Determinism and Fail-stop Races for Sane Multiprocessing

Luis Ceze, *University of Washington*

joint work with Owen Anderson, Tom Bergan, Joe Devietti, Nick Hunt, Brandon Lucia, Jacob Nelson, Steve Gribble, Dan Grossman, Mark Oskin, Karin Strauss, Shaz Qadeer and Hans Boehm.

sampa ***Safe MultiProcessing Architectures at the University of Washington***

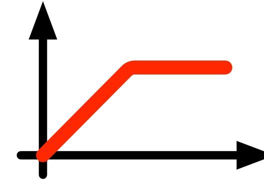
NWCPP meeting, Jan 2011.

me/sampa: multiprocessor architecture, compilers, programming models, memory models, determinism, OS, concurent software reliability, hw/sw for low-power

You guys know a lot more about C++ and large software projects than I do...

You probably heard this many times

- Era of “free” performance is over.



- Most of compute power now scaling in terms of cores
  - Mostly due to power and complexity reasons. Copy & Paste in VLSI :)
- Shared memory is most popular
  - Within a box
  - Simplifies data movement, makes synchronization harder
  - Shared memory vs. Message passing almost a religious argument
- This talk:
  - Shared-memory multiprocessors. Bringing more safety and sanity to parallel programming.

A multithreaded voting machine

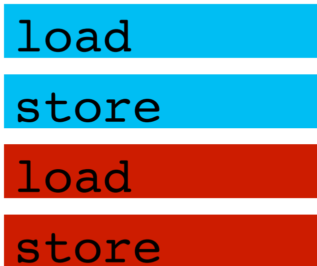
shared variable

thread 0

```
while (more_votes) {
   load t <- votes
   t++
   store t -> votes
}
```

thread 1

```
while (more_votes) {
   load t <- votes
   t++
   store t -> votes
}
```



votes == 2

votes == 2

votes == 1

Two Key Problems

- Data races
  - deep impact on memory model and language semantics (see JMM), pretty much all languages converging to data-race-free models
  - usually incorrect and hard to debug
  - reliability issues: surprising software failures
- Nondeterminism
  - debugging is hard: heisenbugs
  - testing is hard: can't test each input just once
  - fault tolerant replicas might not behave the same way
  - opens timing-based security attacks
- Note: these two are orthogonal

What if...
We Made Data-Races Fail-Stop?

Semantics are clear and simple

Better data-race debugging

Safety: races can't cause problems

When a data-race occurs, throw an exception!

(we have div by 0, segfault, why not concurrency errors?)

Can we provide **strong detection guarantees** at a low cost?

What if...
We Removed Non-determinism?

- Development: bugs are **reproducible by default**, **test** each input only **once**
- Deployment: software behaves as tested, enables **replication** for fault tolerance, timing-based attacks harder

Effectively, make **arbitrary parallel** programs behave like **sequential** programs...

Can we remove undesired nondeterminism without removing performance?

An aside on memory consistency models and the C/C++ standard model.

What is a Memory Consistency Model?

- Define what values a read can return in shared-memory programs
  - What values do you expect the loads below to get? (x,y both start with 0).

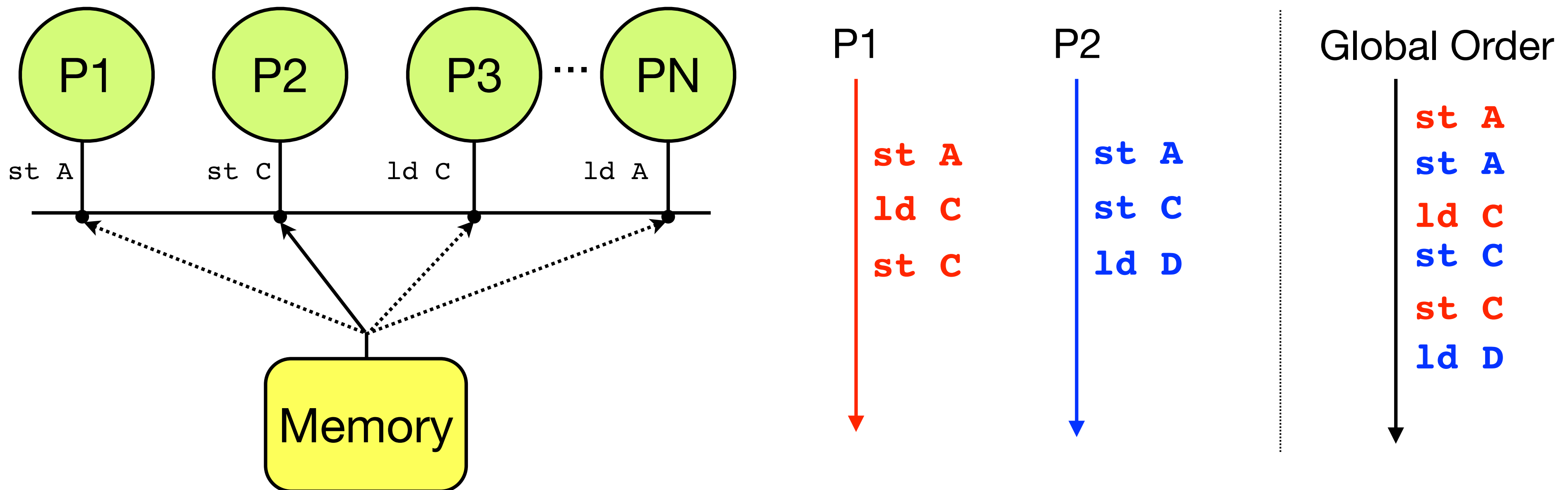
thread 1	thread 2
ld x ld y	st 1 → y st 1 → x

How about (1,0)?

thread 1	thread 2
st 1 → y ld x	st 1 → x ld y

How about (0,0)?

Sequential Consistency (SC)



Per-processor program order: memory operations from individual processors maintain program order

Single sequential order: the memory operations from all processors maintain a single sequential order

[Lamport'79]

Sequential Consistency Implications

- What are the implications of that to:
  - compiler optimizations?
  - hardware optimizations?
- Conclusion:
  - Need to give freedom to compiler writers and HW designers
  - Perhaps at the cost of your sanity :)
  - Many “relaxed” models: TSO (x86), Weak Ordering (PPC/ARM), etc.

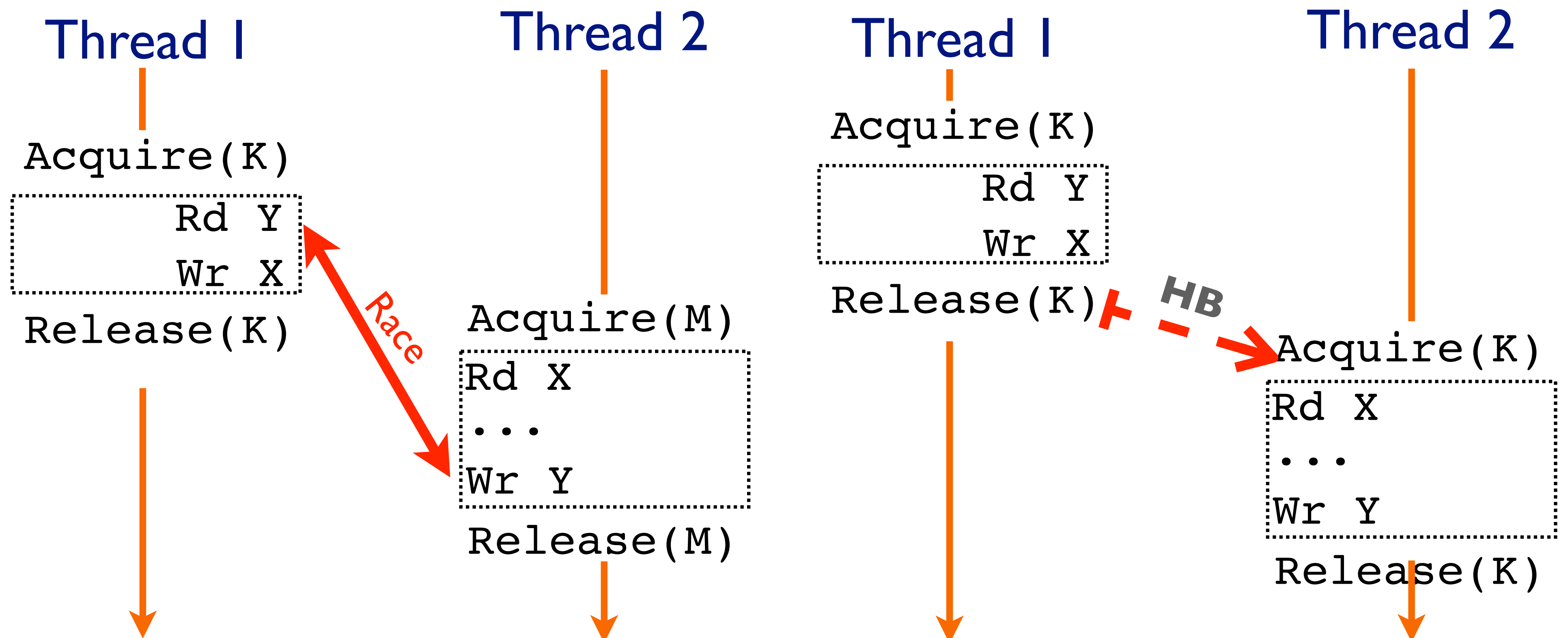
C/C++ Standard on Memory Model

- Sequential Consistency...

- for **Data-Race Free** programs

What is a data-race?

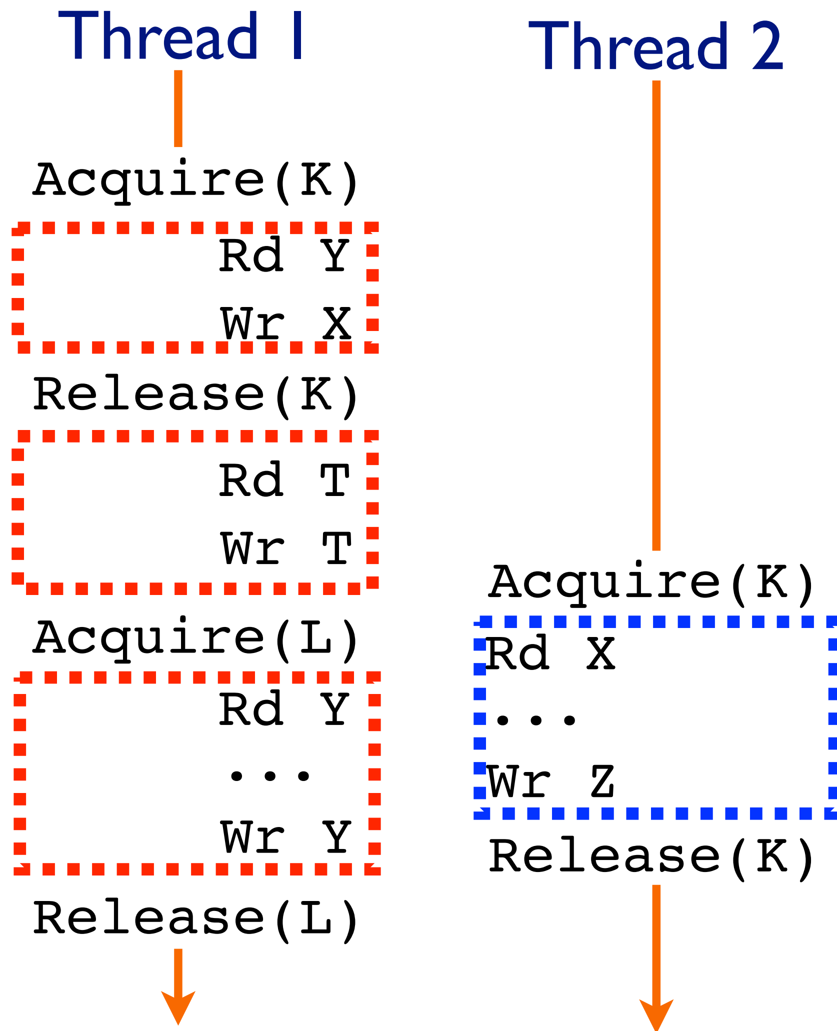
- Many “intuitive” definitions
  - *One technical definition*: two accesses from different threads; at least one a write; accessing the same location; without explicit happens-before ordering via synchronization

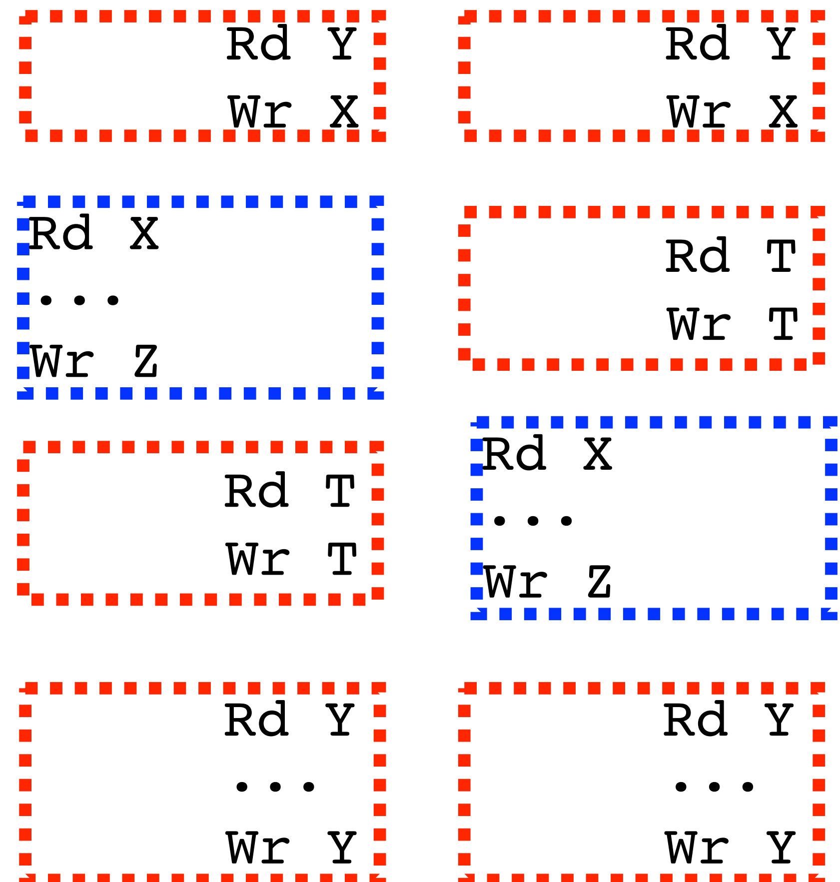


C/C++ Standard on Memory Model

- Sequential Consistency for **Data-Race Free** programs
- What does that mean?
  - If execution of a program has no races, you can reason about it in a sequentially consistent way
  - And execution behaves as some interleaving of regions without synchronization operations

Sequential Consistency for DRF Example





C/C++ Standard on Memory Model

- What does that buy?
  - A *lot* of freedom to compiler and hardware
    - e.g., HW buffers, loop-inv code motion, CSE, etc.
  - Pretty much can do whatever reordering as long as it does not cross synchronization
- Key is to determine if there is a race...
  - **very** hard to do statically

Concurrency Exceptions: The Vision

- Concurrency bugs drive people nuts
  - Show asynchronous, non-local behavior
  - Often lead to silent failures
  - **Significantly** complicate language semantics

➡Generate an exception when a concurrency occurs
  - Put them in the same category as Div-by-zero, SEGFAULTs, etc
  - Which concurrency errors? When should the exception be delivered? To what threads?

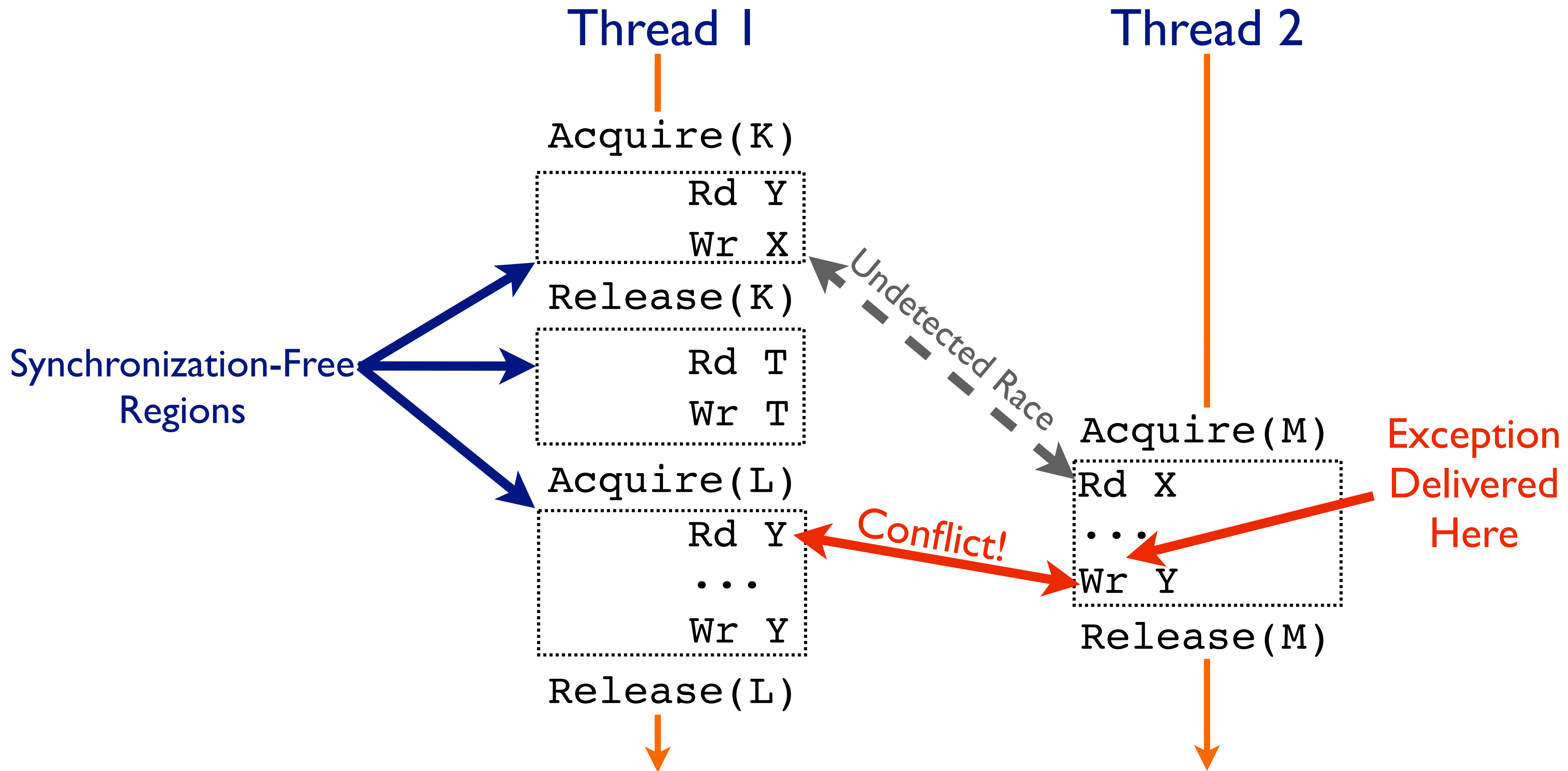
- We are starting with data-races
  - Well defined, doesn't require programmer annotations, language semantics

Goals In Supporting Races as Exceptions

High-Performance - Always-on detection

Precise detection - No false positives

Conflict Exceptions [ISCA'10]



Conflict Exceptions

Ignoring "unimportant" races is key to performance

(much lower space and time overheads)

Precisely detect only races that can effect consistency

The Guarantee:

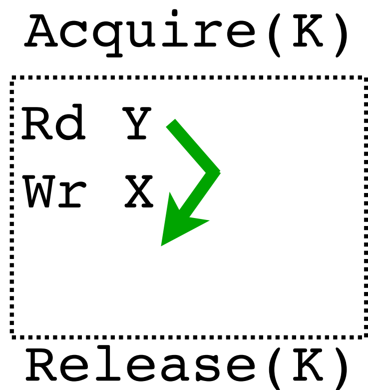
Exception-Thrown? There was a data-race.

Exception-Free? Sequential Consistency.

(dramatically simplifies checking, while making PL and systems people happy :).

Language Level Benefits

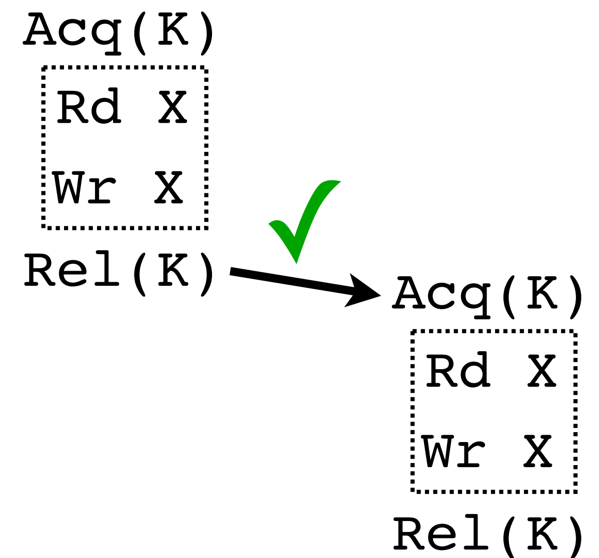
Reordering in SFRs is legal




```
Acquire(K)
Wr64_Low X
Wr64_Hi X
Release(K)
```

Granularity independence

Exception-Free executions are SC



Language Level Benefits

Programming model is largely the same

```
pthread_lock(K)
  Rd Y
  Wr X
  Wr Q
  Wr Z
pthread_unlock(K)
```

```
Acq(K)
  Rd X
  Wr X    !
            Acq(L)
              Rd X
```

Racy programs are well-behaved

Race semantics are simpler

w such that $w.v = r.v$ and $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$.

5.4 Causality Requirements for Executions

A well-formed execution

$$E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$$

is validated by *committing* actions from A. If all of the actions in A can be committed, then the execution satisfies the causality requirements of the Java memory model.

Starting with the empty set as C_0, we perform a sequence of steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E_i containing C_i that meets certain conditions.

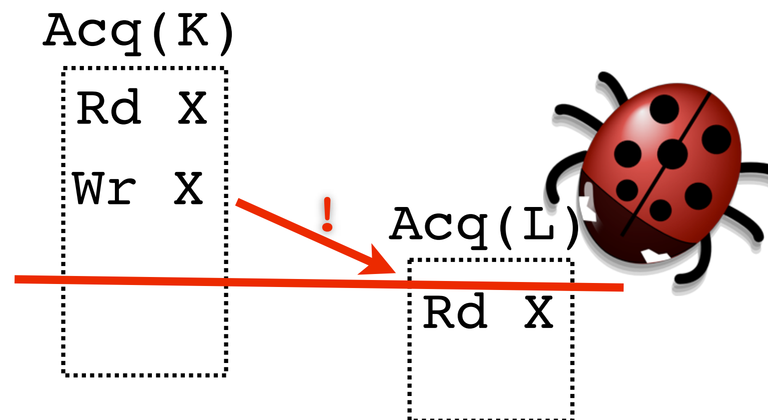
Formally, an execution E satisfies the causality requirements of the Java memory model if and only if there exist

- Sets of actions $C_0, C_1, \ldots$ such that
  - $C_0 = \emptyset$
  - $C_i \subset C_{i+1}$

Debugging and Reliability

Concurrent, conflicting SFRs *throw exceptions*

All races have some exceptional schedule



Exception Handling: Log + Recover

Damage Control: Shut down buggy module

Hardware Support in a Nutshell

Hardware Transactional Memory

- Versioning

+ Byte-level conflict detection

+ Exception support

Hardware/Software Interface

New Instructions:
`BeginRegion` and `EndRegion`

Synchronization Operations are Singleton Regions

Exceptions Thrown Precisely Before Conflicting Instruction

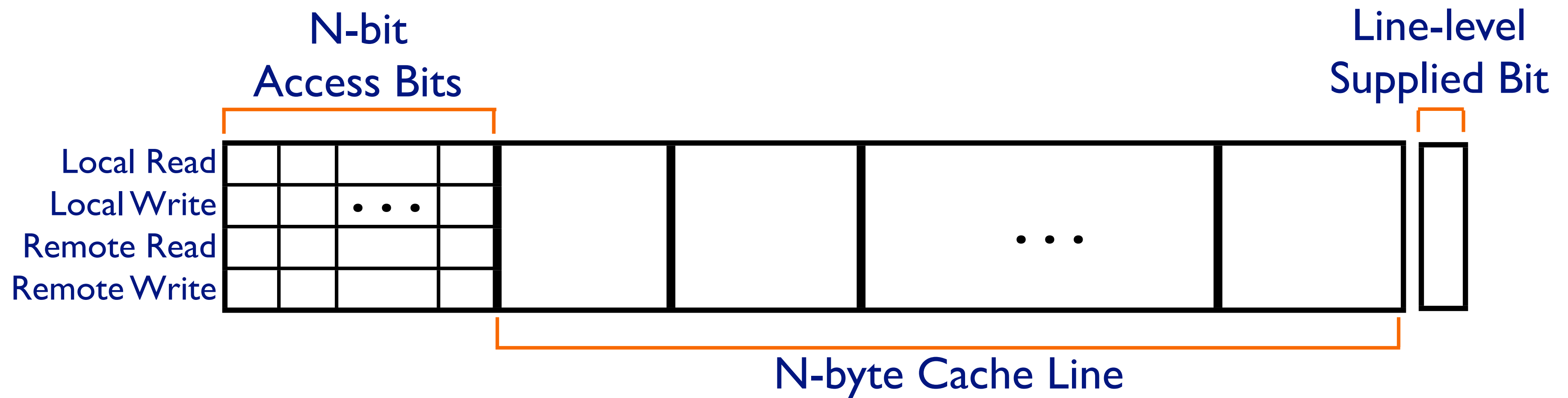
```
Acquire(K)

BeginRegion
        Rd Y
        Wr X
EndRegion

Release(K)

BeginRegion
        Rd T
        Wr T
EndRegion
```

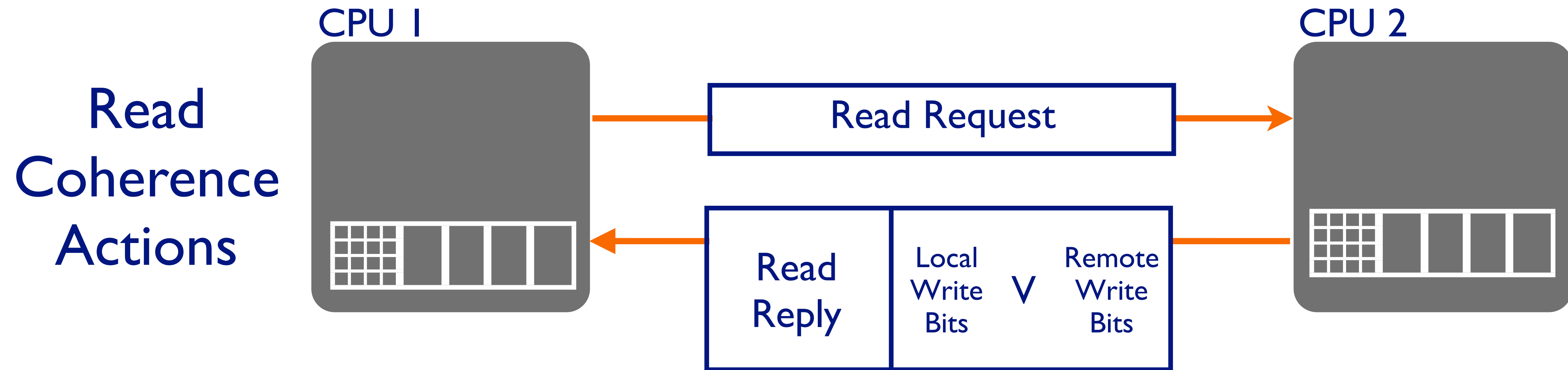
Access Monitoring

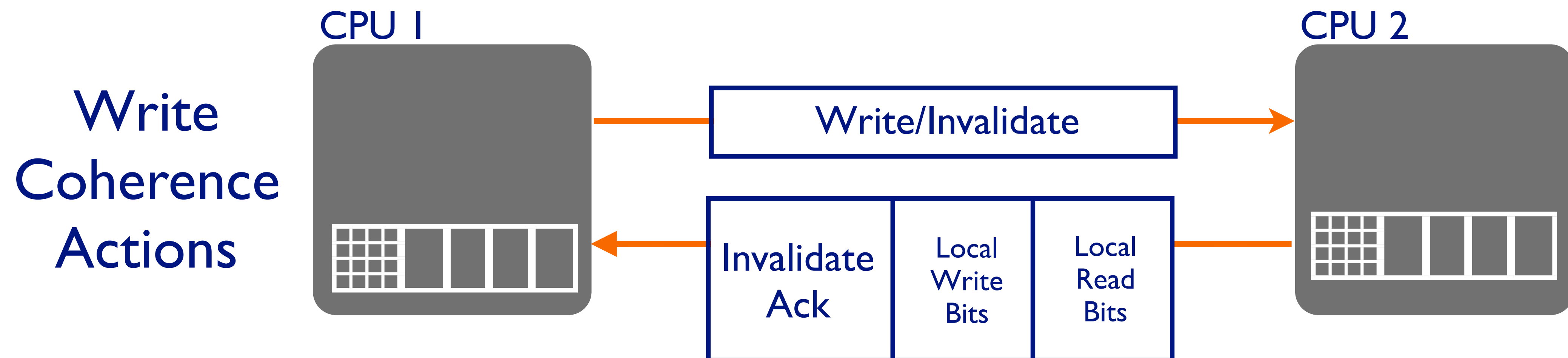


Exception Test: compare local and remote bits

Overheads significantly reduced via type-safety and reusing data-array for access bits. [ISCA'11sub]

Leveraging Coherence Support





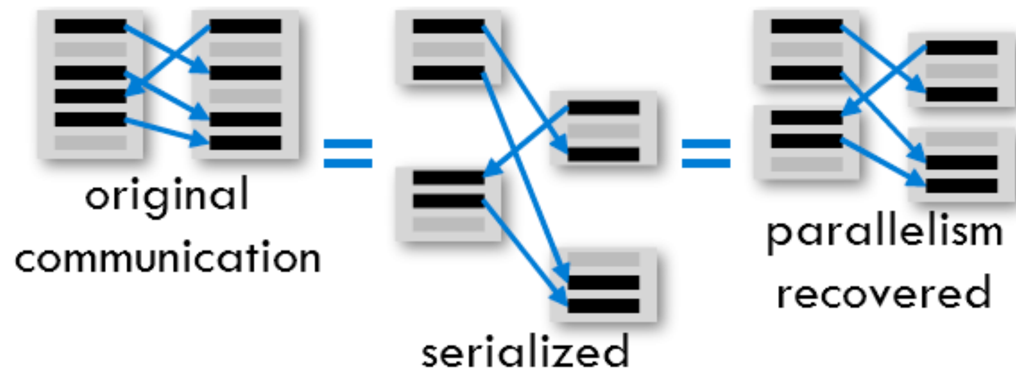
Now that we know how to get SC executions (or an exception)....

Deterministic Multiprocessing

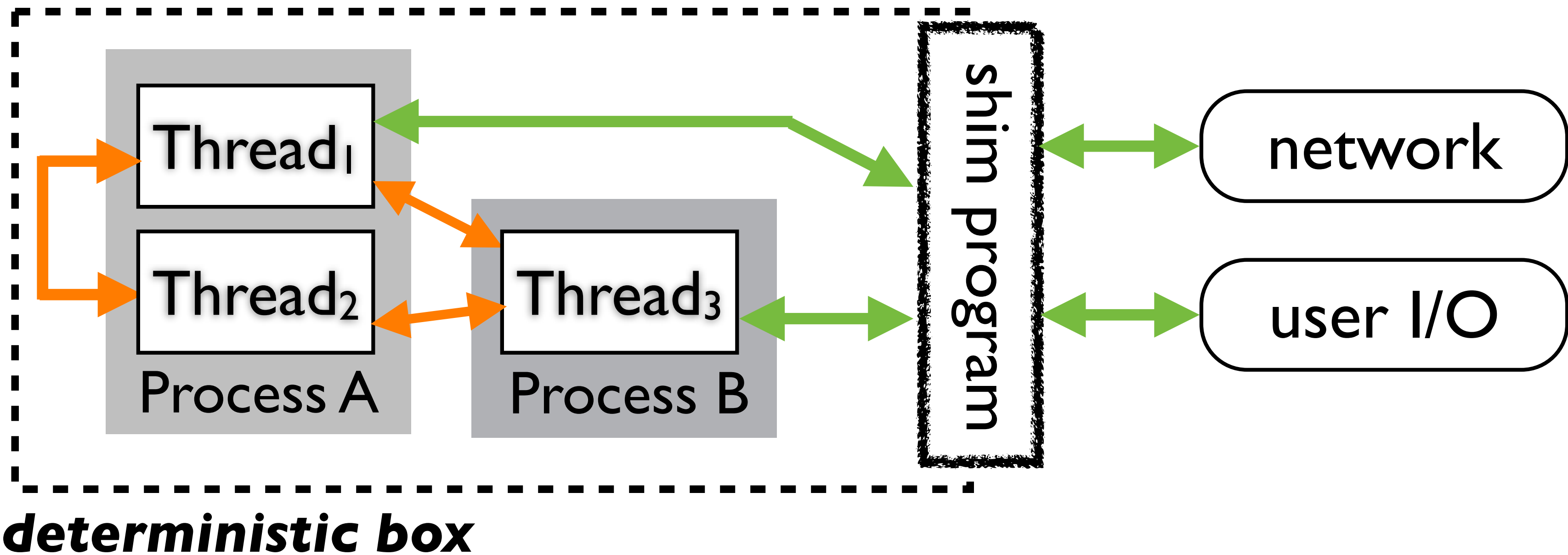
Deterministic Multiprocessing at 10,000’

[ASPLOS’09, ASPLOS’10, OSDI’10, ASPLOS’11]

- DMP provides *execution-level* determinism for arbitrary multithreaded programs:
  - execution is only function of explicit inputs => single execution per input
  - this is not record-replay of multithreaded programs
- ***Key idea***: conceptually serialize execution, recover parallelism while preserving serial execution semantics
  - several techniques to make this fast: actual goal is to preserve inter-thread communication, still freedom left for efficient schedules



Deterministic Process Groups (DPGs)



System ensures:

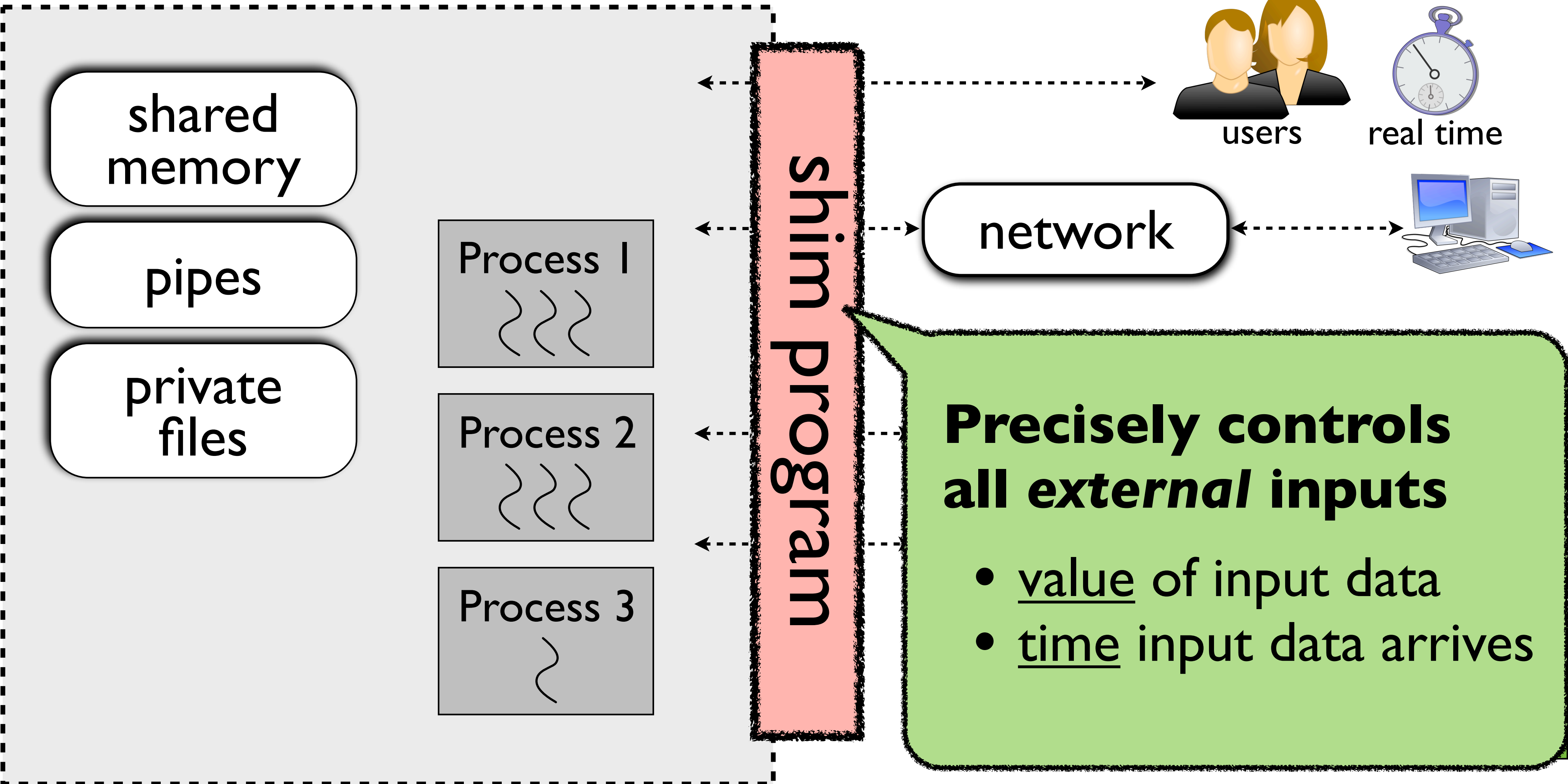
- internal nondeterminism is eliminated
  (for shared-memory, pipes, signals, local files, ...)
- external nondeterminism funneled through *shim program*

Shim Program:

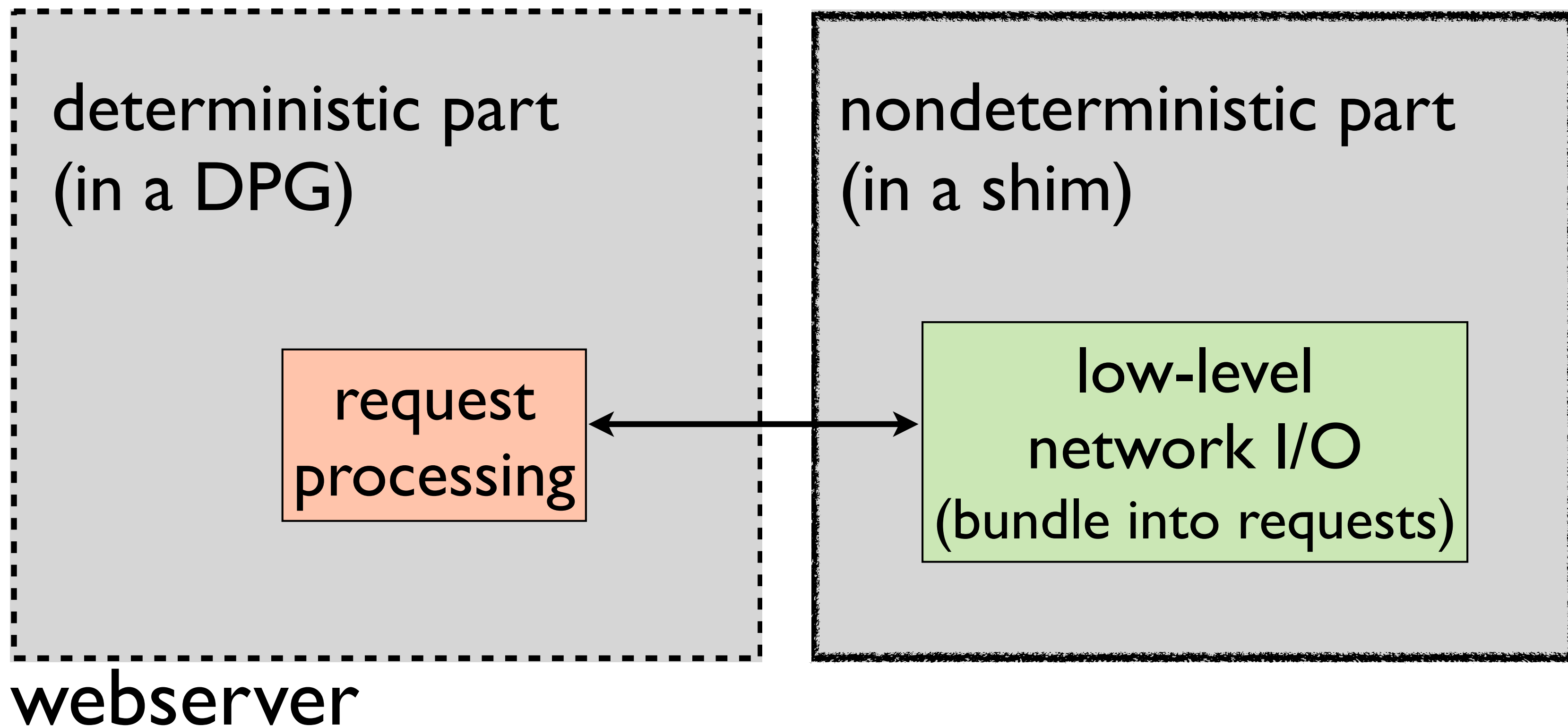
- user-space program that precisely *controls* all external nondeterministic inputs

Internal Determinism

External Nondeterminism



Aside: Using DPGs When Constructing Apps



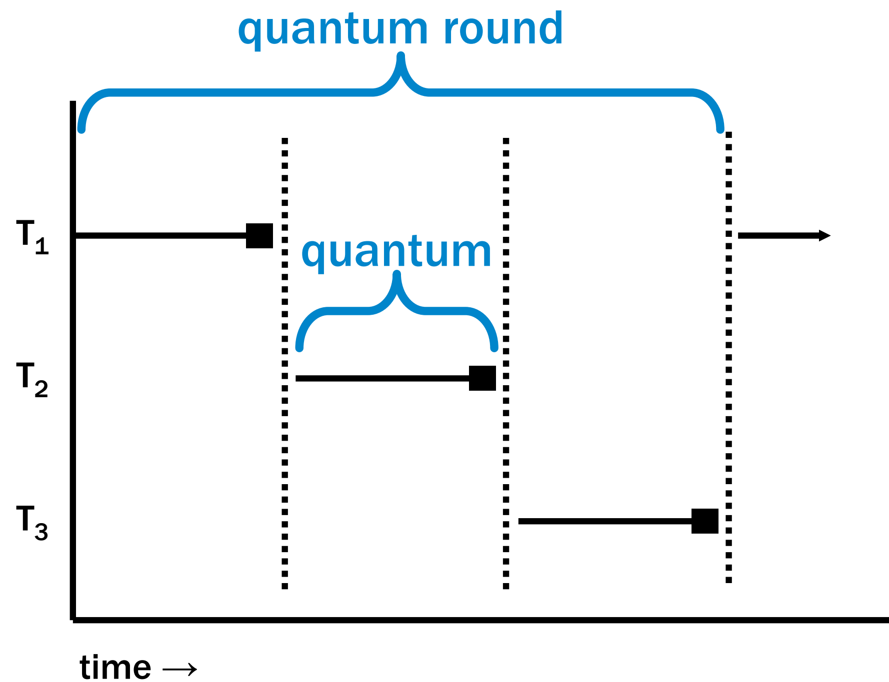
webserver

- behaves deterministically w.r.t. *requests* rather than *packets*

Shim program defines the nondeterministic interface

How is determinism actually enforced?

Starting simple: DMP-Serial



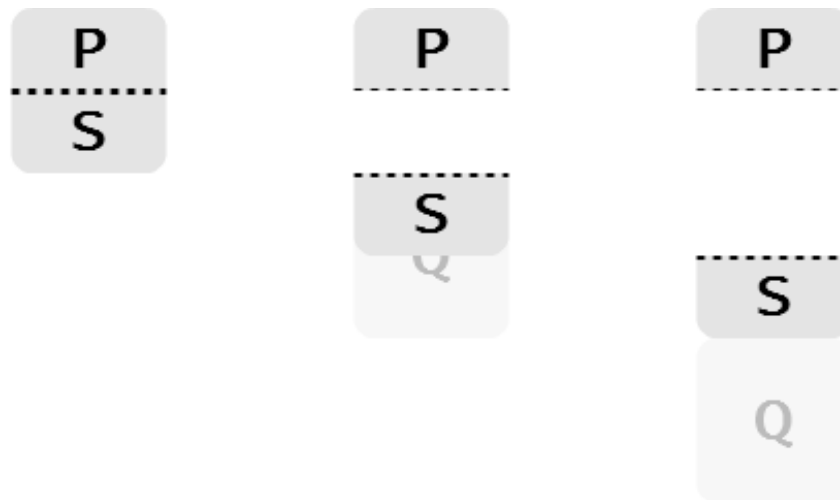
deterministic quantum size
(in logical time, e.g., instructions)
+
deterministic scheduling

---

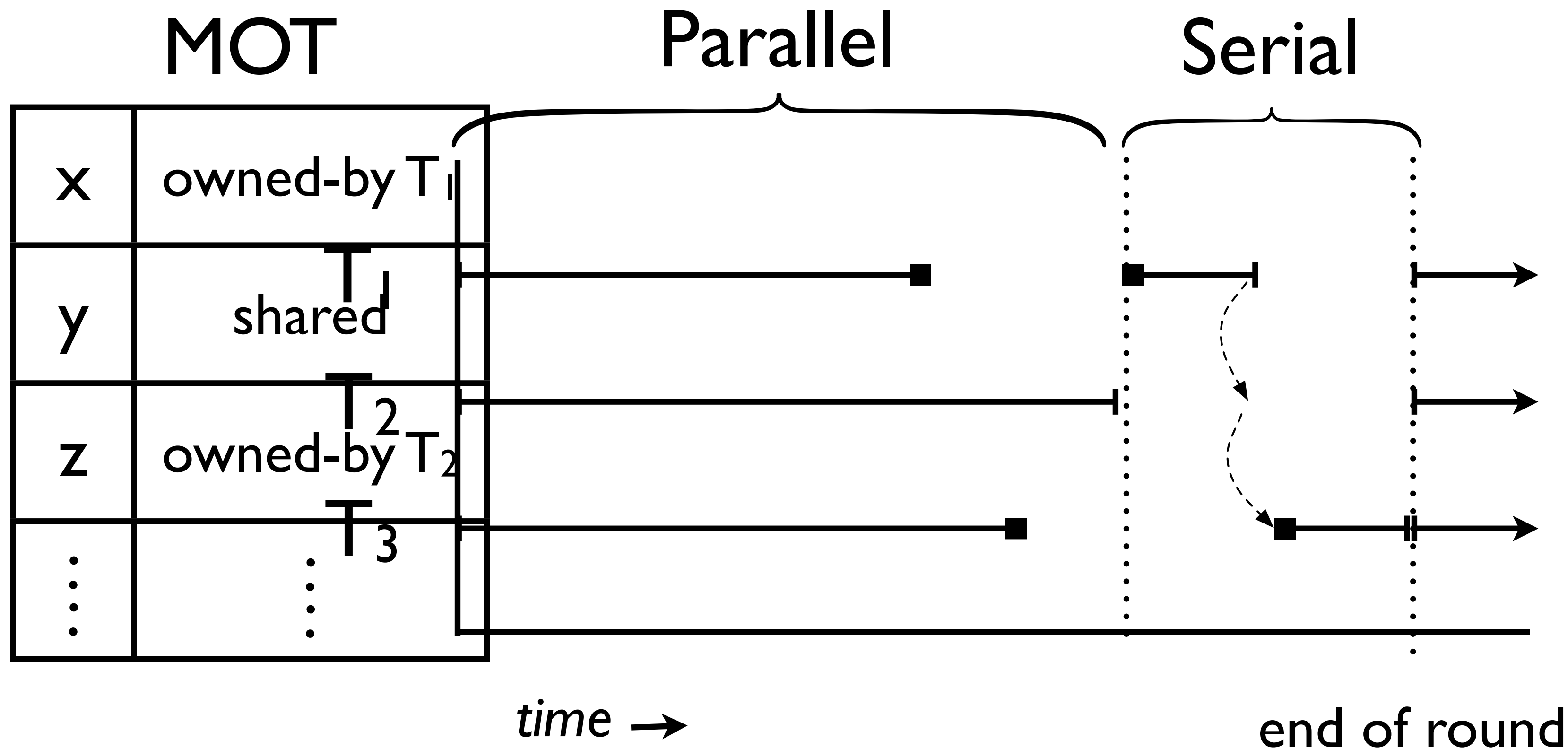
determinism

Can we do better?

- Only need to serialize communicating instructions
- Break each quantum into communication-free parallel mode and communicative serial mode
- Need to know when communication happens
  - The **Memory Ownership Table (MOT)** tracks information about ownership



DMP-O (Ownership)



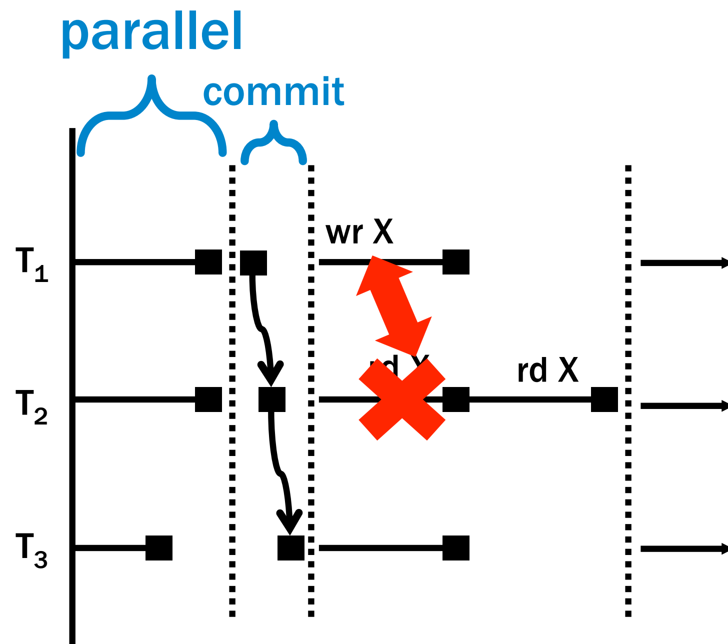
Parallel mode: no communication (can write only to private data)
Serial mode: arbitrary communication

Important: State of the MOT needs to evolve deterministically; updates are limited to serial suffix

DMP-TM: Recovering Parallelism with Speculation

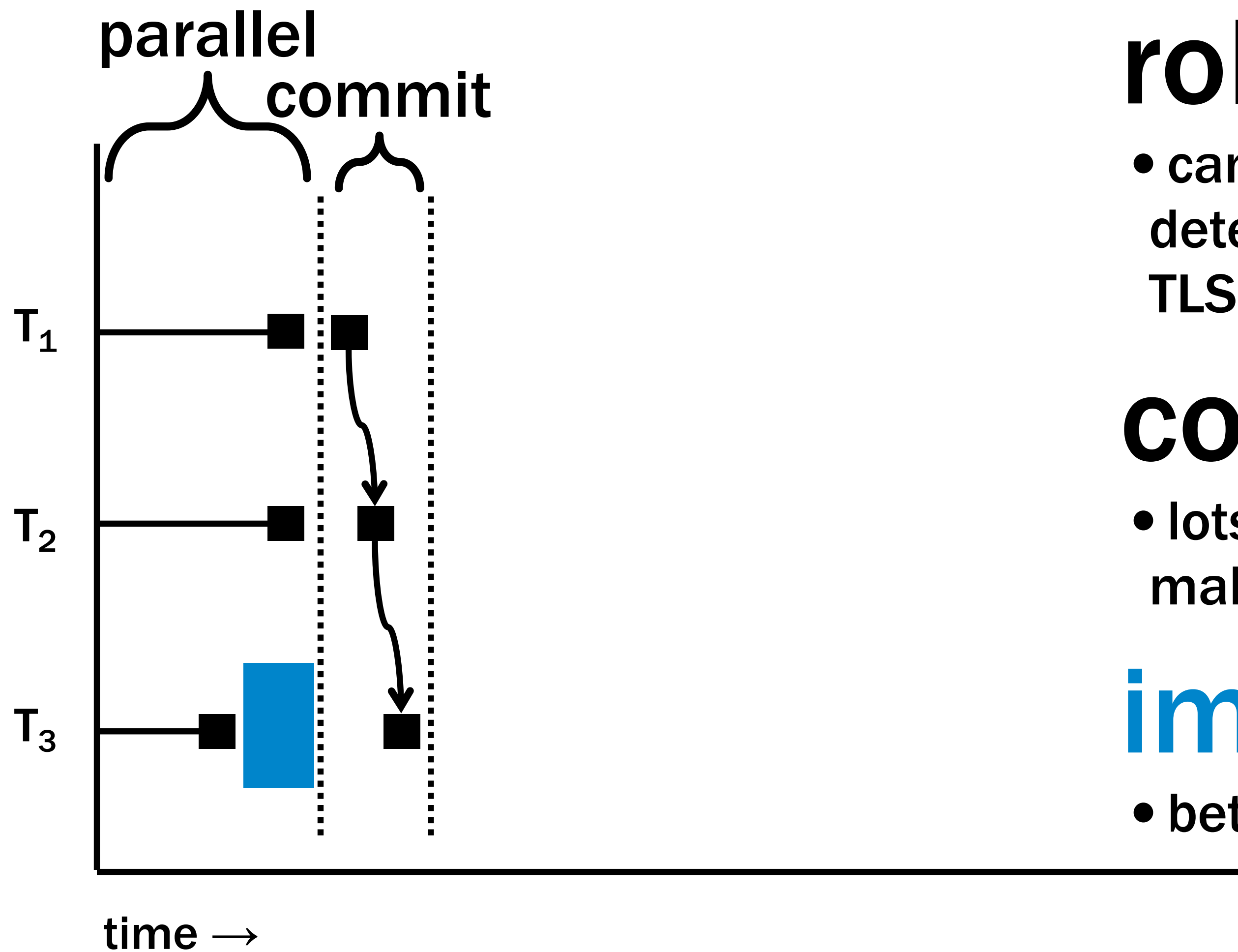
- DMP-O conservatively assumes that all cache line state transitions are communication
  - …but **many transitions are not communication**
- Use TM support to speculate that a quantum is not involved in communication
  - If communication happens, rollback + re-execute
  - Commit quanta in a deterministic order

DMP-TM



- quanta are implicit transactions
- commit quanta in deterministic order
- rollback+restart on conflicts
- leverage (best effort) HTM support
- functionally equivalent to DMP-Serial

DMP-TM Overheads



rollbacks

- can use relaxed conflict detection like TLS & other TLS tricks like forwarding

commit

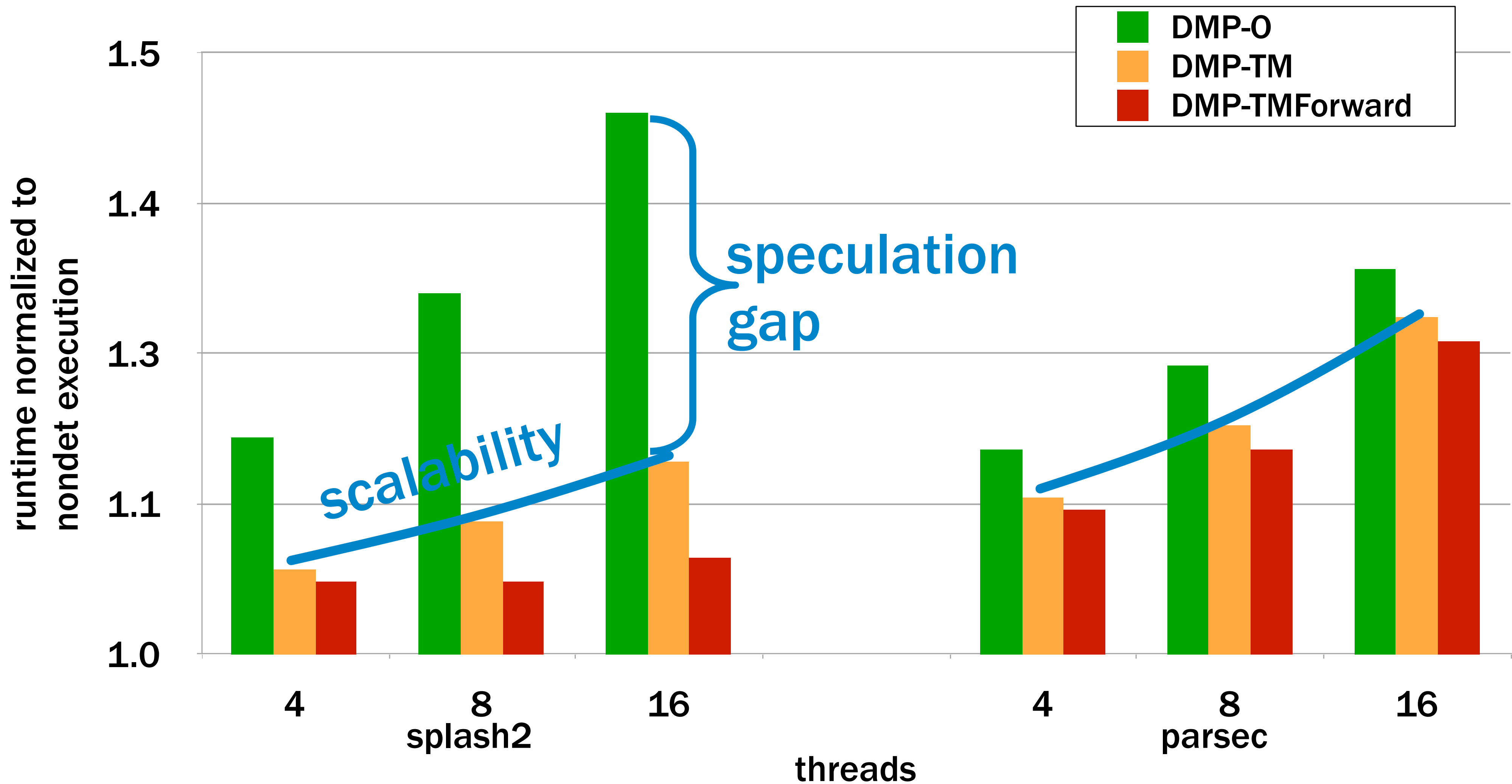
- lots of TM techniques to make commit fast

imbalance

- better quantum formation

DMP-O and DMP-TM Evaluation

(HW version)



Performance Summary

- DMP-O: Low overheads, ok (not great) scalability
- DMP-B: More overheads, good scalability
- DMP-TM: Even more overheads, great scalability (tricks)
- Exacerbates inherent lack of scalability of applications
  - Relaxing memory ordering helps a **lot**, even more so than in nondet MPs
- Implementations:
  - HW implementation: ~5% to 50%
  - Compiler implementation: 2x to 3x (instrumentation cost)
  - OS (paging tricks): 0% to 10x (false sharing at page granularity)

In case you want to learn more...

- DMP:
  - “Deterministic Shared Memory Multiprocessing”, ASPLOS’09, IEEE Micro Top Picks
  - “CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution”, ASPLOS’10
  - “Deterministic Process Groups in dOS”, OSDI’10
  - “RCDC: A Relaxed Consistency Deterministic Computer”, ASPLOS’11
- FailStop Races:
  - “A Case for System Support for Concurrency Exceptions”, Usenix HotPar’09
  - “Conflict Exceptions”, ISCA’10

Other work @ SAMPA

(ask me about it if you are interested...)

- Dynamic analysis for arbitrary concurrency bug detection
  - using graphs, machine learning
- Automatic concurrency bug avoidance
- Architecture support for Dynamic Languages
- Energy-exposed programming models
- JavaScript parallelization
- OS support for non-volatile main memory systems

Introducing Corensic Jinx

JINX
For Windows

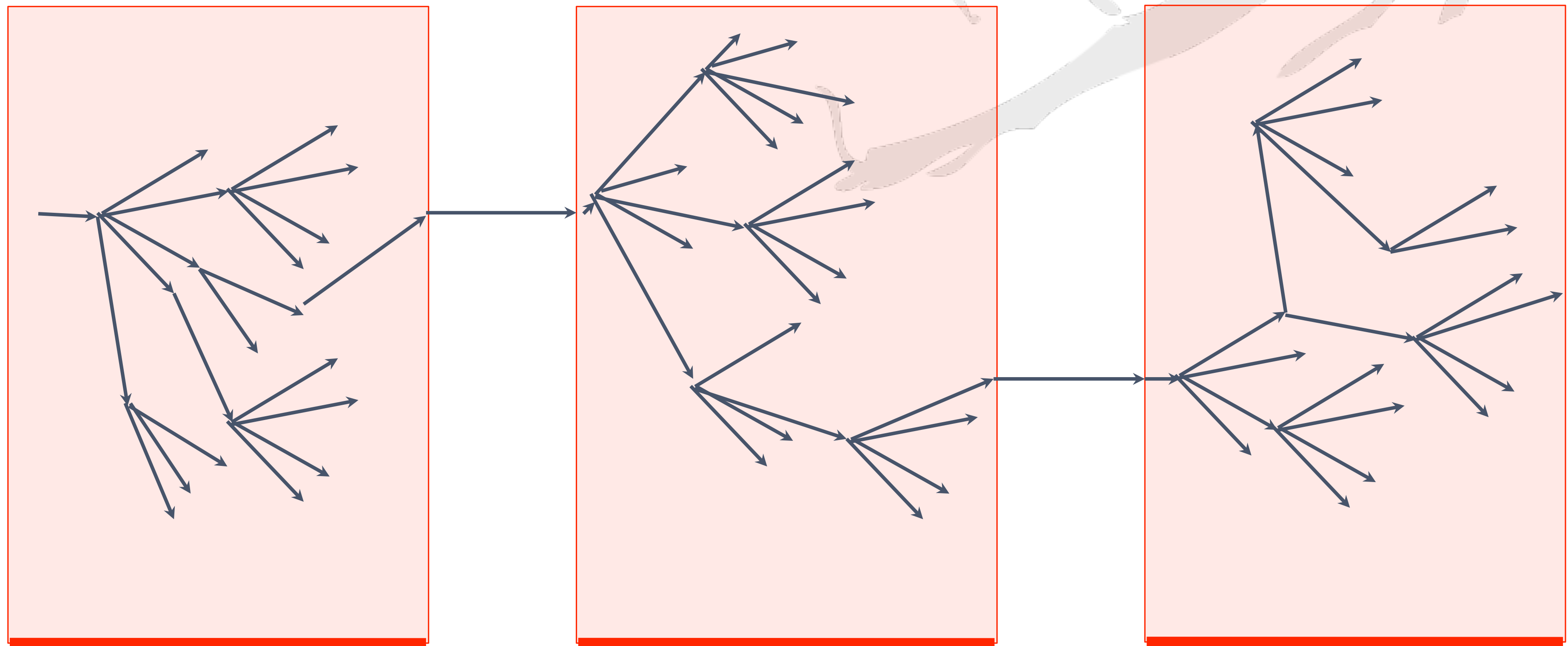
JINX
For Linux

Jinx is a tool that makes multi-core bugs happen quickly, and enables developers and testers to identify the root cause of bugs with immediate forensics

- Windows and Linux versions available now
- Kernel mode support for finding bugs in the full software stack during acceptance testing or system validation
- **Works the way you do**: Integrated into development and test processes…bugs happen faster and they make more sense
- **Reliable**: No false positives, definitively finds bugs in code
- **Easy**: 3 minutes from download to install to finding bugs

How Jinx Works: Forcing Bugs to Happen (1)

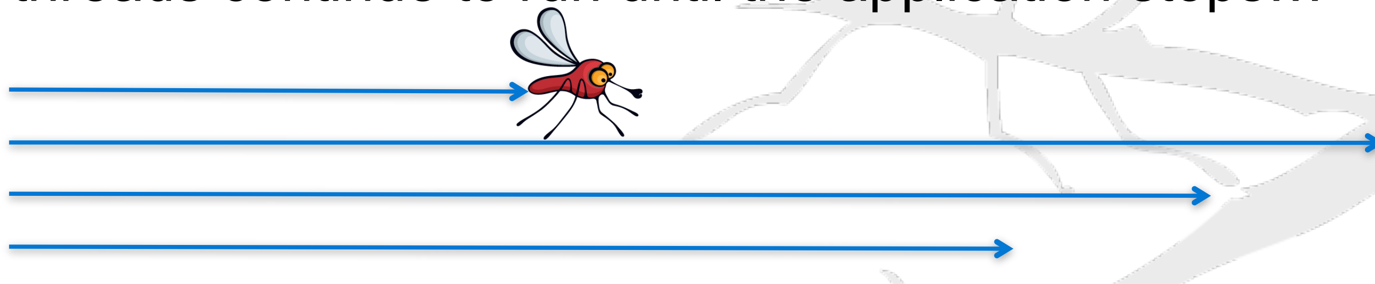
Jinx intelligently samples periods of execution and explores many different possibilities for thread timing…



corensic

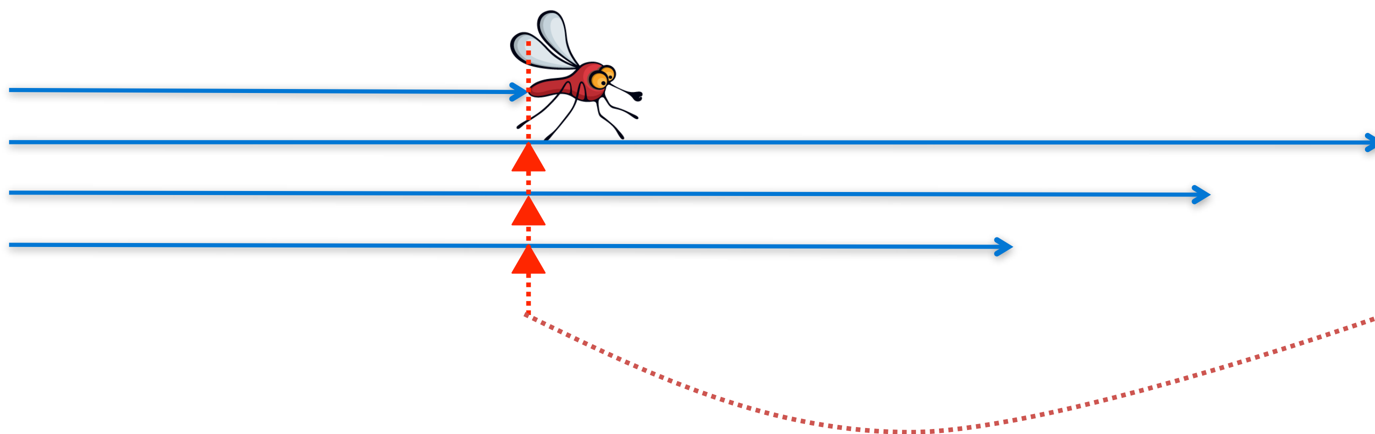
How Jinx Works: SmartStop (2)

In a typical multi-threaded program, when a bug occurs in one thread, the other threads continue to run until the application stops…



A developer viewing the end state of the program gets no meaningful info

Jinx is different. With SmartStop, Jinx automatically positions the end state of the program in a meaningful location for a developer to find the root cause of the concurrency bug…



corensic™

The Jinx Advantage

- Fully complementary to existing tools and development processes
  - Jinx accelerates the rate at which bugs manifest themselves
  - Jinx pinpoints causality of bugs
- Enhances the effectiveness of common testing techniques
  - Easily plugs into stress and load testing environments
- No false positives
- Intelligently samples execution (no random timing intervals)
- Operating system and application platform independent
- No changes necessary to source code and most testing scripts

Determinism and Fail-stop Races for Sane Multiprocessing

Luis Ceze, *University of Washington*

sampa

Safe MultiProcessing Architectures at the University of Washington