism of parallel execution

ReEnact Evaluation

- Low overhead in error-free execution: 6% avg
- Highly effective: Detect, Analyze, and Correct (?) SW bugs
  - Existing bugs
    - *Synchronization through plain variables*
    - *Other existing data races*
  - Induced bugs
    - *Remove critical section*
    - *Remove barrier*

How Good ReEnact is to:

	Detect	Rollback	Analyze	Match
Sync through plain variables	✓	✓	✓	✓
Other Existing Data Races	✓	✓	✓	No
Induced Bugs: Removed Lock	✓	✓	✓	✓
Induced Bugs: Removed Barrier	✓	~	~	~

Conclusion

- Speculative multithreading boosts software productivity
  - Enhances **Software Debugging** (detection, analysis, correction)
- Exciting area: **cost effective use of transistors** for reliability or debuggability

Thank You

Questions?