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.

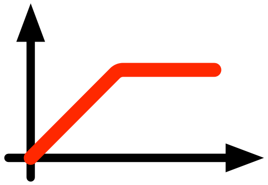
Safe MultiProcessing Architectures
at the University of Washington

Guest Lecture at University of Cambridge, November 2013.

You probably heard this many times

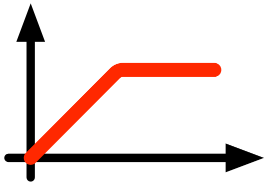
You probably heard this many times

- Era of “free” performance is over.



You probably heard this many times

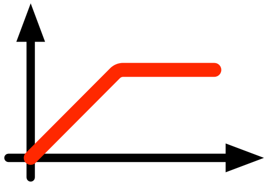
- Era of “free” performance is over.



- Most of compute power now scaling in terms of cores
  - Mostly due to power and complexity reasons. Copy & Paste in VLSI :)

You probably heard this many times

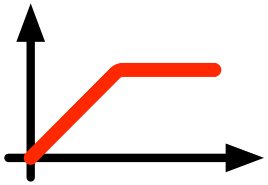
- Era of “free” performance is over.



- 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

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

thread 0

shared variable

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

thread 1

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

A multithreaded voting machine

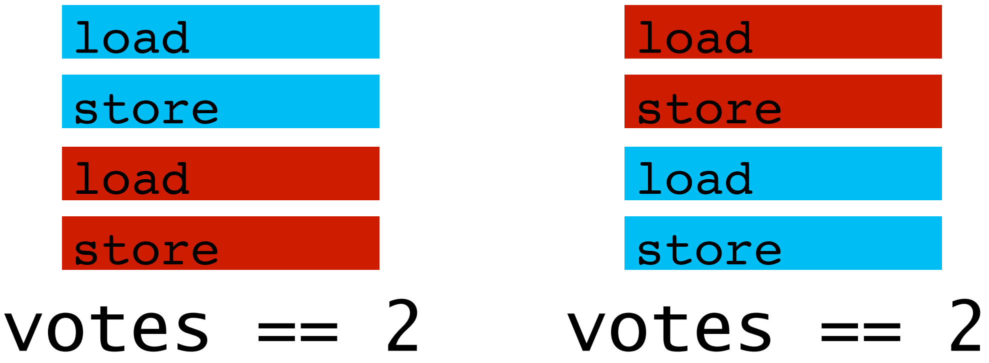
shared variable

thread 0

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

thread 1

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



load
store
load
store

votes == 2

load
store
load
store

votes == 2

A multithreaded voting machine

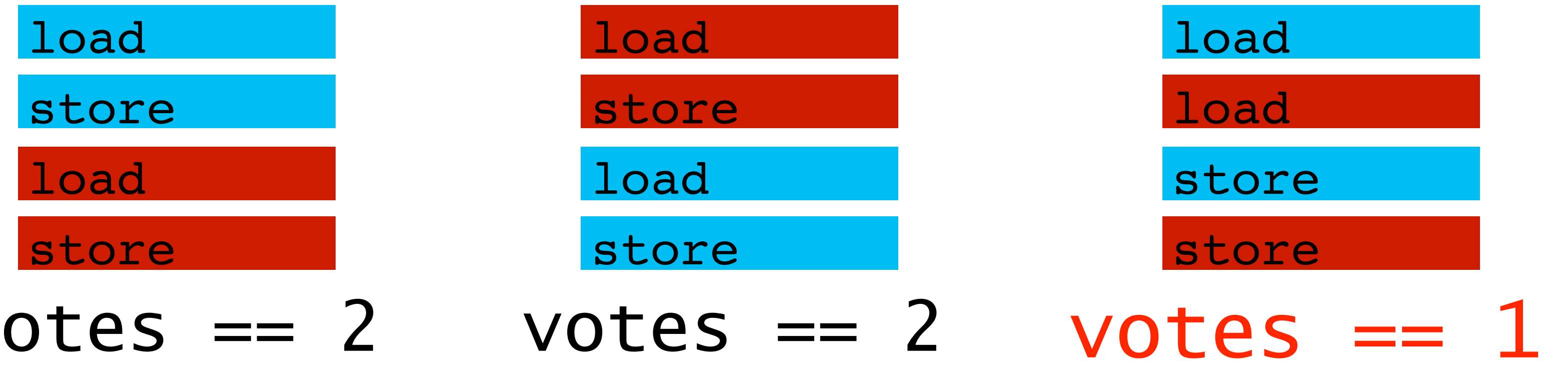
shared variable

thread 0

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

thread 1

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



🟦 load
🟦 store
🟥 load
🟥 store

votes == 2

🟥 load
🟥 store
🟦 load
🟦 store

votes == 2

🟦 load
🟥 load
🟦 store
🟥 store

votes == 1

Two Key Problems

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

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

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?

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

Semantics are clear and simple

Better data-race debugging

Safety: races can’t cause problems

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?)

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?

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

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

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

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`	`st 1 → y`
`ld y`	`st 1 → x`

How about (1,0)?

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

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)?

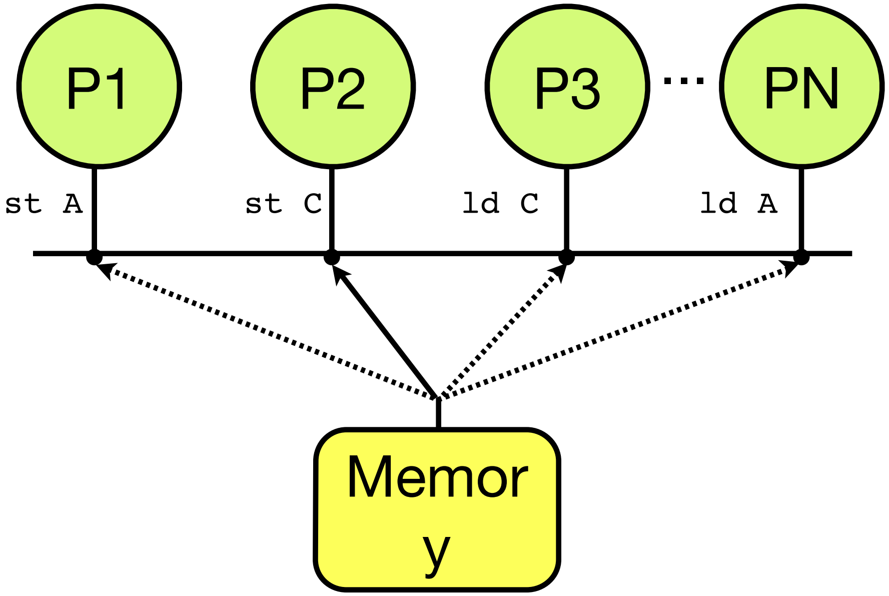
thread 1	thread 2

`st 1 → y`
`ld x`
	`st 1 → x`
	`ld y`

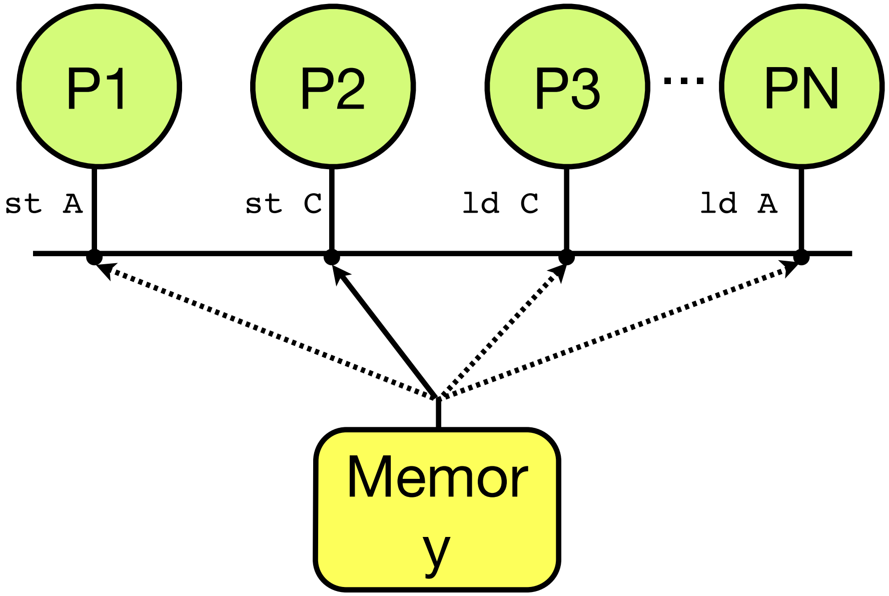
How about (0,0)?

Sequential Consistency (SC)

Sequential Consistency (SC)

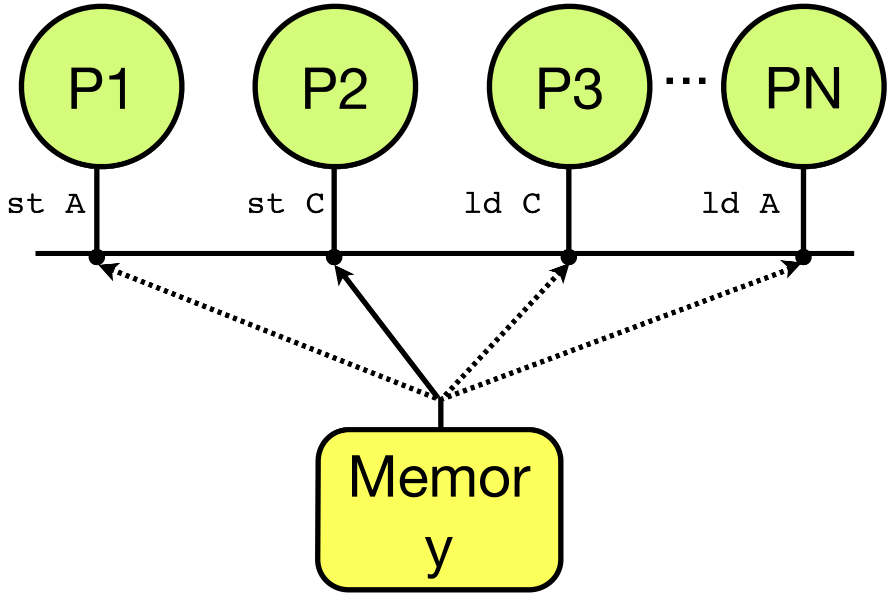


Sequential Consistency (SC)



[Lamport'79]

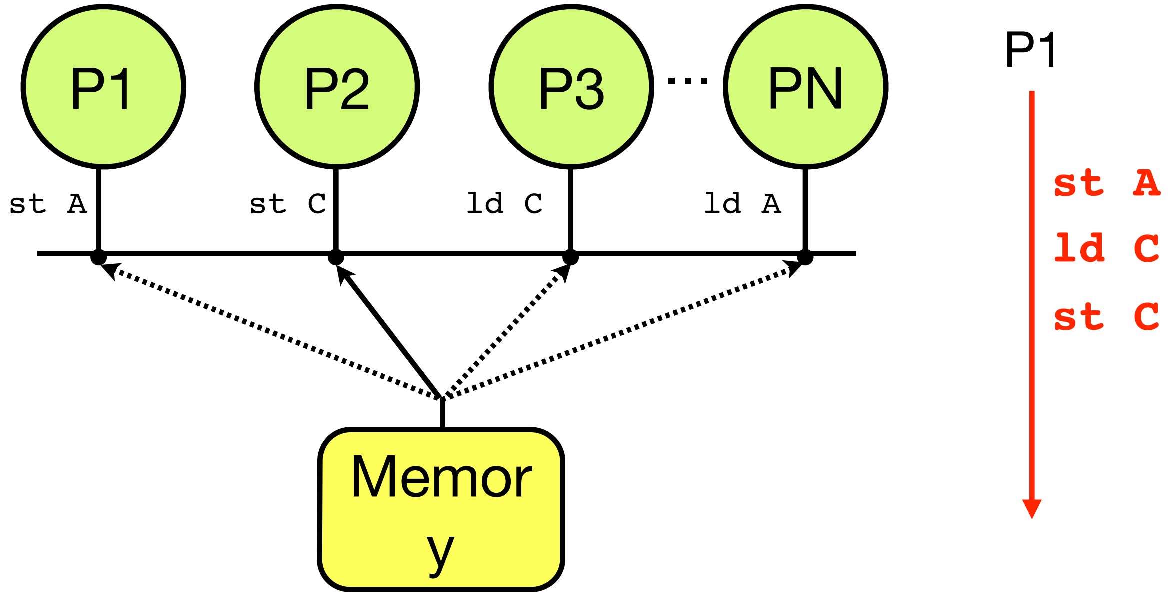
Sequential Consistency (SC)



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

[Lamport'79]

Sequential Consistency (SC)



P1 P2 P3 … PN

st A st C ld C ld A

Memory

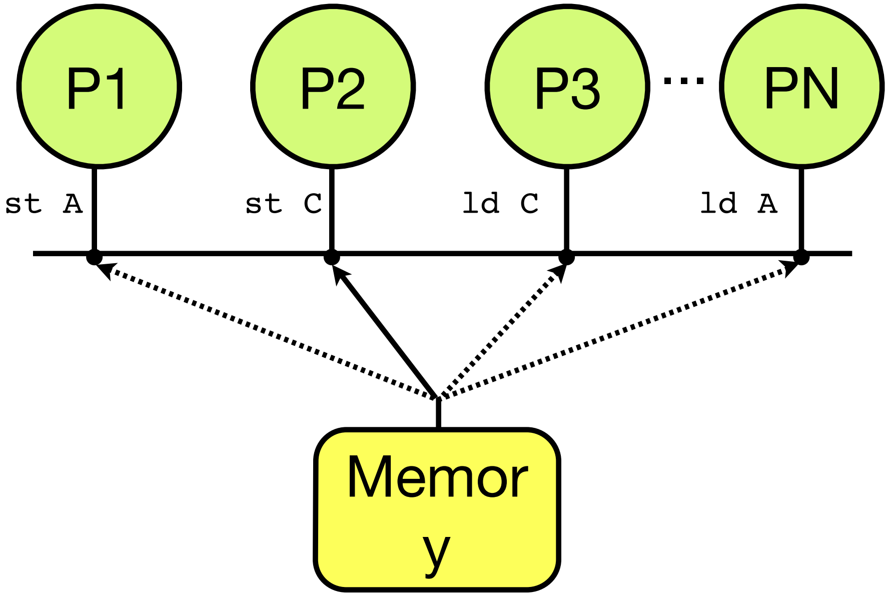
P1

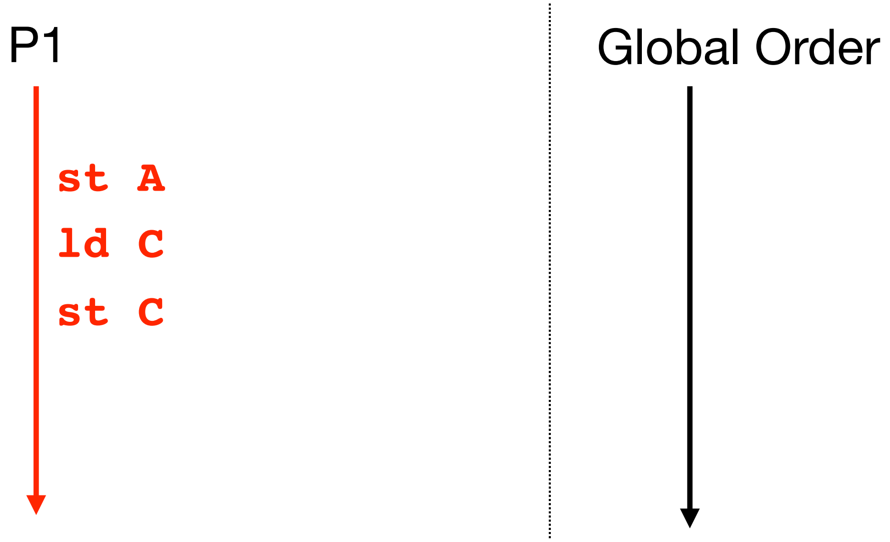
```
st A
ld C
st C
```

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

[Lamport'79]

Sequential Consistency (SC)

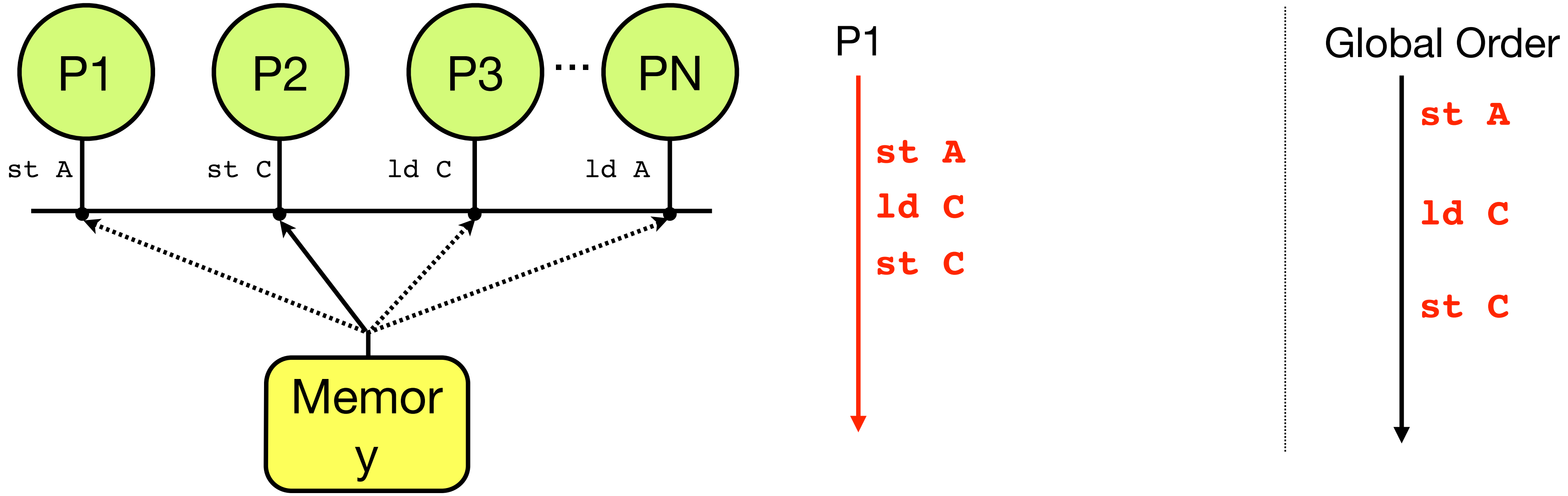




Per-processor program order: memory operations from individual processors maintain program order

[Lamport'79]

Sequential Consistency (SC)



P1

st A
ld C
st C

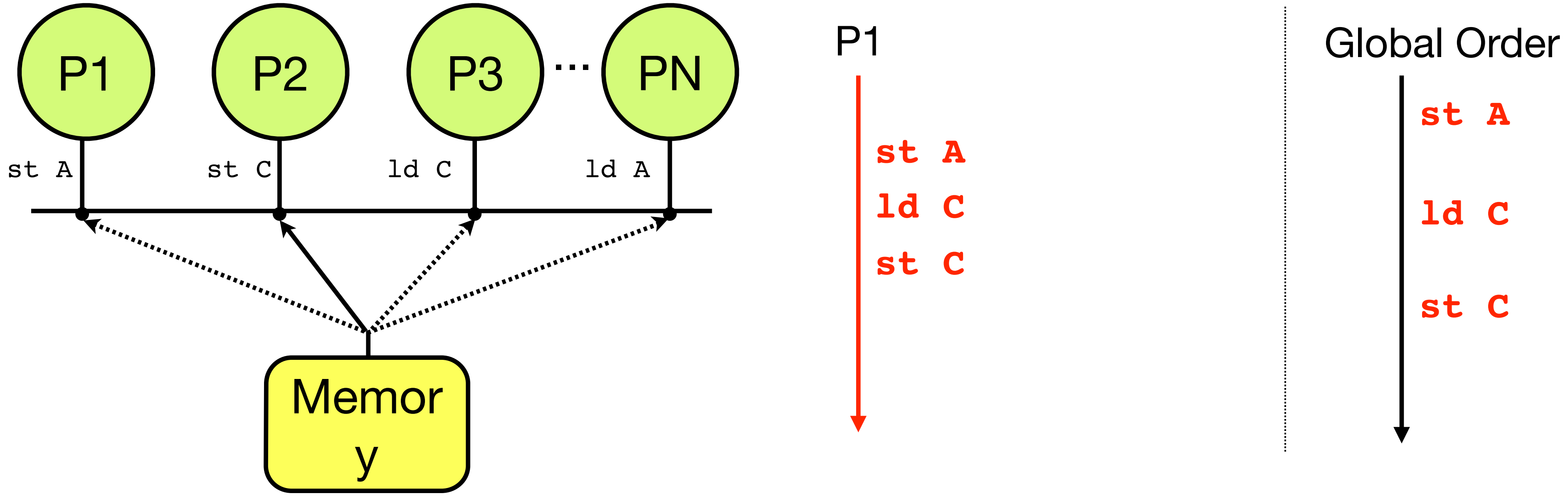
Global Order

st A
ld C
st C

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

[Lamport’79]

Sequential Consistency (SC)

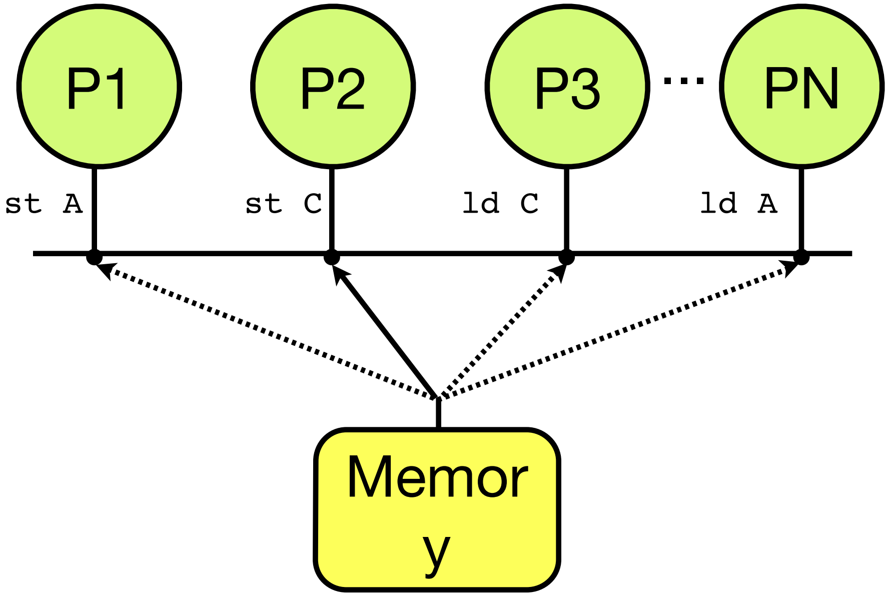


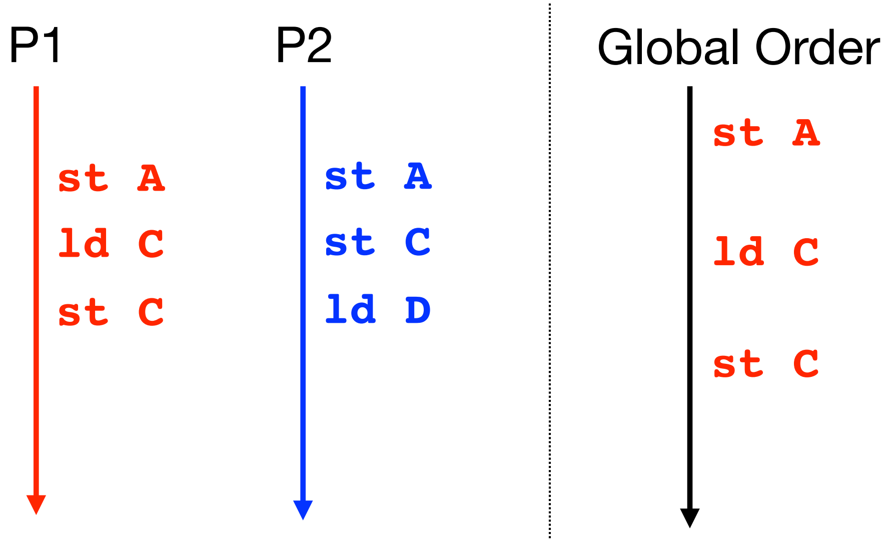
Per-processor program order: memory operations from individual processors maintain program order

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

[Lamport'79]

Sequential Consistency (SC)



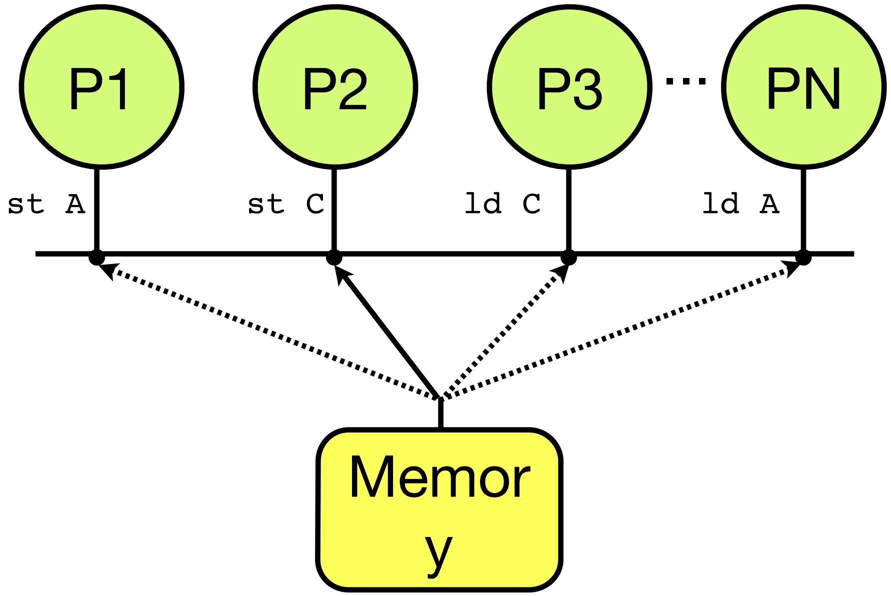


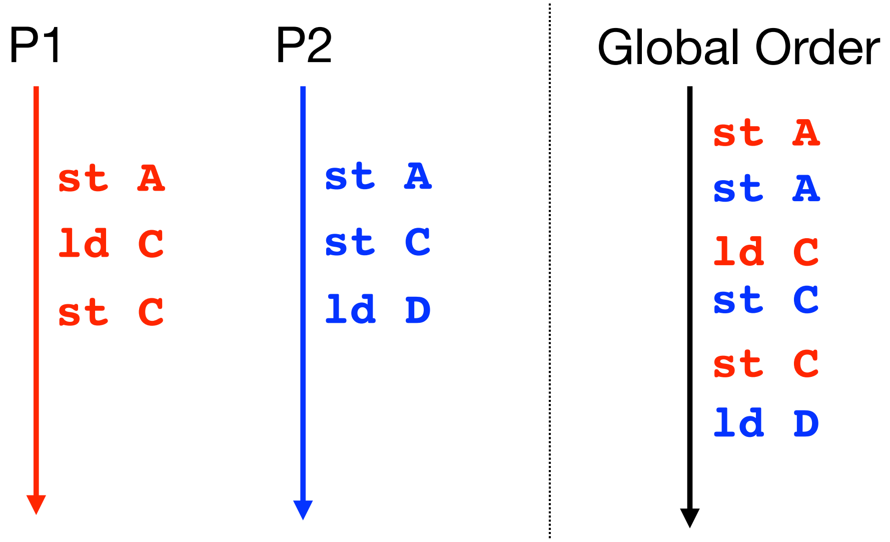
Per-processor program order: memory operations from individual processors maintain program order

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

[Lamport'79]

Sequential Consistency (SC)





P1

st A
ld C
st C

P2

st A
st C
ld D

Global Order

st A
st A
ld C
st C
st C
ld D

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

Sequential Consistency Implications

- What are the implications of that to:
  - compiler optimizations?
  - hardware optimizations?

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

C/C++ Standard on Memory Model

- Sequential Consistency...

C/C++ Standard on Memory Model

- Sequential Consistency...

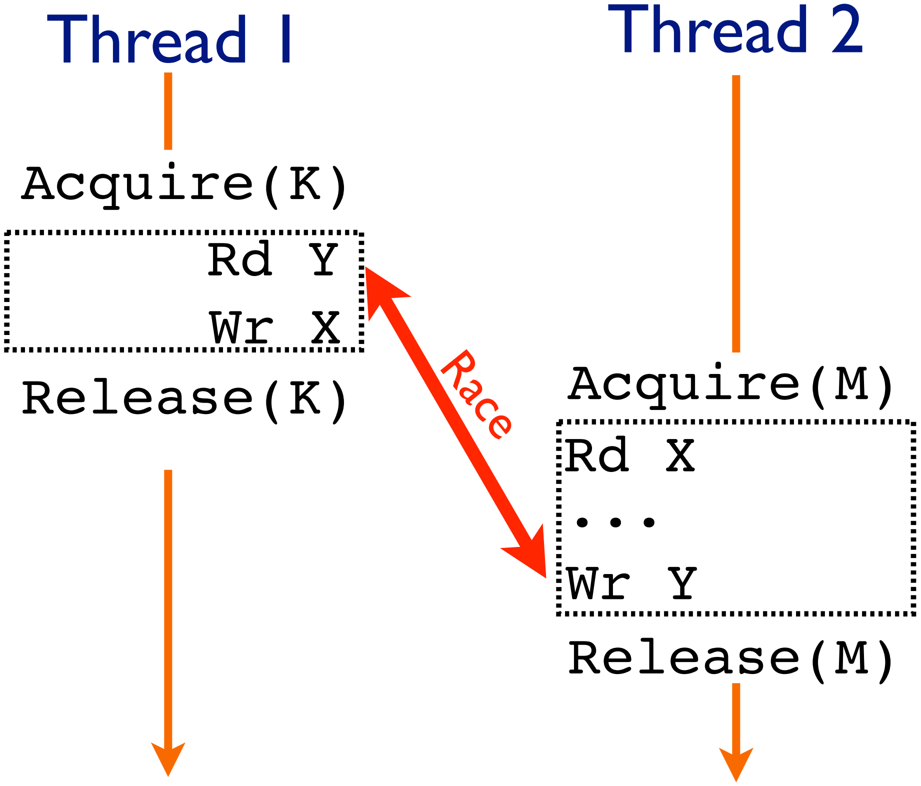
- for **Data-Race Free** programs

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

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



Thread 1

Acquire(K)

Rd Y

Wr X

Release(K)

Thread 2

Acquire(M)

Rd X

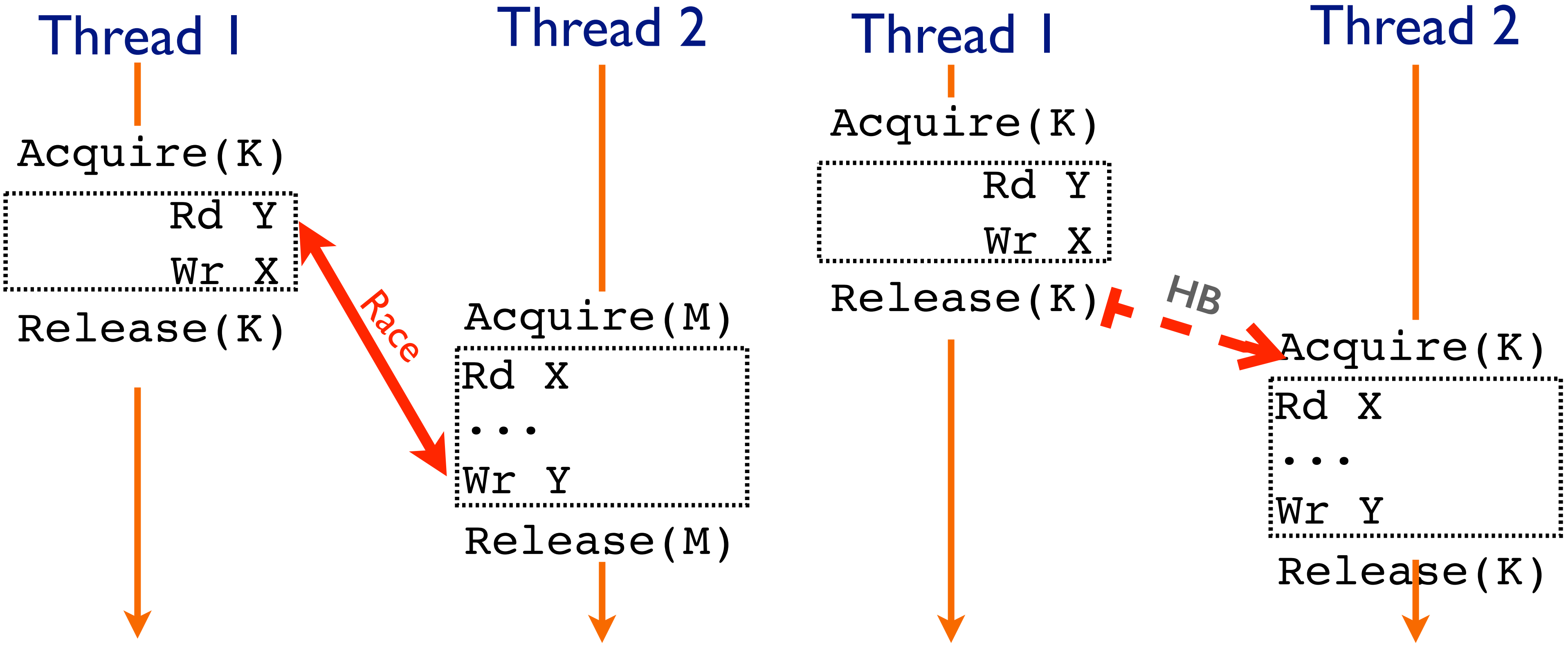
...

Wr Y

Release(M)

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

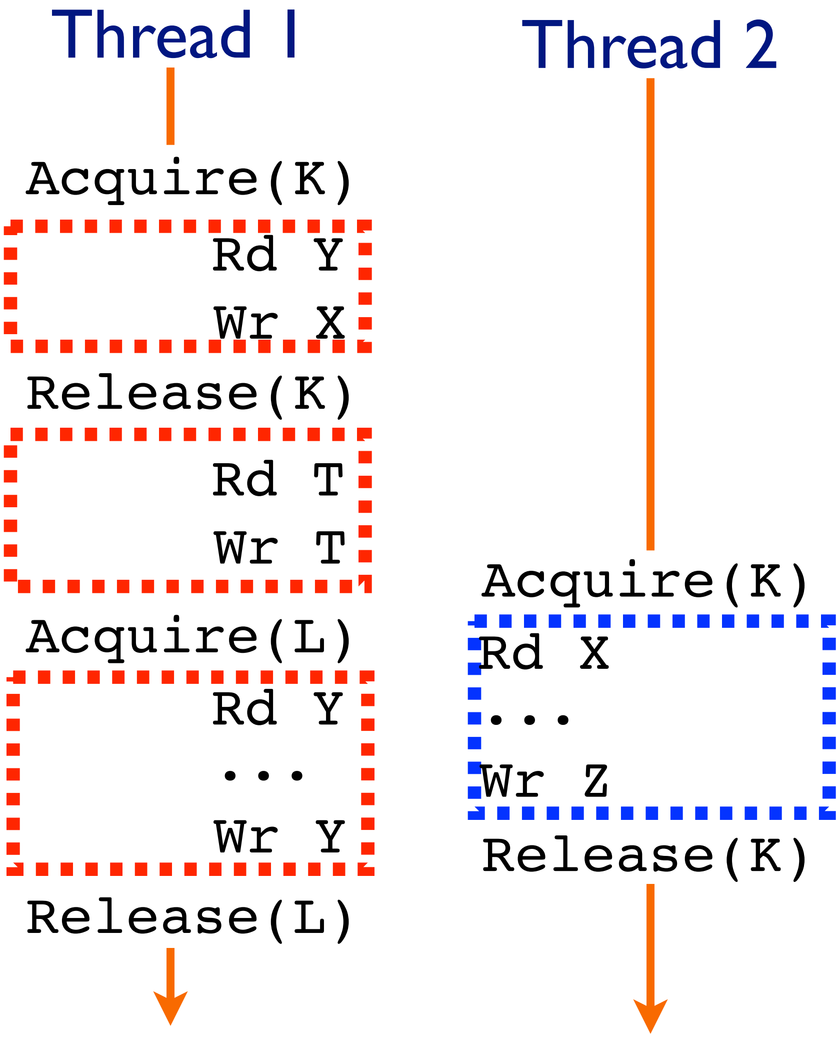


C/C++ Standard on Memory Model

- 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

Sequential Consistency for DRF Example



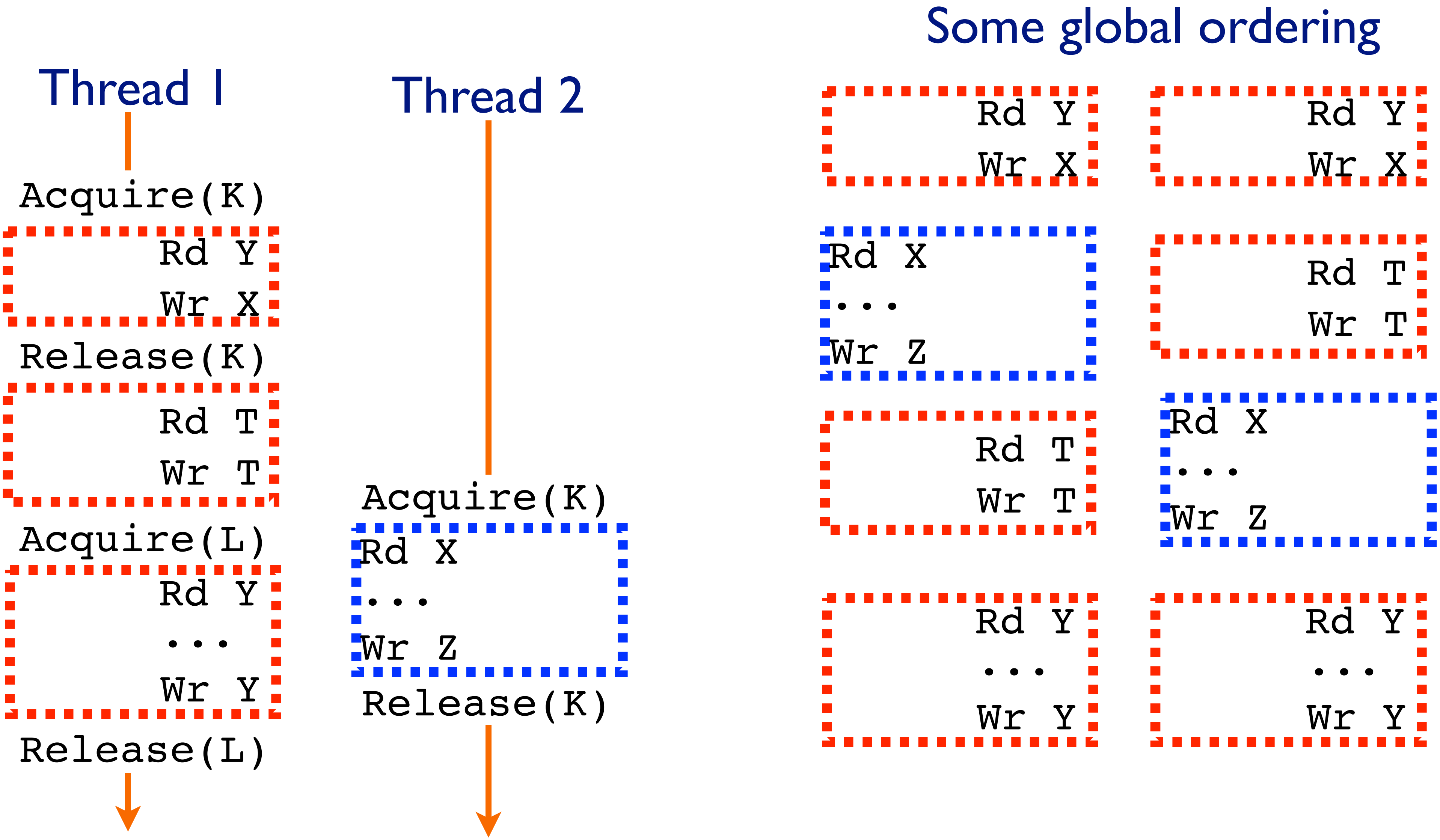
Thread 1

Acquire(K)
Rd Y
Wr X
Release(K)
Rd T
Wr T
Acquire(L)
Rd Y
...
Wr Y
Release(L)

Thread 2

Acquire(K)
Rd X
...
Wr Z
Release(K)

Sequential Consistency for DRF Example



Thread 1

```
Acquire(K)
      Rd Y
      Wr X
Release(K)
      Rd T
      Wr T
Acquire(L)
      Rd Y
      ...
      Wr Y
Release(L)
```

Thread 2

```
Acquire(K)
Rd X
...
Wr Z
Release(K)
```

Some global ordering

```
      Rd Y          Rd Y
      Wr X          Wr X

Rd X                Rd T
...                 Wr T
Wr Z
                Rd X
      Rd T      ...
      Wr T      Wr Z

      Rd Y          Rd Y
      ...           ...
      Wr Y          Wr Y
```

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 Exceptions: The Vision

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

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?

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?

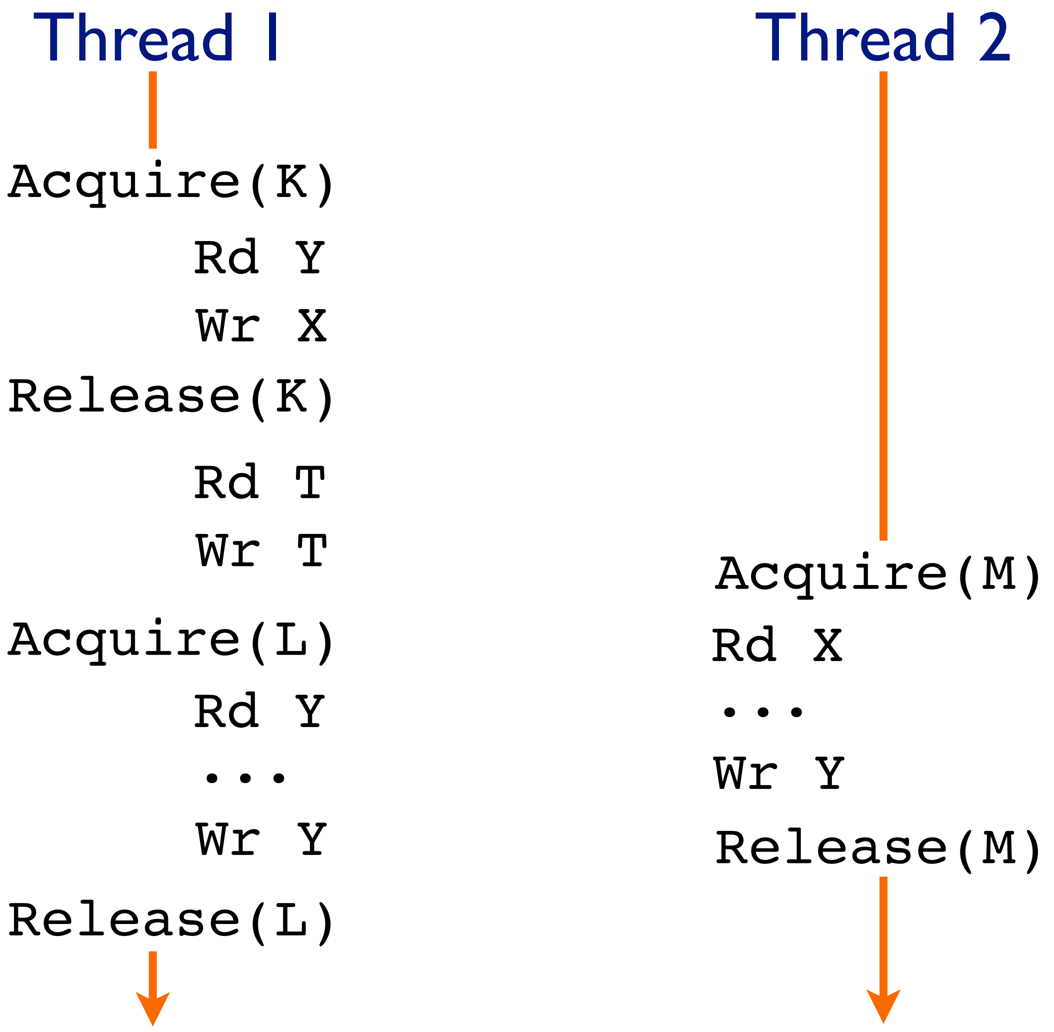
- We are starting with data-races
  - Well defined, doesn't require programmer annotations, language semantics

Goals In Supporting Races as Exceptions

High-Performance - Always-on detection

Precise detection - No false positives

Conflict Exceptions [ISCA'10]



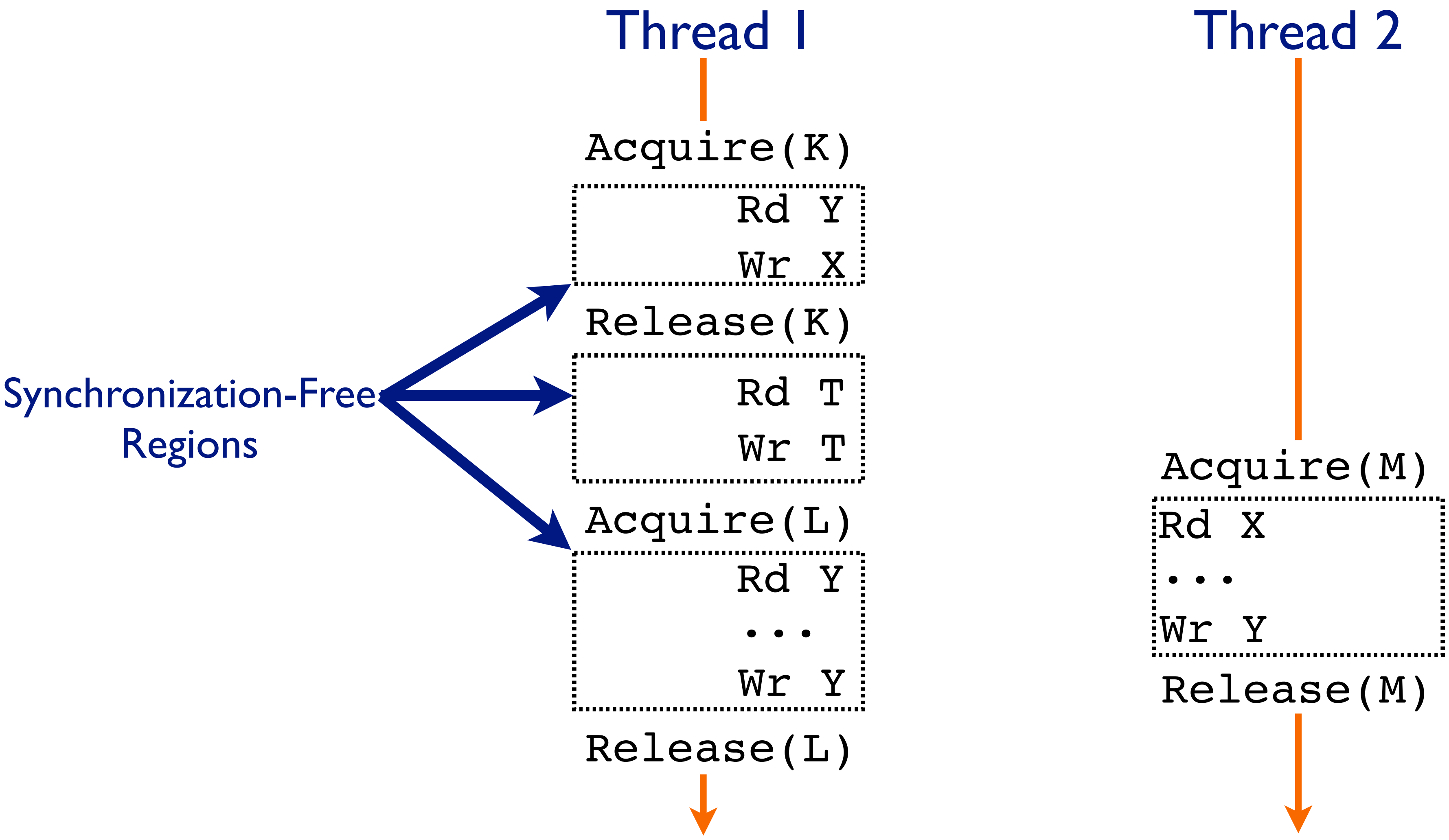
Thread 1

```
Acquire(K)
        Rd Y
        Wr X
Release(K)
        Rd T
        Wr T
Acquire(L)
        Rd Y
        ...
        Wr Y
Release(L)
```

Thread 2

```
Acquire(M)
Rd X
...
Wr Y
Release(M)
```

Conflict Exceptions [ISCA'10]



Thread 1

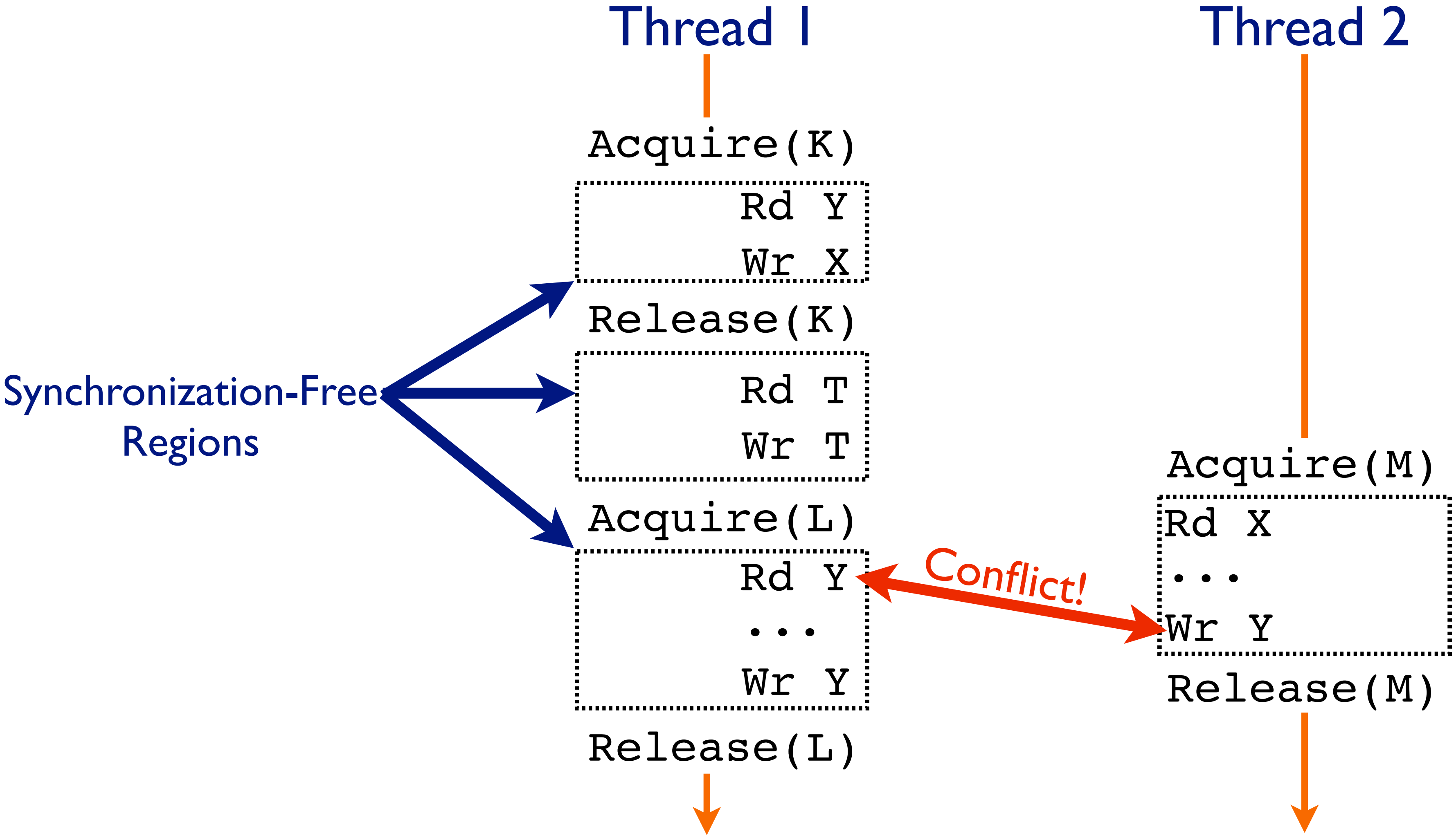
```
Acquire(K)
    Rd Y
    Wr X
Release(K)
    Rd T
    Wr T
Acquire(L)
    Rd Y
    ...
    Wr Y
Release(L)
```

Thread 2

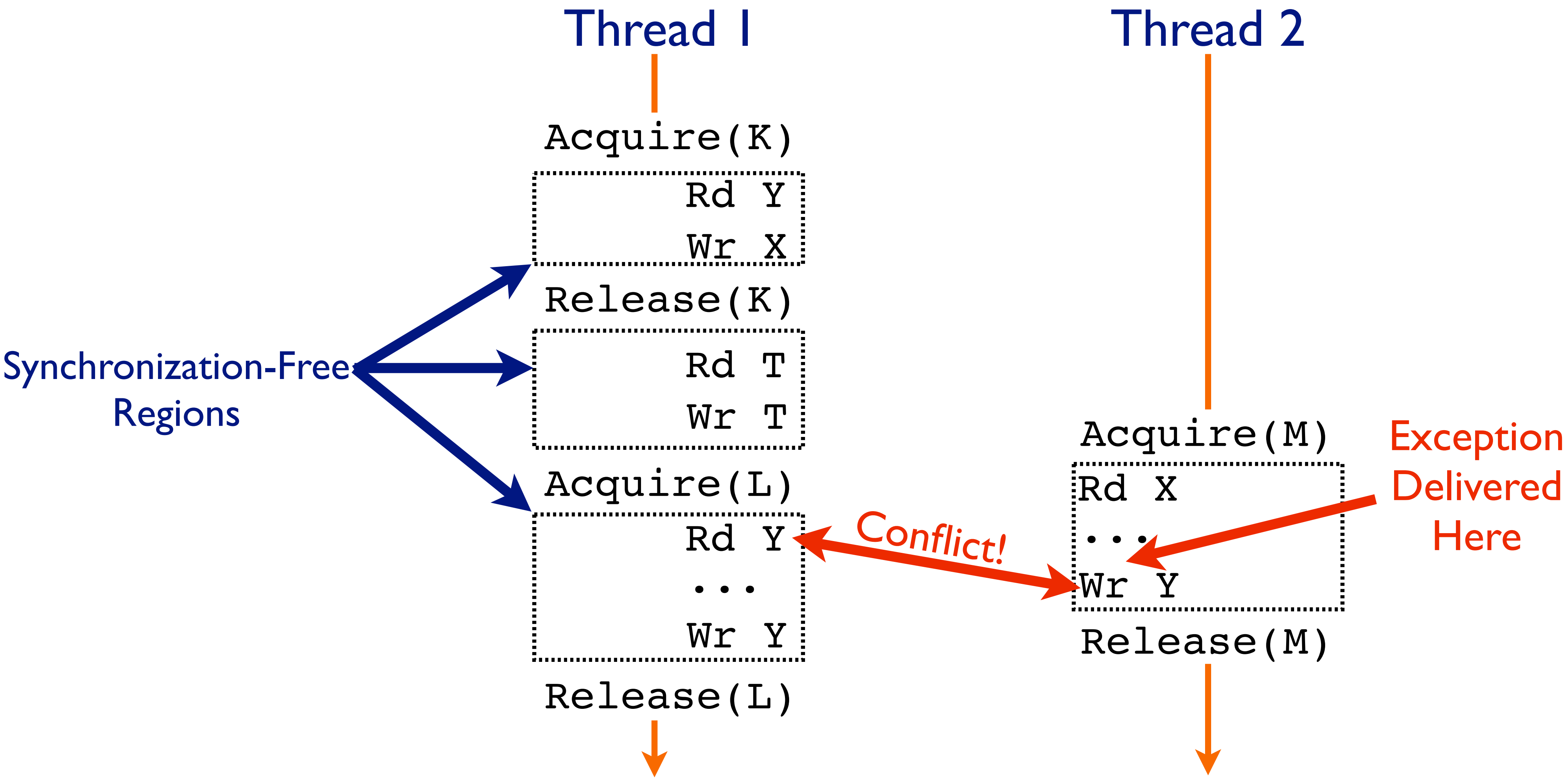
```
Acquire(M)
Rd X
...
Wr Y
Release(M)
```

Synchronization-Free Regions

Conflict Exceptions [ISCA'10]



Conflict Exceptions [ISCA'10]



Thread 1

Thread 2

```
Acquire(K)
      Rd  Y
      Wr  X
Release(K)
      Rd  T
      Wr  T
Acquire(L)
      Rd  Y
      ...
      Wr  Y
Release(L)
```

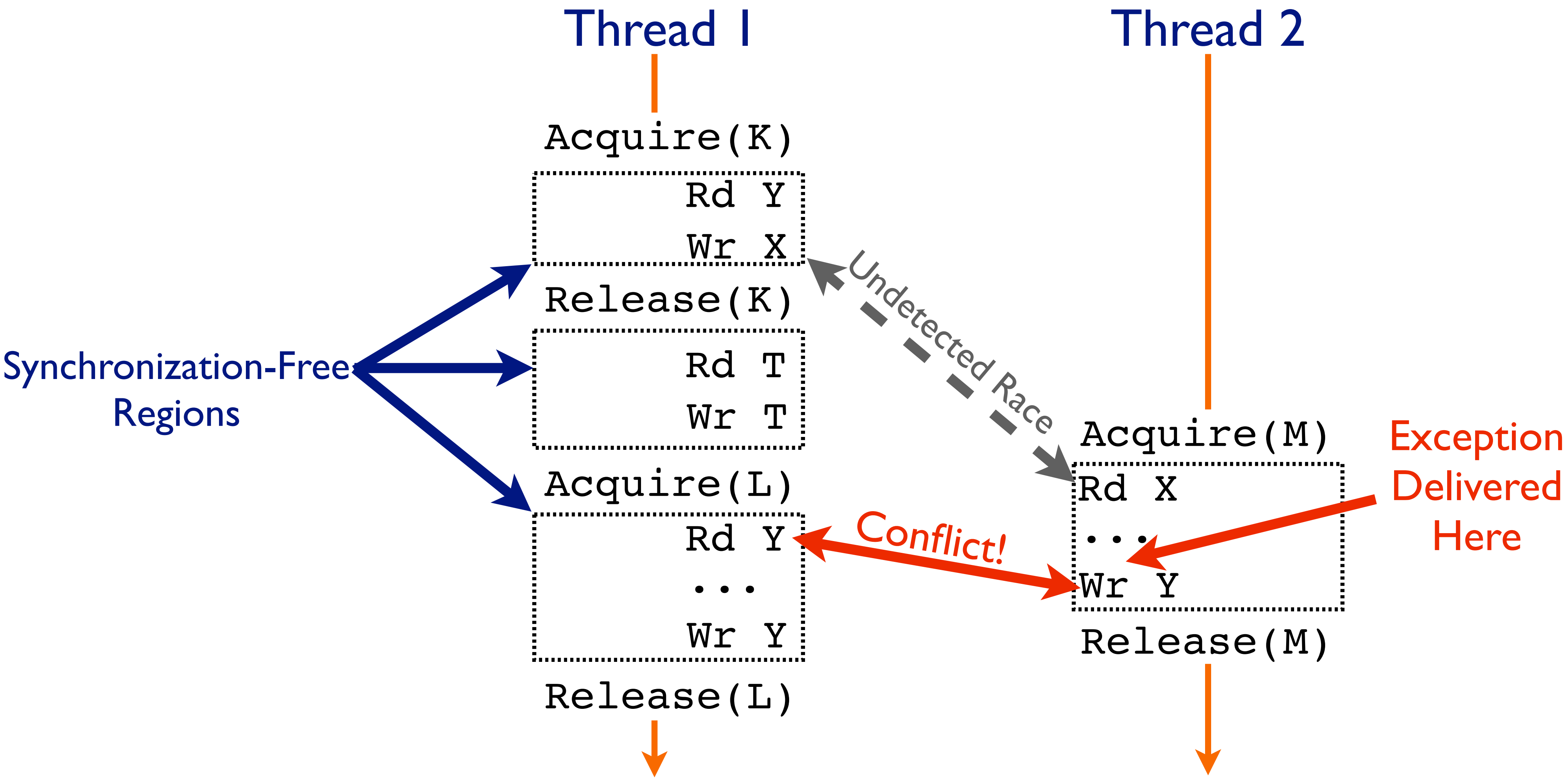
```
Acquire(M)
Rd  X
...
Wr  Y
Release(M)
```

Synchronization-Free Regions

Conflict!

Exception Delivered Here

Conflict Exceptions [ISCA'10]

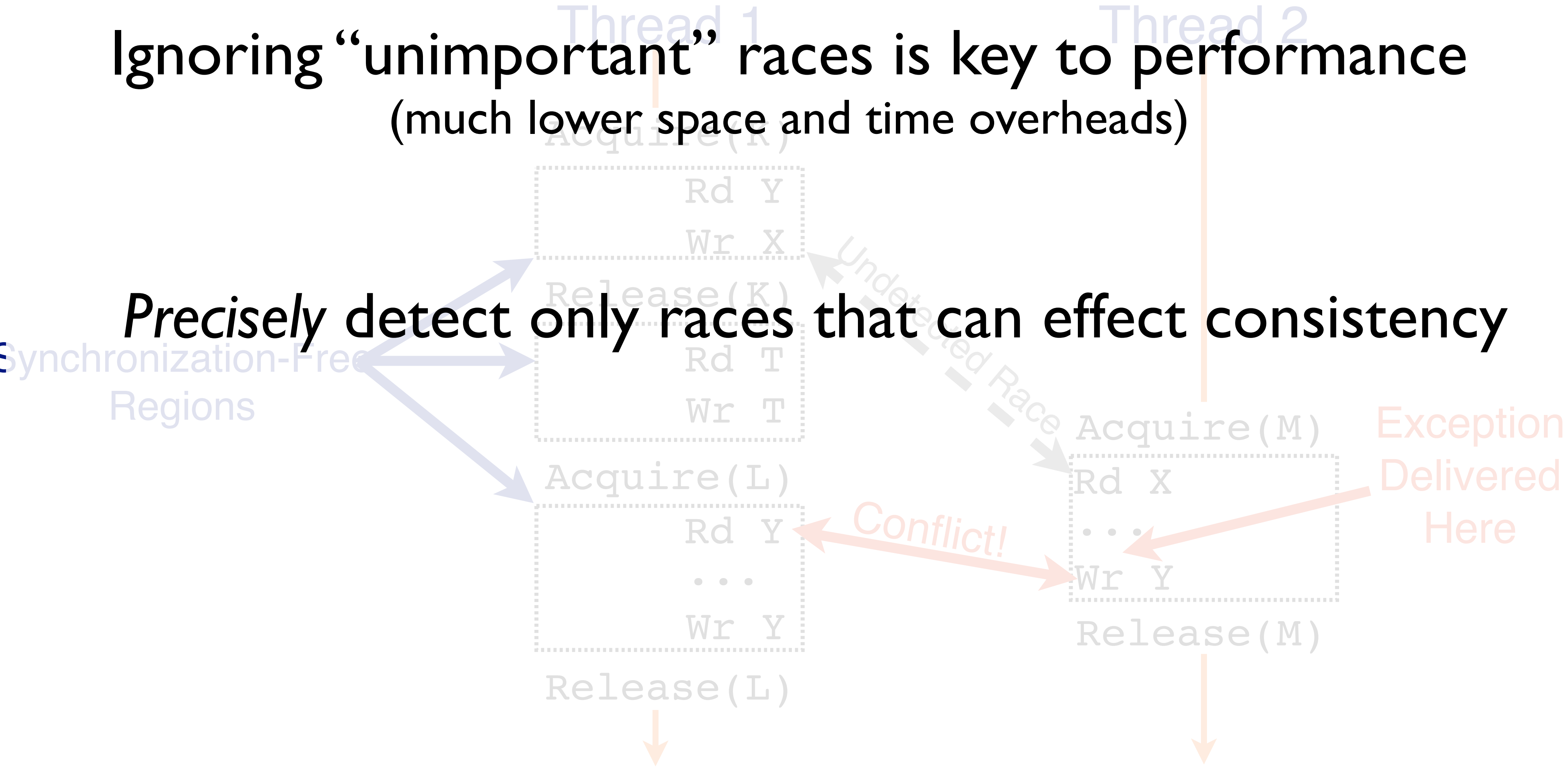


Conflict Exceptions

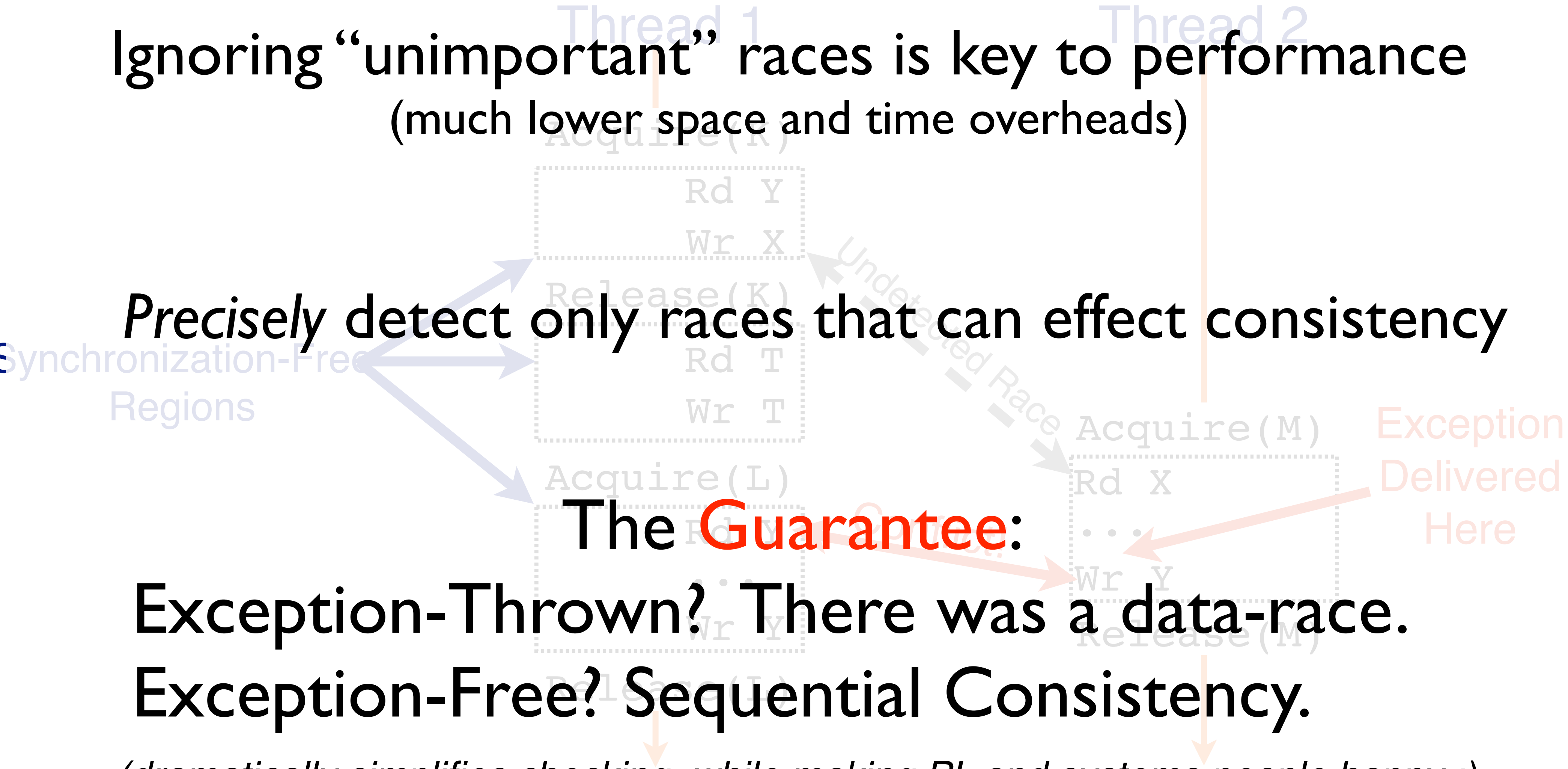
Ignoring “unimportant” races is key to performance

(much lower space and time overheads)

Precisely detect only races that can effect consistency



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:

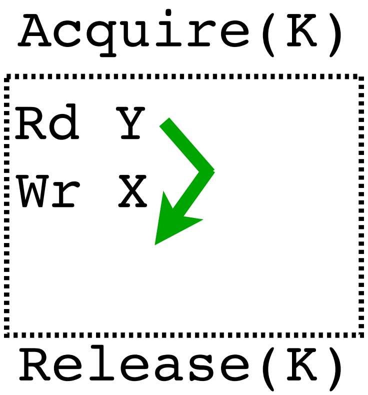
Exception-Thrown? There was a data-race.

Exception-Free? Sequential Consistency.

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

Language Level Benefits

Reordering in SFRs is legal

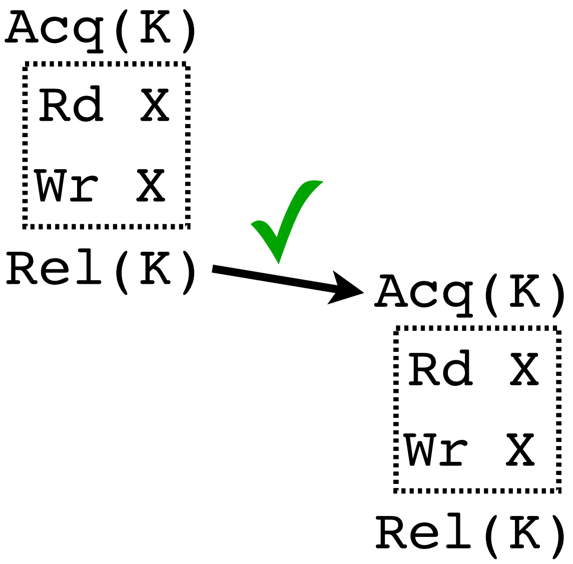


```
Acquire(K)
Rd Y
Wr X
Release(K)
```

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

Granularity independence

Exception-Free executions are SC



```
Acq(K)
Rd X
Wr X
Rel(K)
```

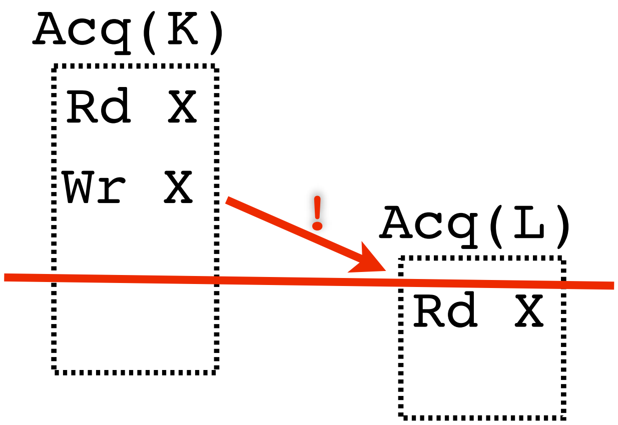
✓

```
Acq(K)
Rd X
Wr X
Rel(K)
```

Language Level Benefits

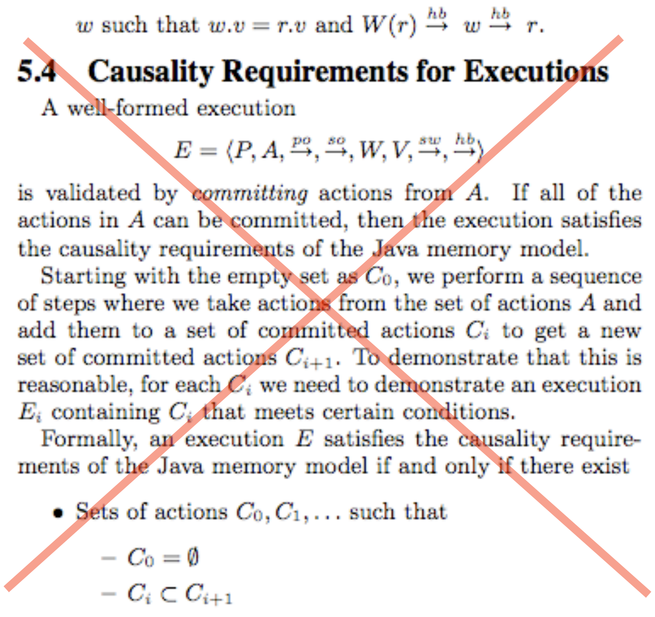
Programming model is largely the same

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



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.

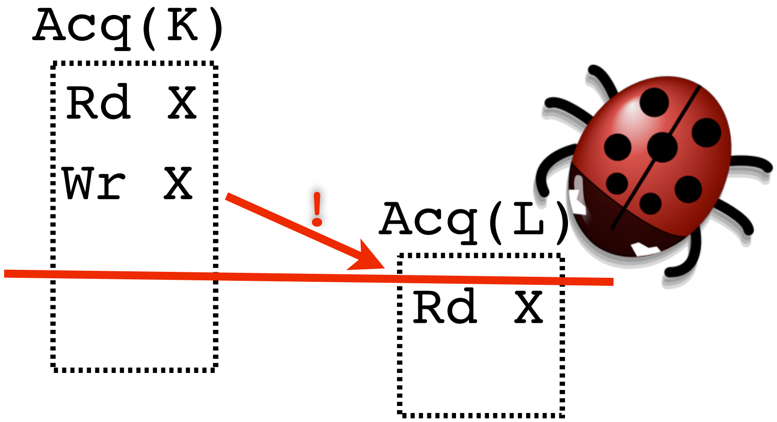
Formally, an execution E satisfies the causality requirements of the Java memory model if and only if there exist

- 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

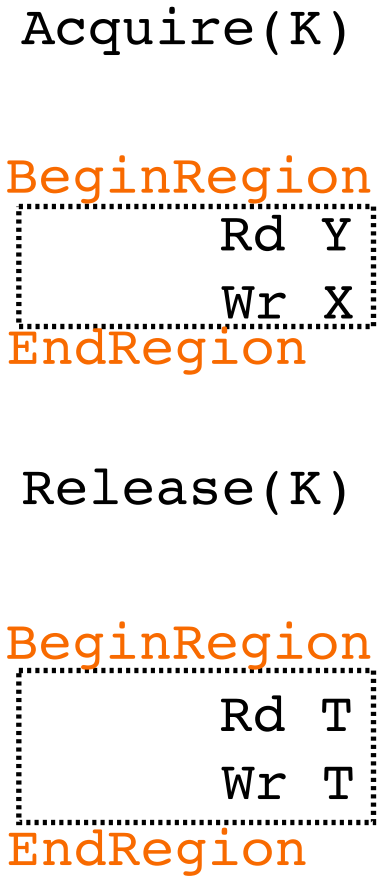
Exceptions Thrown Precisely Before Conflicting Instruction

Hardware/Software Interface

New Instructions:
`BeginRegion` and `EndRegion`

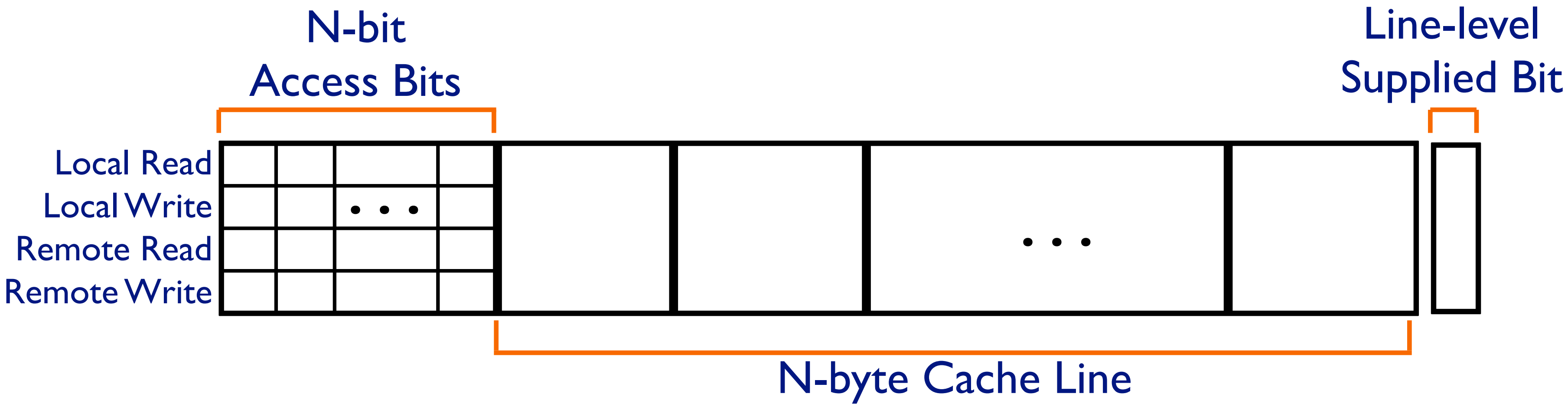
Synchronization Operations are Singleton Regions

Exceptions Thrown Precisely Before Conflicting Instruction

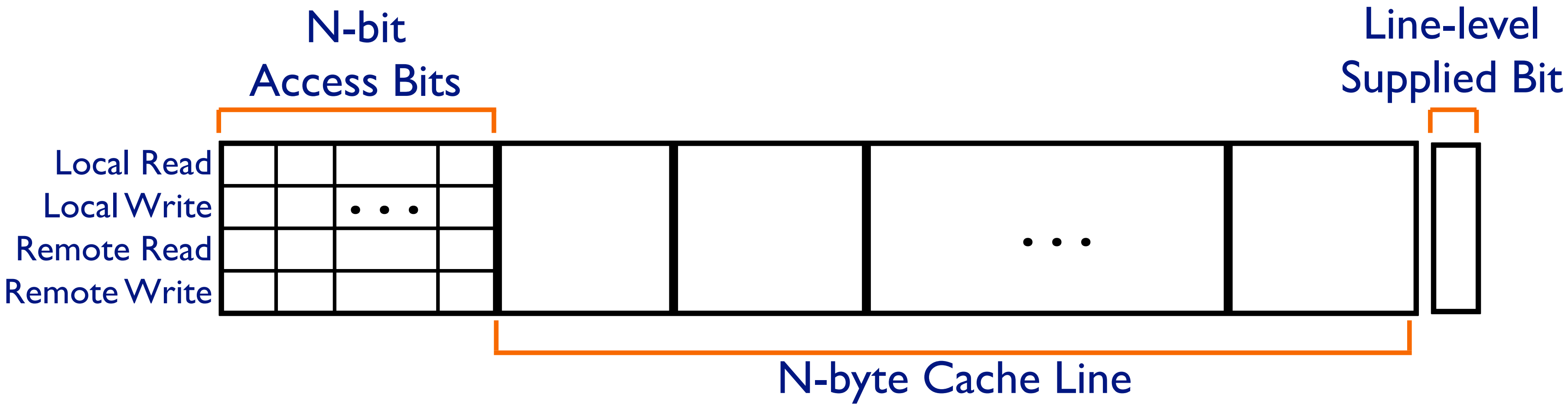


```
Acquire(K)

BeginRegion
        Rd Y
        Wr X
EndRegion

Release(K)

BeginRegion
        Rd T
        Wr T
EndRegion
```

Access Monitoring

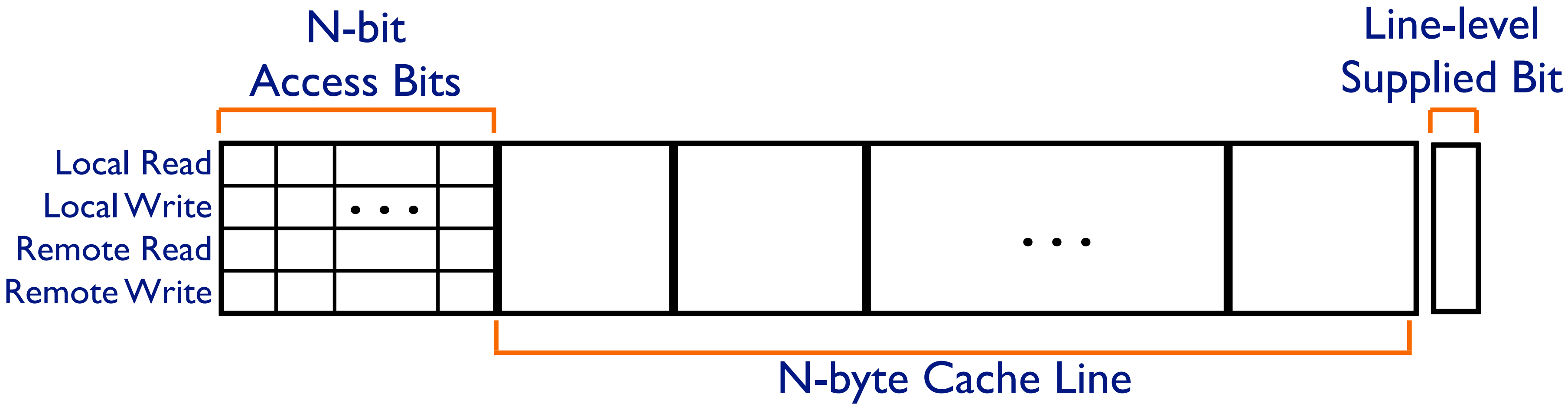


Access Monitoring



Exception Test: compare local and remote bits

Access Monitoring

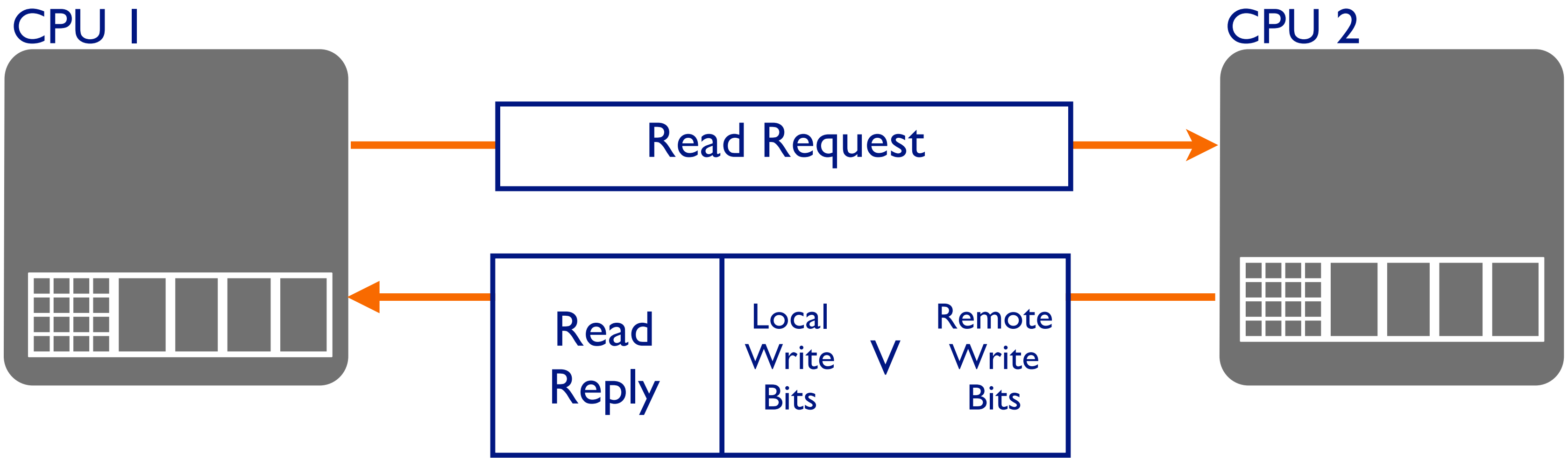


Exception Test: compare local and remote bits

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

Leveraging Coherence Support

Read Coherence Actions



CPU 1

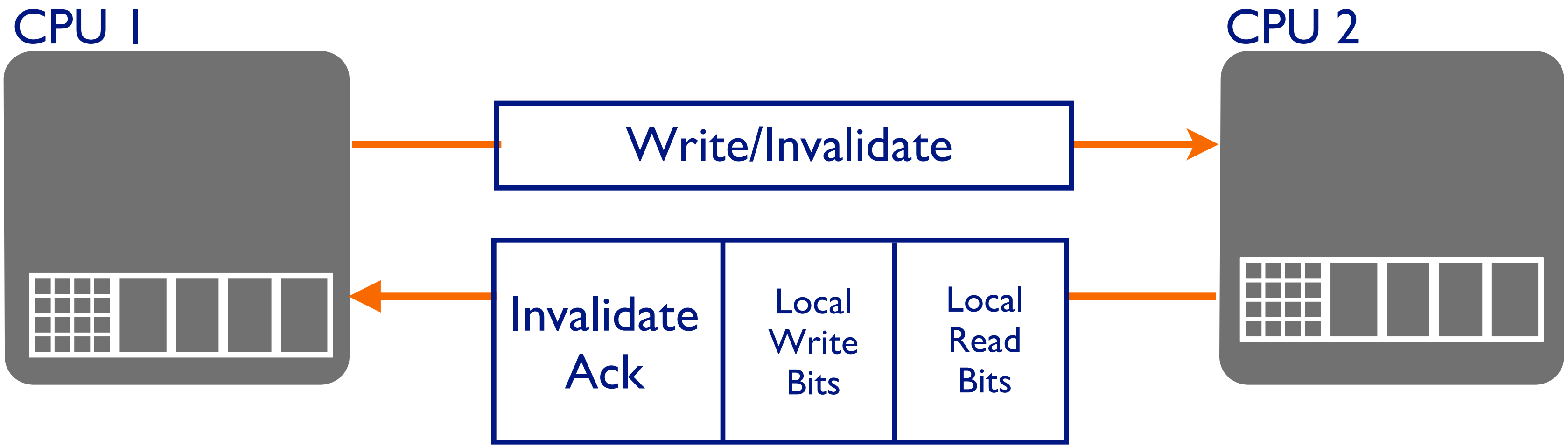
CPU 2

Read Request

Read Reply

Local Write Bits ∨ Remote Write Bits

Write Coherence Actions



CPU 1

CPU 2

Write/Invalidate

Invalidate Ack

Local Write Bits

Local Read Bits

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]

Deterministic Multiprocessing at 10,000’

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

- DMP provides *execution-level* determinism for arbitrary multithreaded programs:

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

Deterministic Multiprocessing at 10,000’

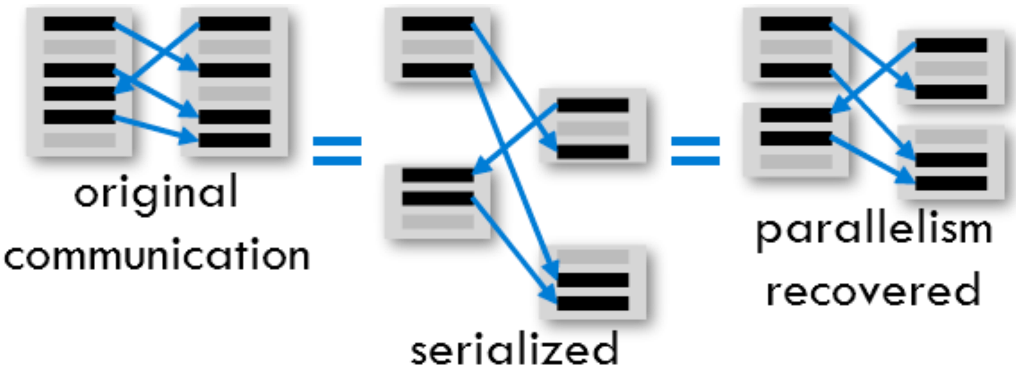
[ASPLOS’09, ASPLOS’10, OSDI’10, ASPLOS’11]

- 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

Deterministic Multiprocessing at 10,000’

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

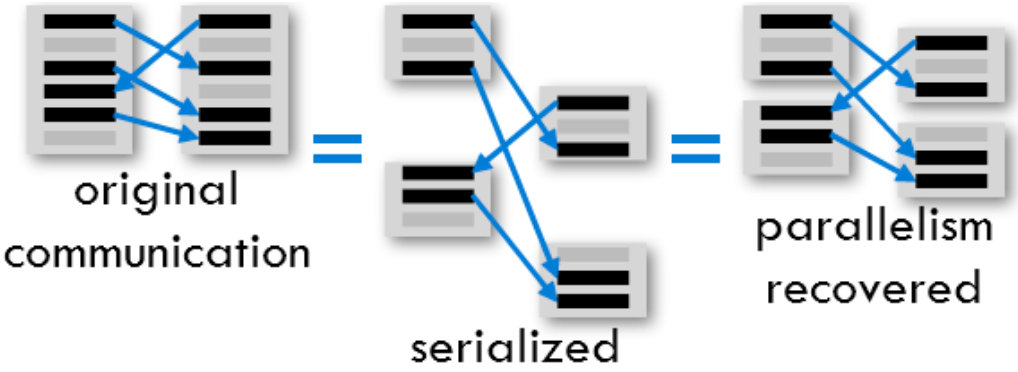
- DMP provides *execution-level* determinism for arbitrary multithreaded programs:
  - execution is only function of explicit inputs => single execution per input
  - this is not record-replay of multithreaded programs
- ***Key idea***: conceptually serialize execution, recover parallelism while preserving serial execution semantics



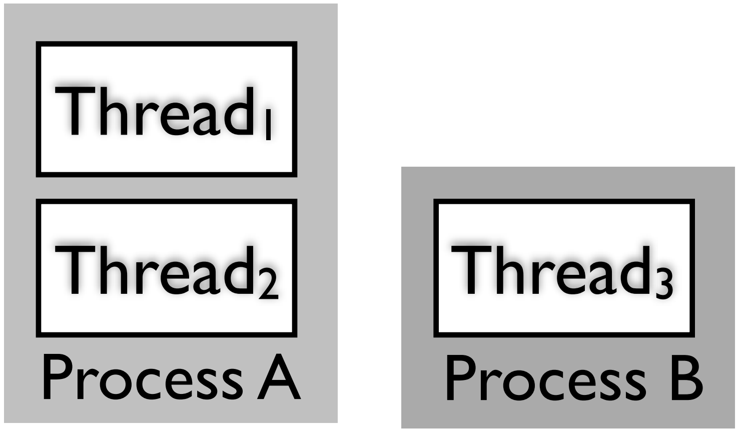
Deterministic Multiprocessing at 10,000’

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

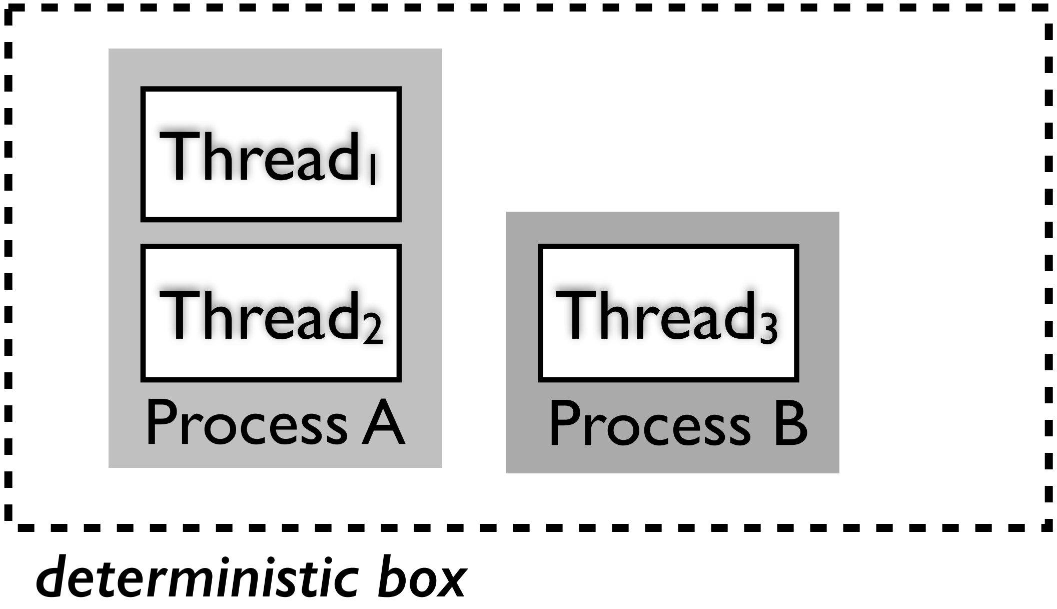
- DMP provides *execution-level* determinism for arbitrary multithreaded programs:
  - execution is only function of explicit inputs => single execution per input
  - this is not record-replay of multithreaded programs
- ***Key idea***: conceptually serialize execution, recover parallelism while preserving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



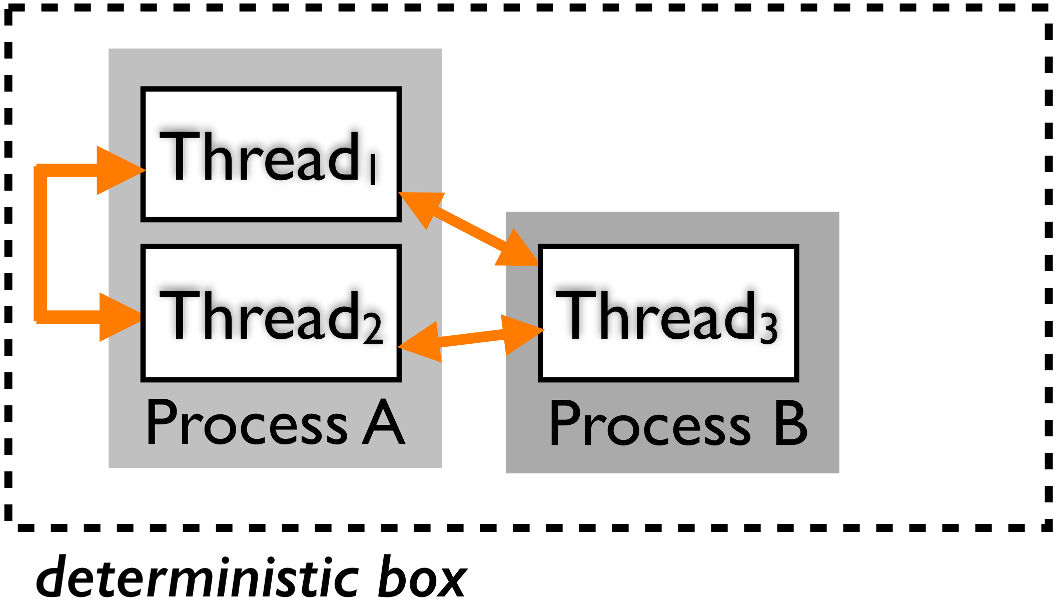
Deterministic Process Groups (DPGs)



Deterministic Process Groups (DPGs)



Deterministic Process Groups (DPGs)

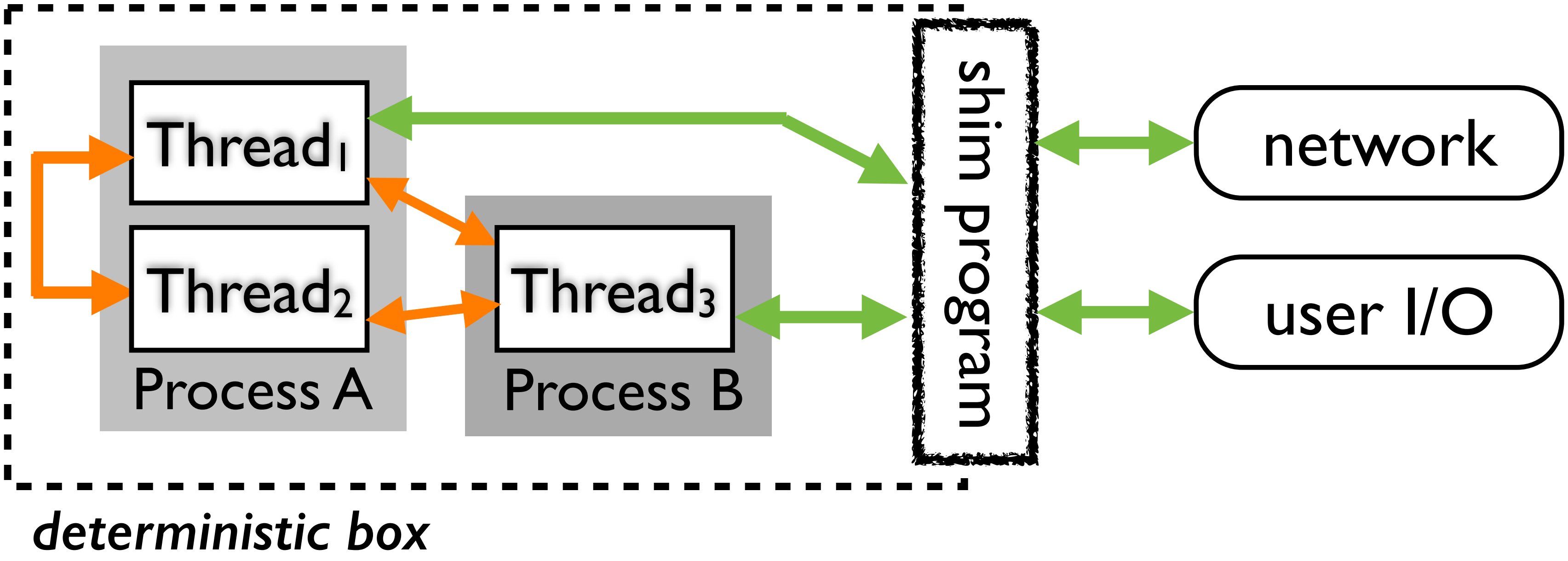


System ensures:

- *internal* nondeterminism is eliminated

  (for shared-memory, pipes, signals, local files, ...)

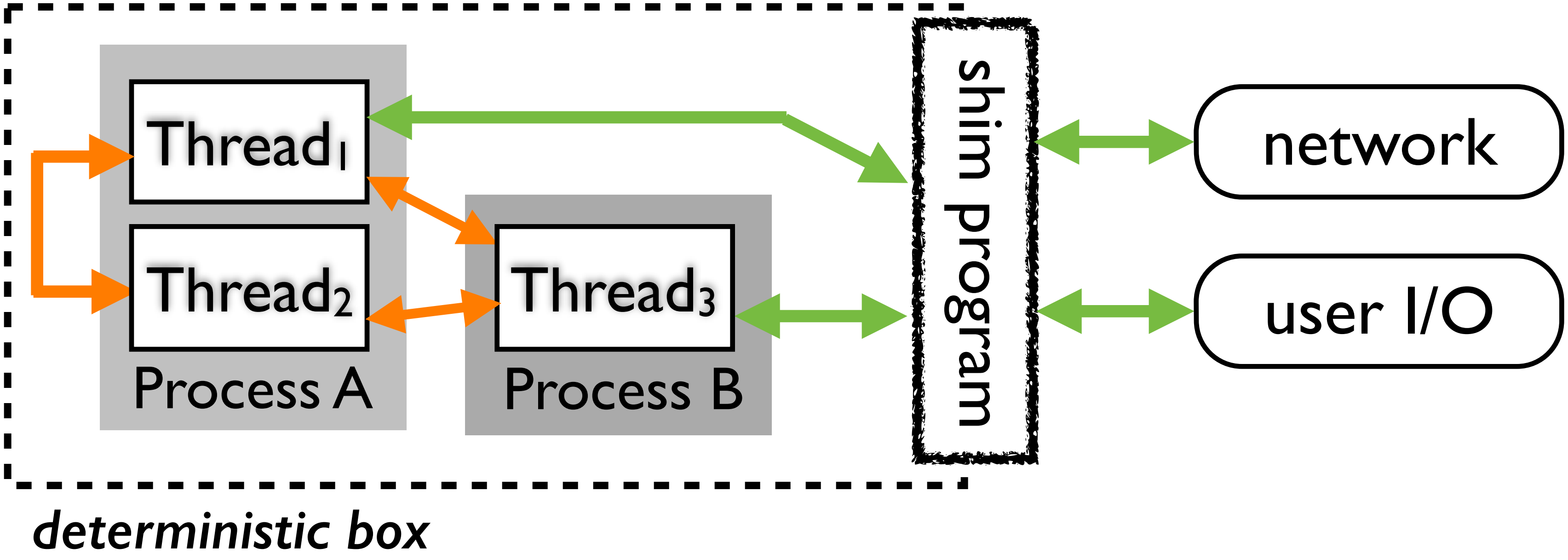
Deterministic Process Groups (DPGs)



System ensures:

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

Deterministic Process Groups (DPGs)



System ensures:

- *internal* nondeterminism is eliminated

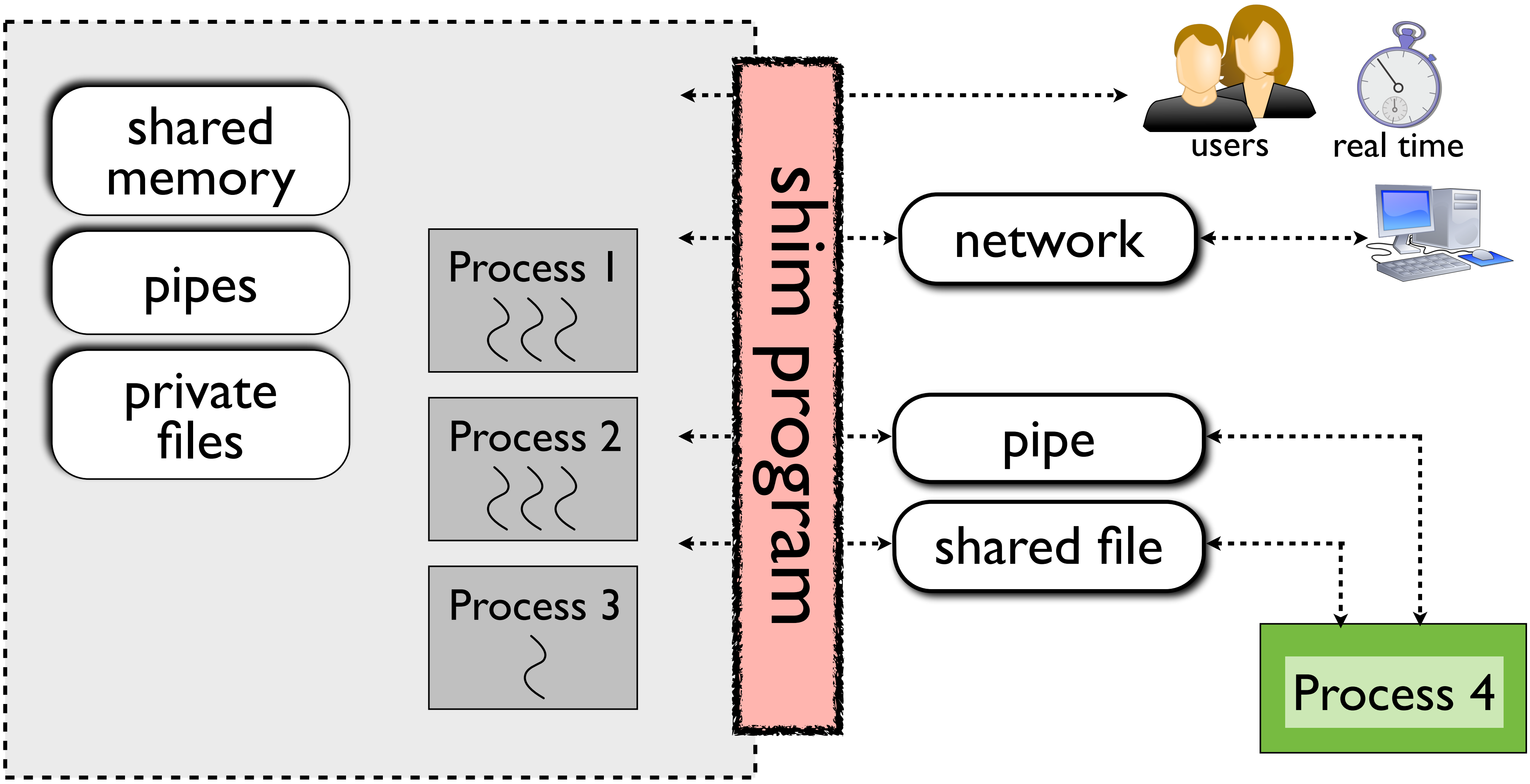
  (for shared-memory, pipes, signals, local files, ...)
- *external* nondeterminism funneled through *shim program*

Shim Program:

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

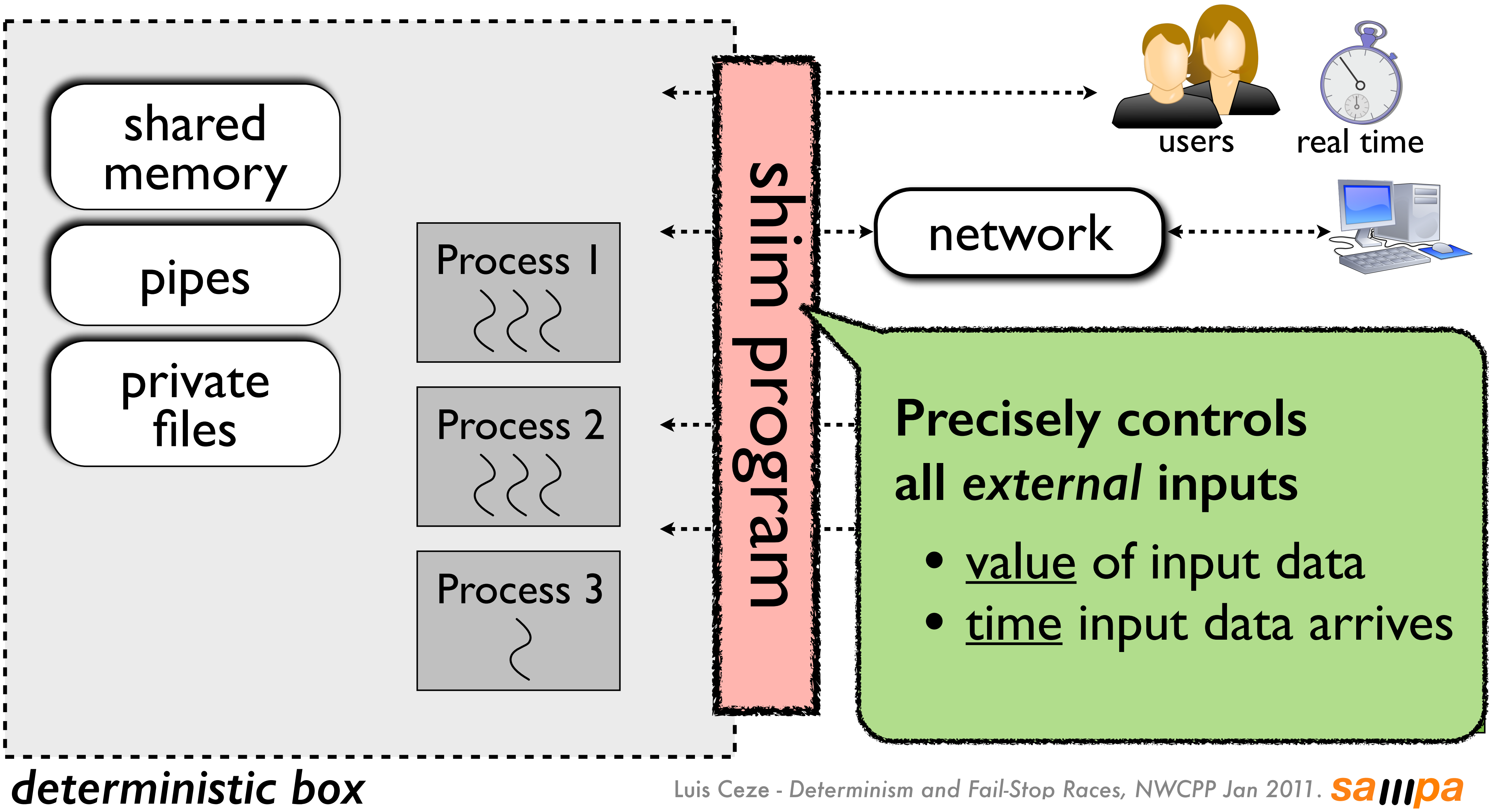
Internal Determinism

External Nondeterminism



deterministic box

Internal Determinism | External Nondeterminism

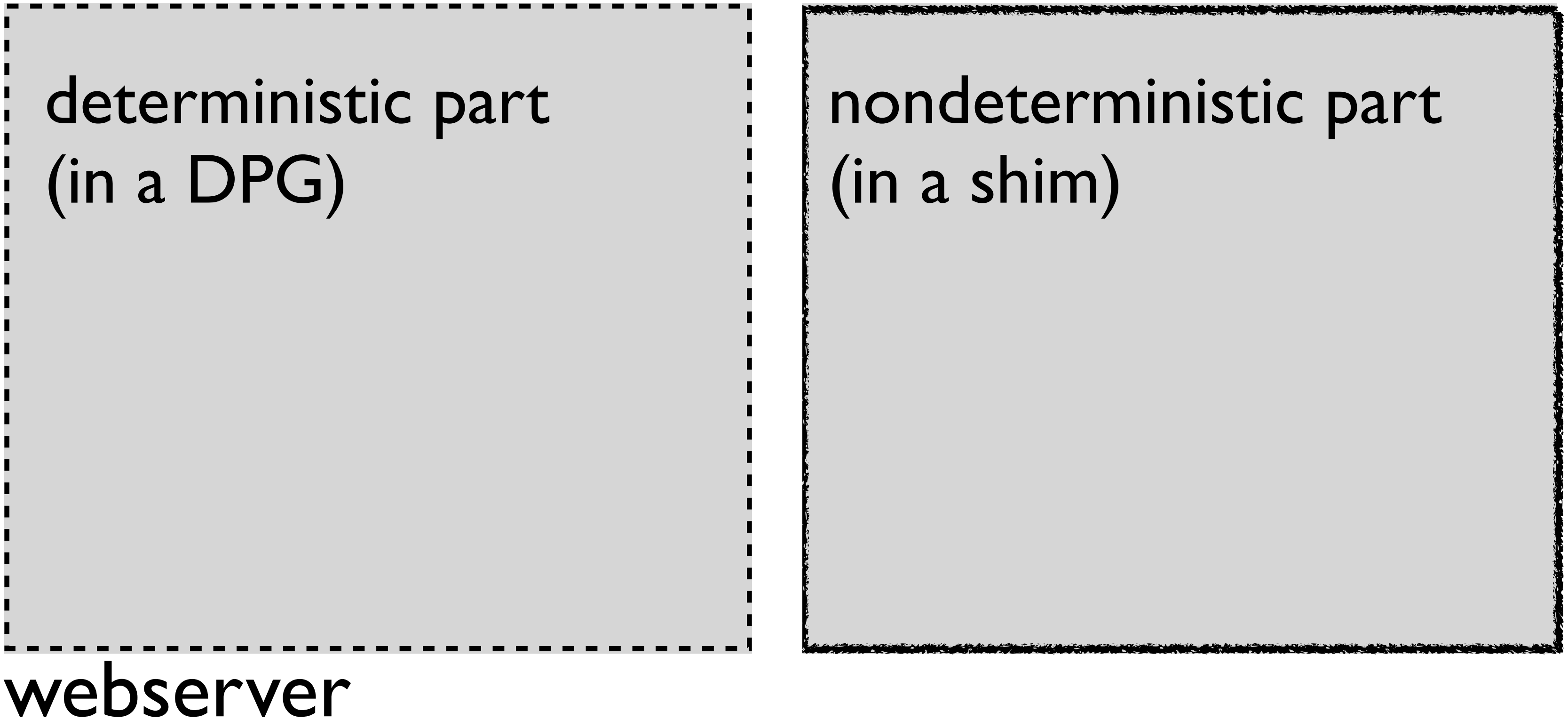


Precisely controls all *external* inputs

- <u>value</u> of input data
- <u>time</u> input data arrives

Aside: Using DPGs When Constructing Apps

Aside: Using DPGs When Constructing Apps

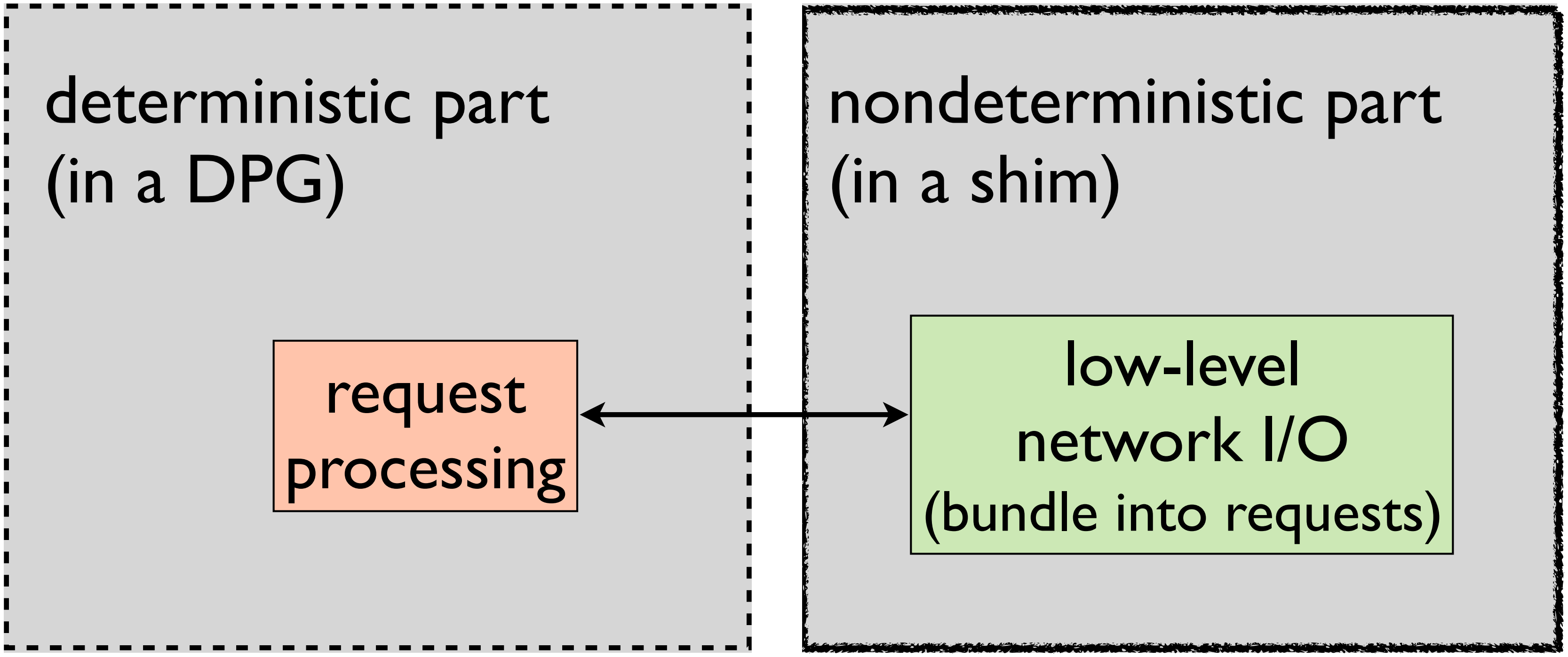


deterministic part
(in a DPG)

nondeterministic part
(in a shim)

webserver

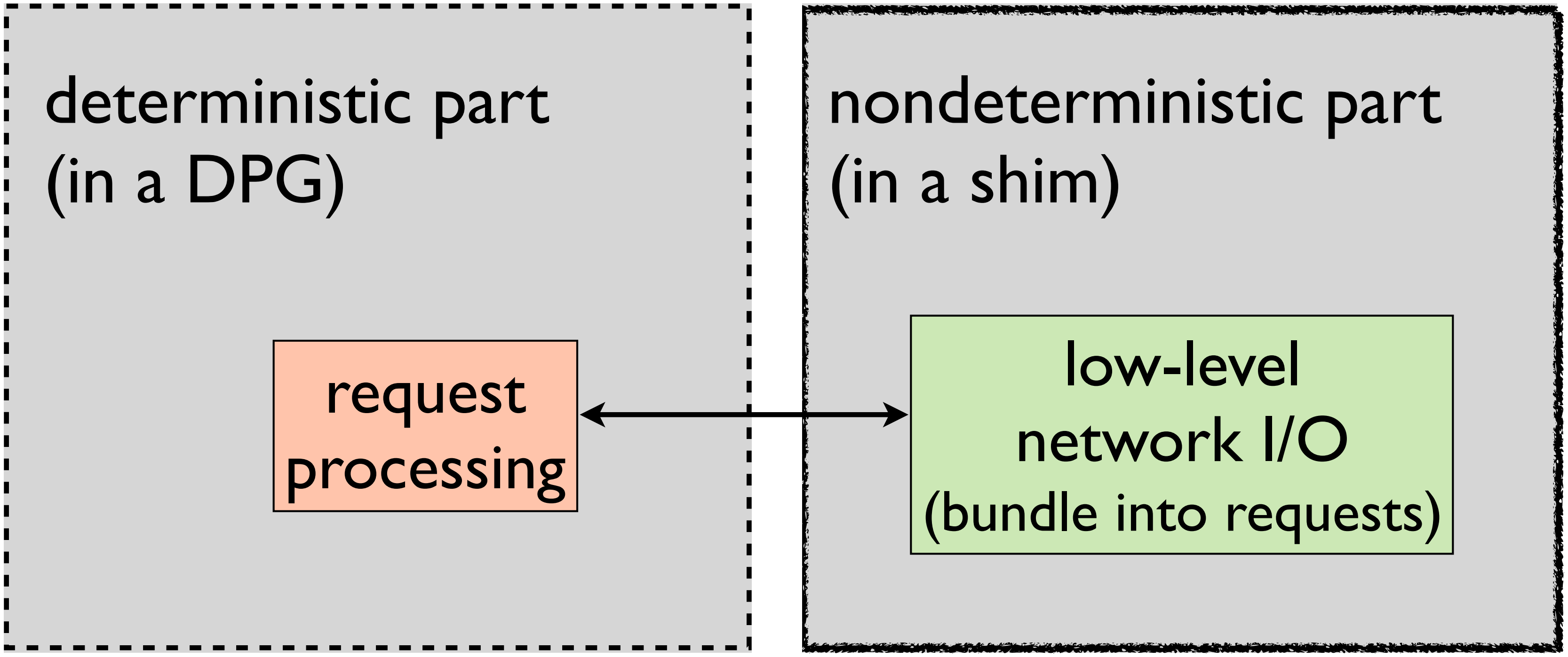
Aside: Using DPGs When Constructing Apps



webserver

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

Aside: Using DPGs When Constructing Apps



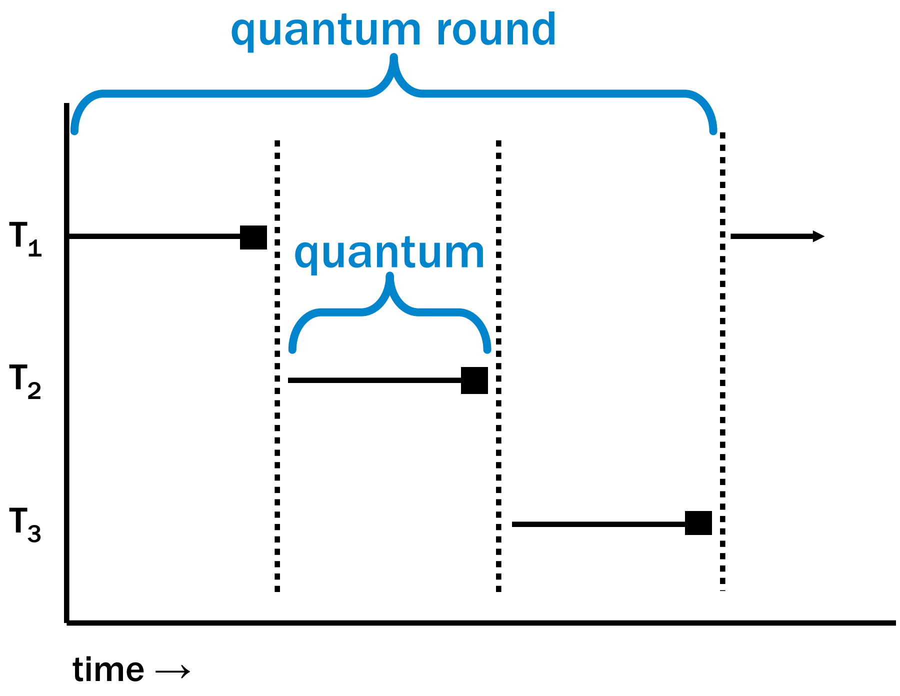
webserver

- 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?

Can we do better?

- Only need to serialize communicating instructions

Can we do better?

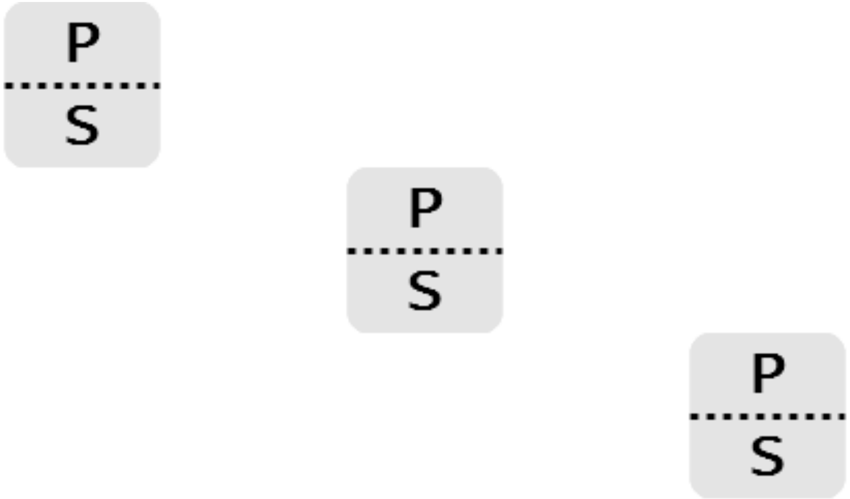
- Only need to serialize communicating instructions
- Break each quantum into communication-free parallel mode and communicative serial mode

Can we do better?

- Only need to serialize communicating instructions
- Break each quantum into communication-free parallel mode and communicative serial mode

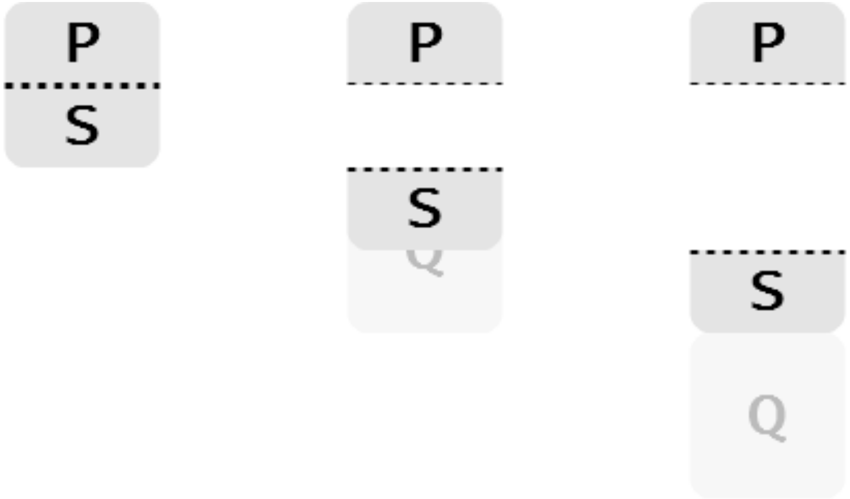
Can we do better?

- Only need to serialize communicating instructions
- Break each quantum into communication-free parallel mode and communicative serial mode



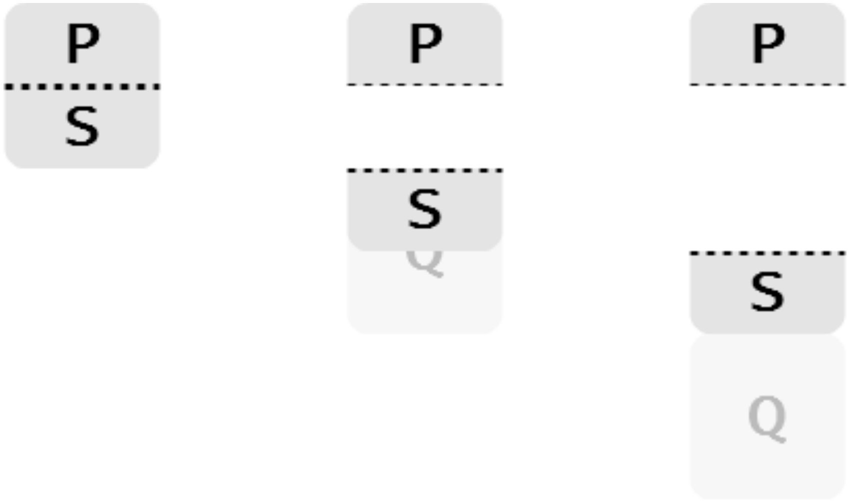
Can we do better?

- Only need to serialize communicating instructions
- Break each quantum into communication-free parallel mode and communicative serial mode

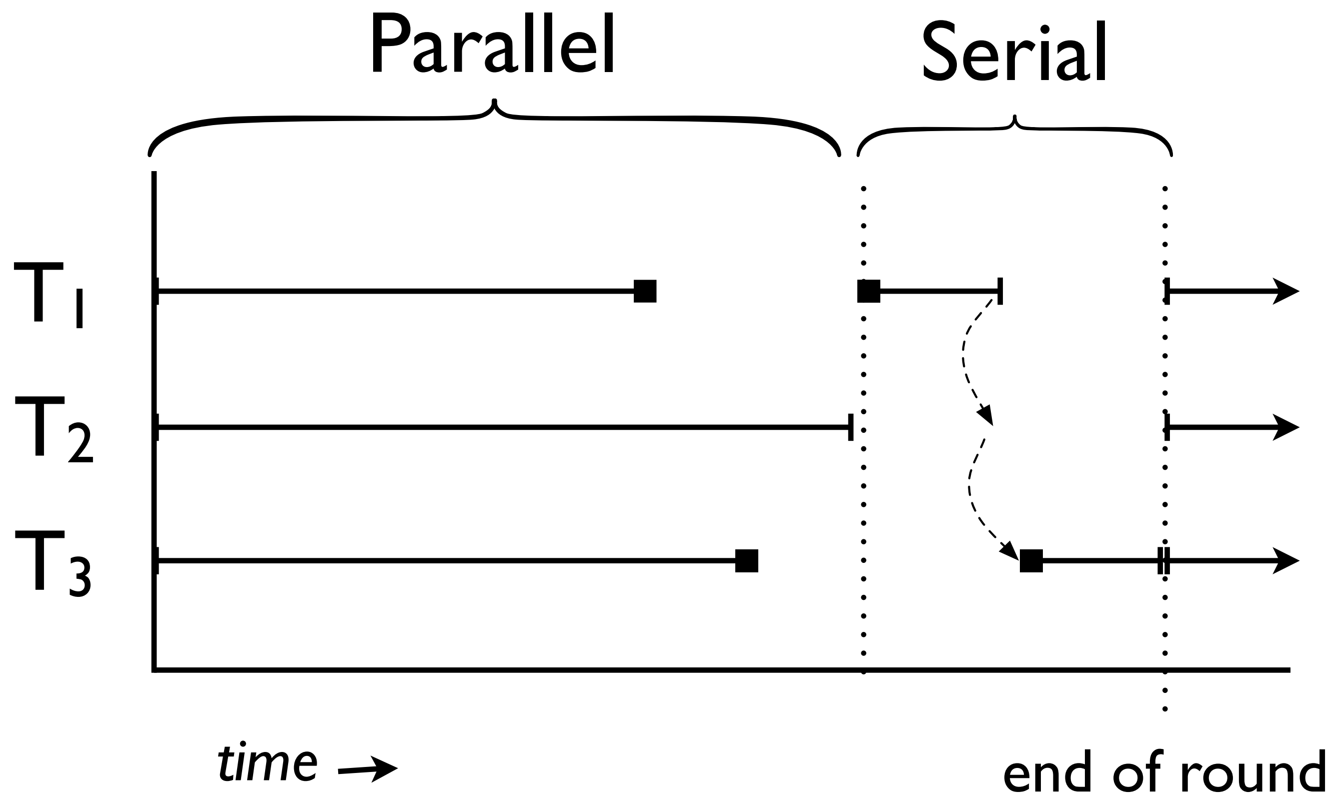


Can we do better?

- 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)

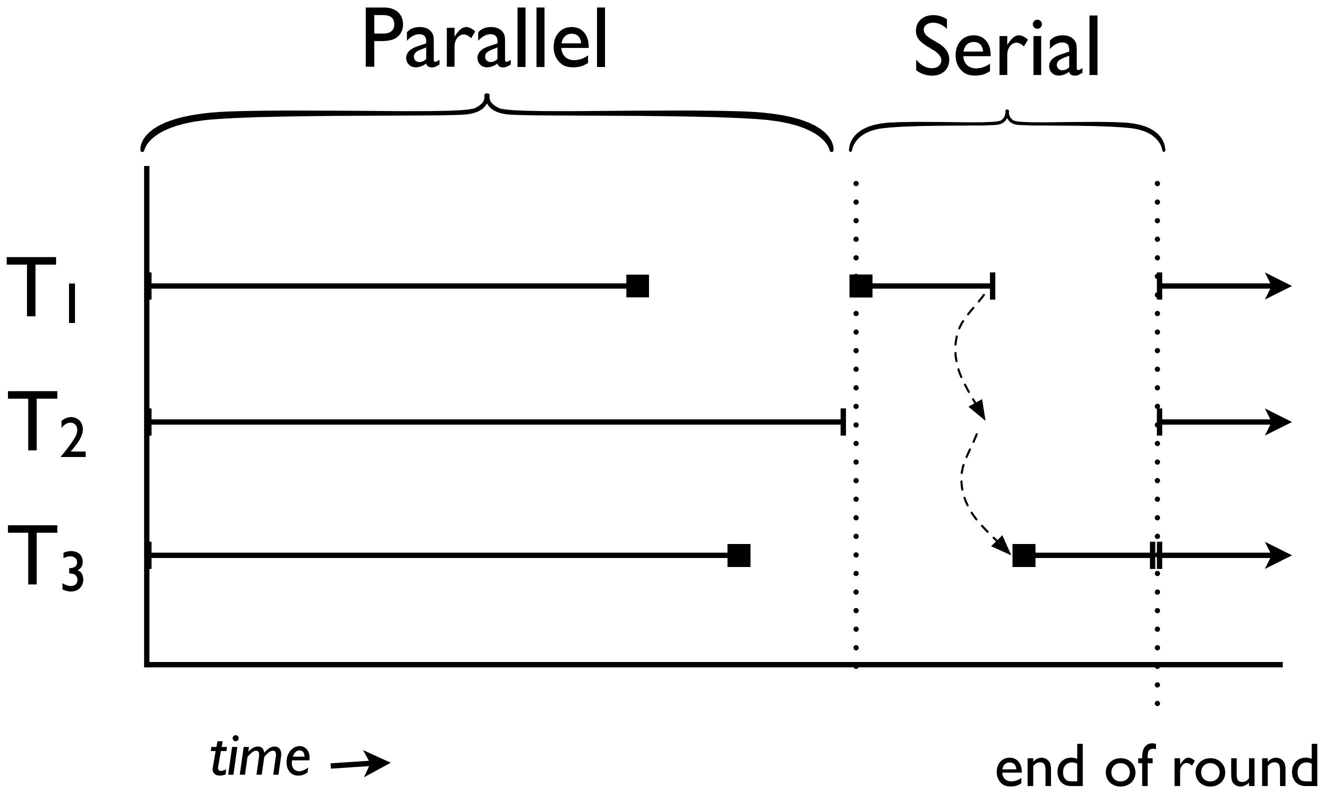


Parallel mode: no communication (can write only to private data)
Serial mode: arbitrary communication

DMP-O (Ownership)

MOT

x	owned-by T_1
y	shared
z	owned-by T_2
⋮	⋮



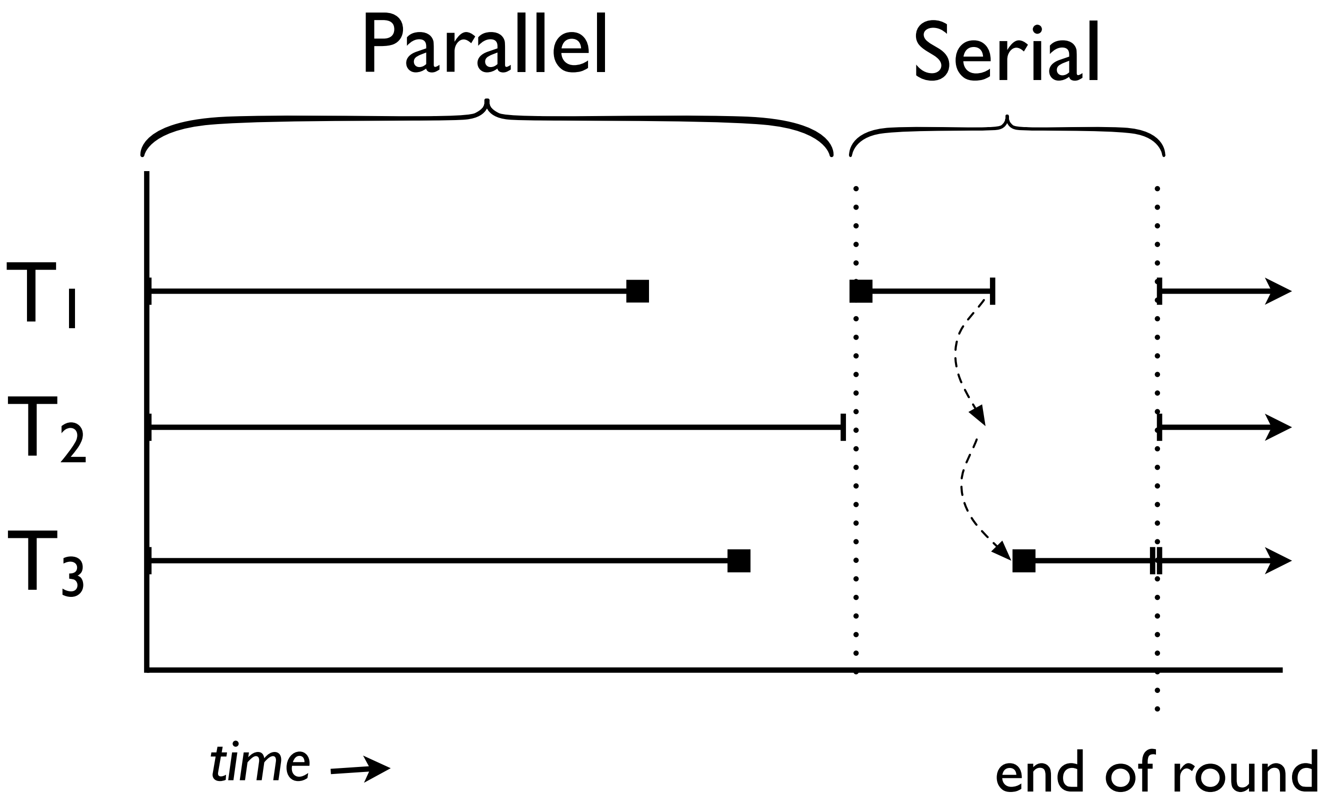
Parallel mode: no communication (can write only to private data)

Serial mode: arbitrary communication

DMP-O (Ownership)

MOT

x	owned-by T_1
y	shared
z	owned-by T_2
⋮	⋮



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

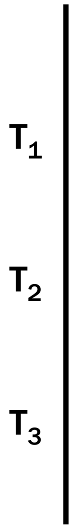
DMP-TM: Recovering Parallelism with Speculation

- DMP-O conservatively assumes that all cache line state transitions are communication
    - ...but **many transitions are not communication**

DMP-TM: Recovering Parallelism with Speculation

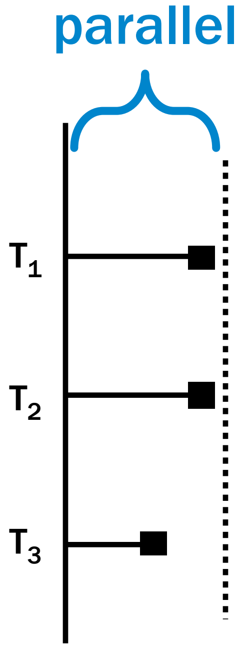
- DMP-O conservatively assumes that all cache line state transitions are communication
  - …but **many transitions are not communication**
- Use TM support to speculate that a quantum is not involved in communication
  - If communication happens, rollback + re-execute
  - Commit quanta in a deterministic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

DMP-TM



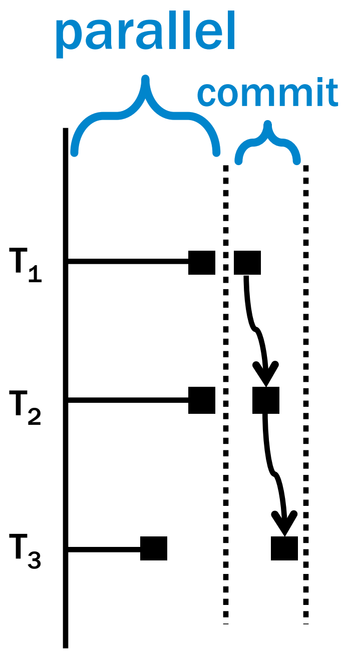
- quanta are implicit transactions
- commit quanta in deterministic order

DMP-TM



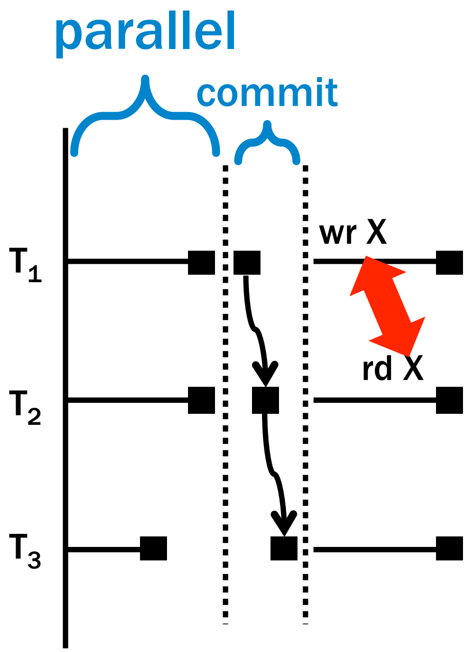
- quanta are implicit transactions
- commit quanta in deterministic order

DMP-TM



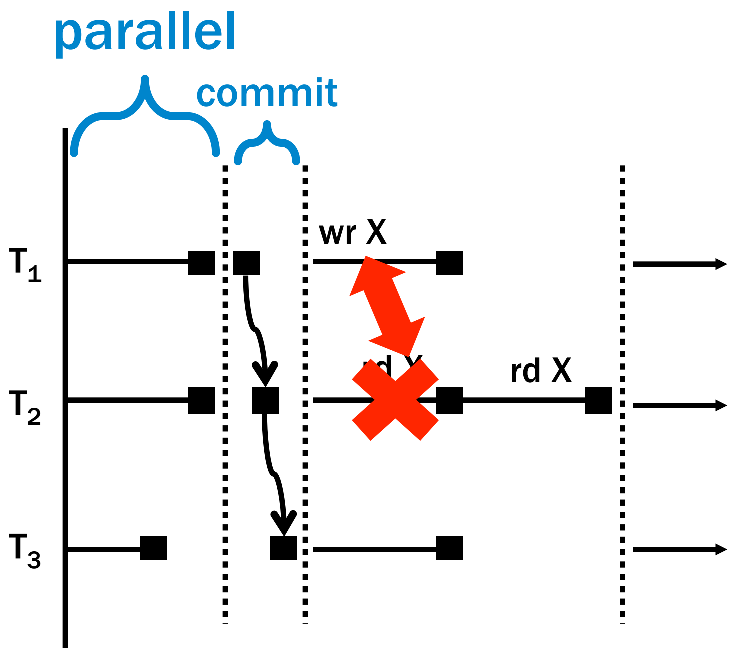
- quanta are implicit transactions
- commit quanta in deterministic order

DMP-TM



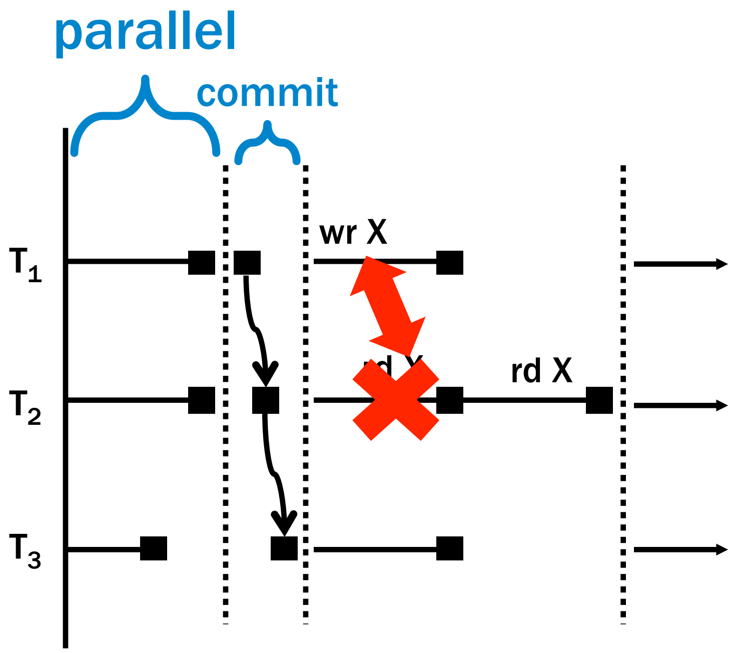
- quanta are implicit transactions
- commit quanta in deterministic order
- rollback+restart on conflicts

DMP-TM



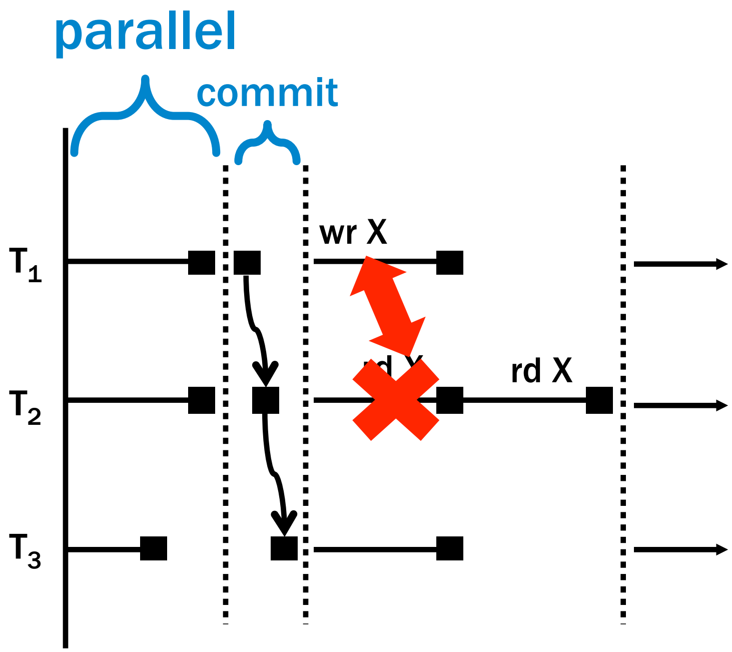
- quanta are implicit transactions
- commit quanta in deterministic order
- rollback+restart on conflicts

DMP-TM



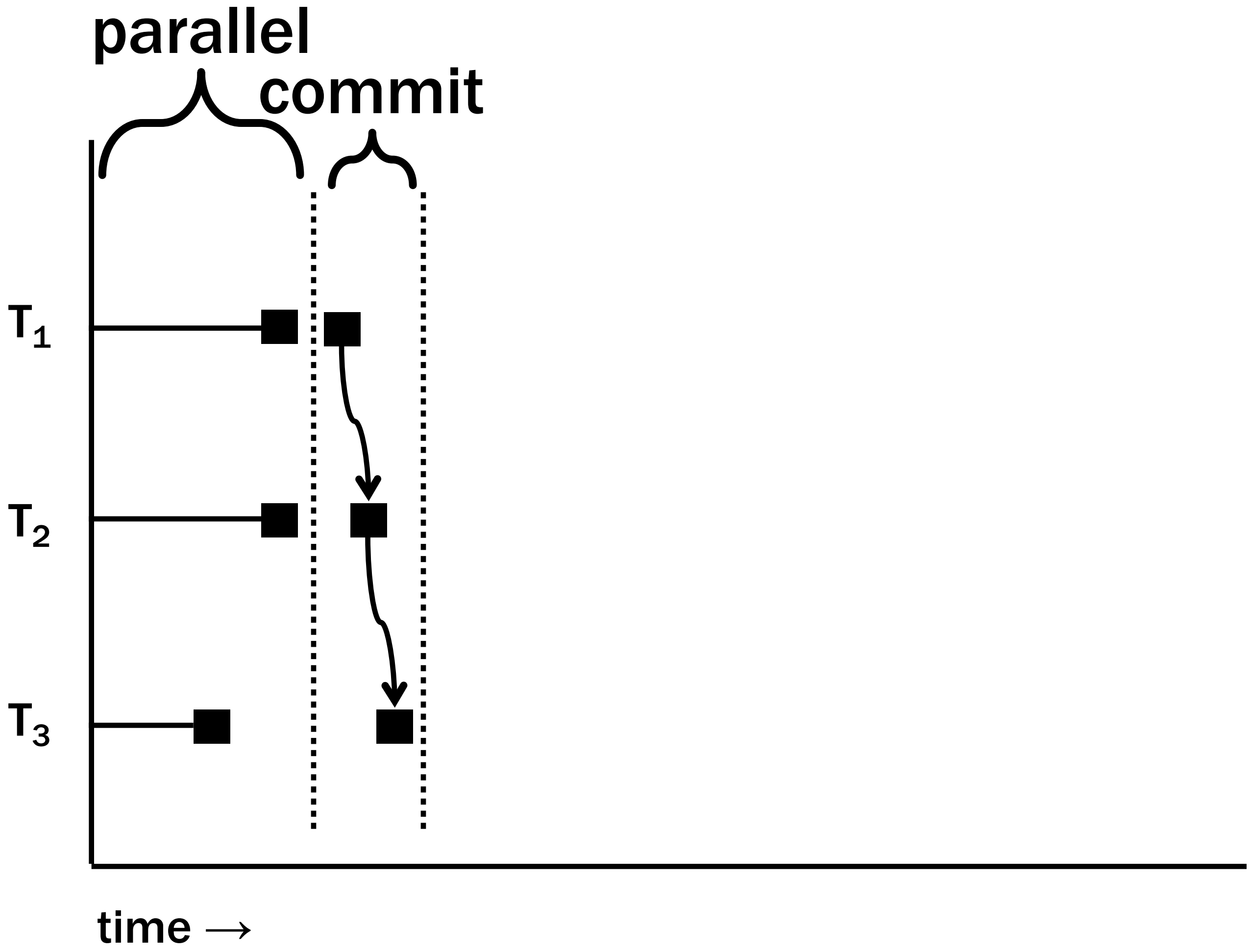
- quanta are implicit transactions
- commit quanta in deterministic order
- rollback+restart on conflicts
- leverage (best effort) HTM support

DMP-TM

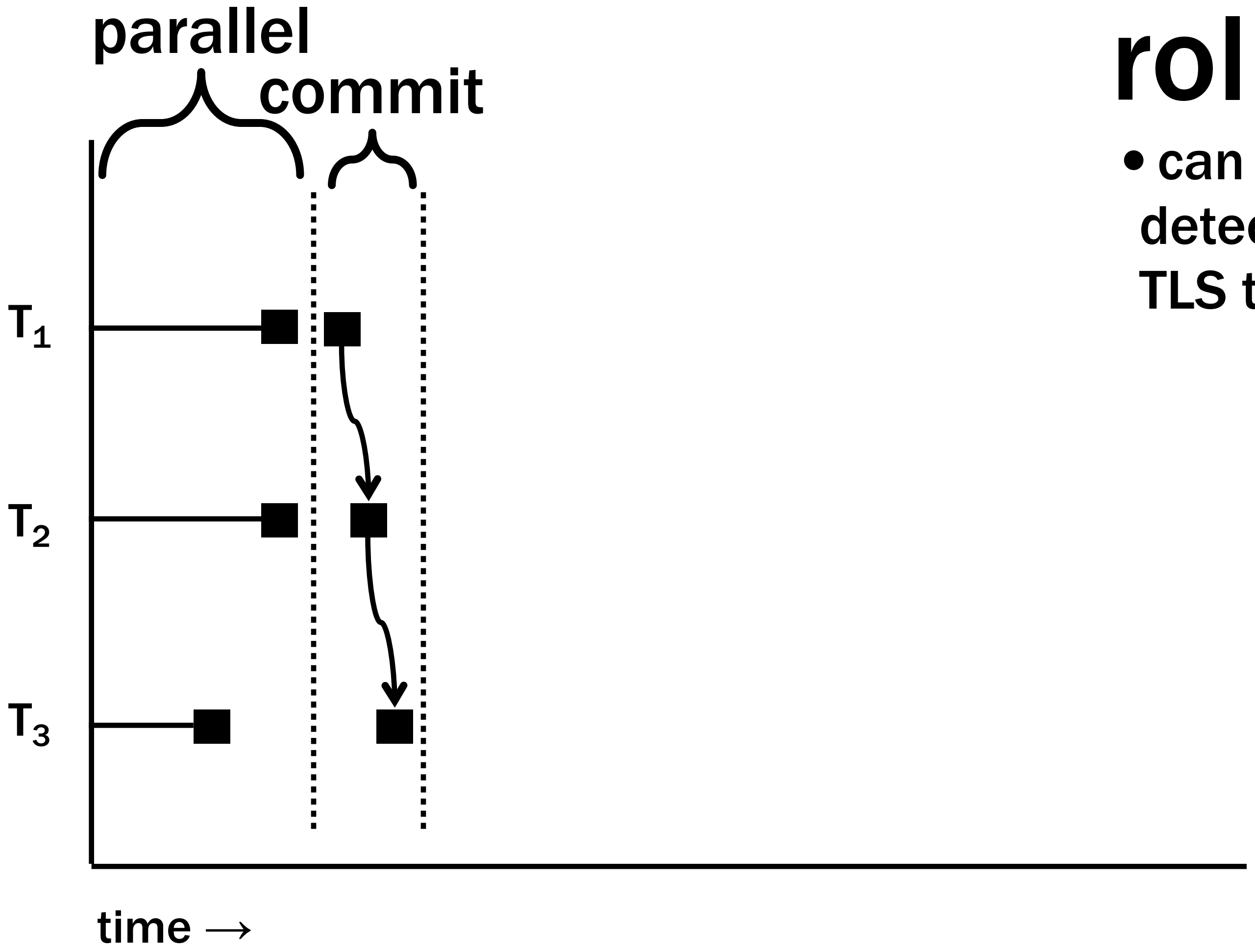


- quanta are implicit transactions
- commit quanta in deterministic order
- rollback+restart on conflicts
- leverage (best effort) HTM support
- functionally equivalent to DMP-Serial

DMP-TM Overheads



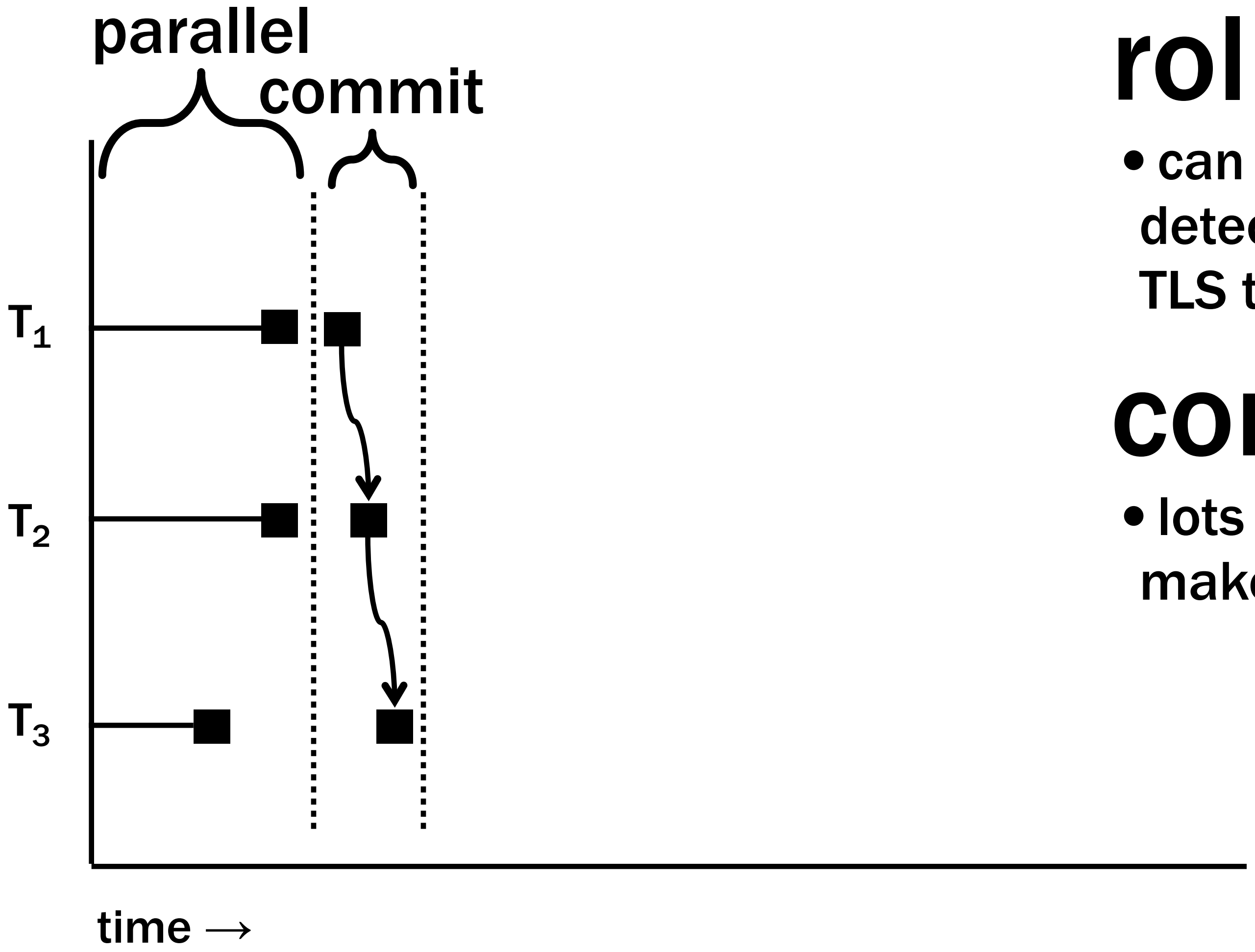
DMP-TM Overheads



rollbacks

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

DMP-TM Overheads



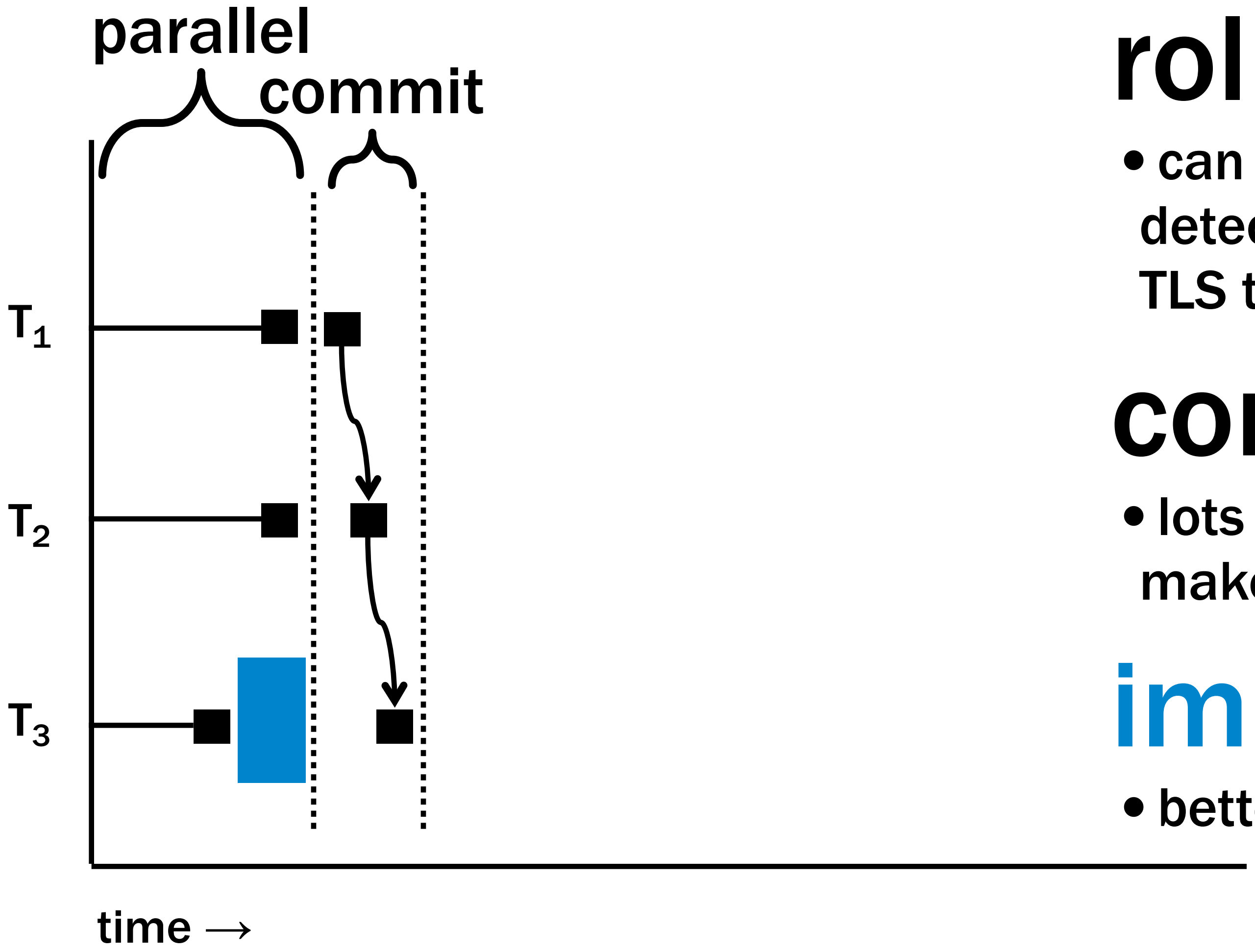
rollbacks

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

commit

- lots of TM techniques to make commit fast

DMP-TM Overheads



rollbacks

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

commit

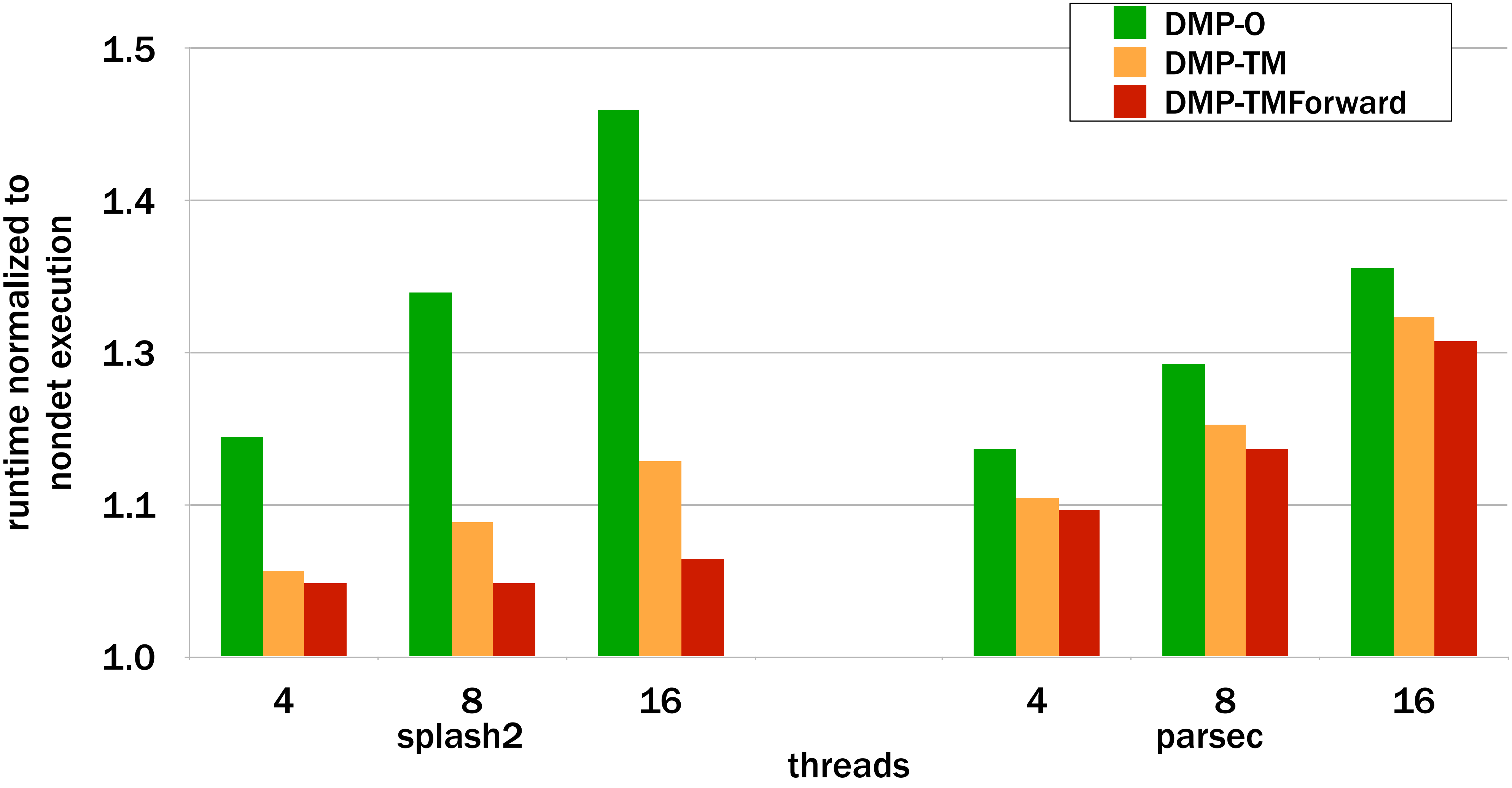
- lots of TM techniques to make commit fast

imbalance

- better quantum formation

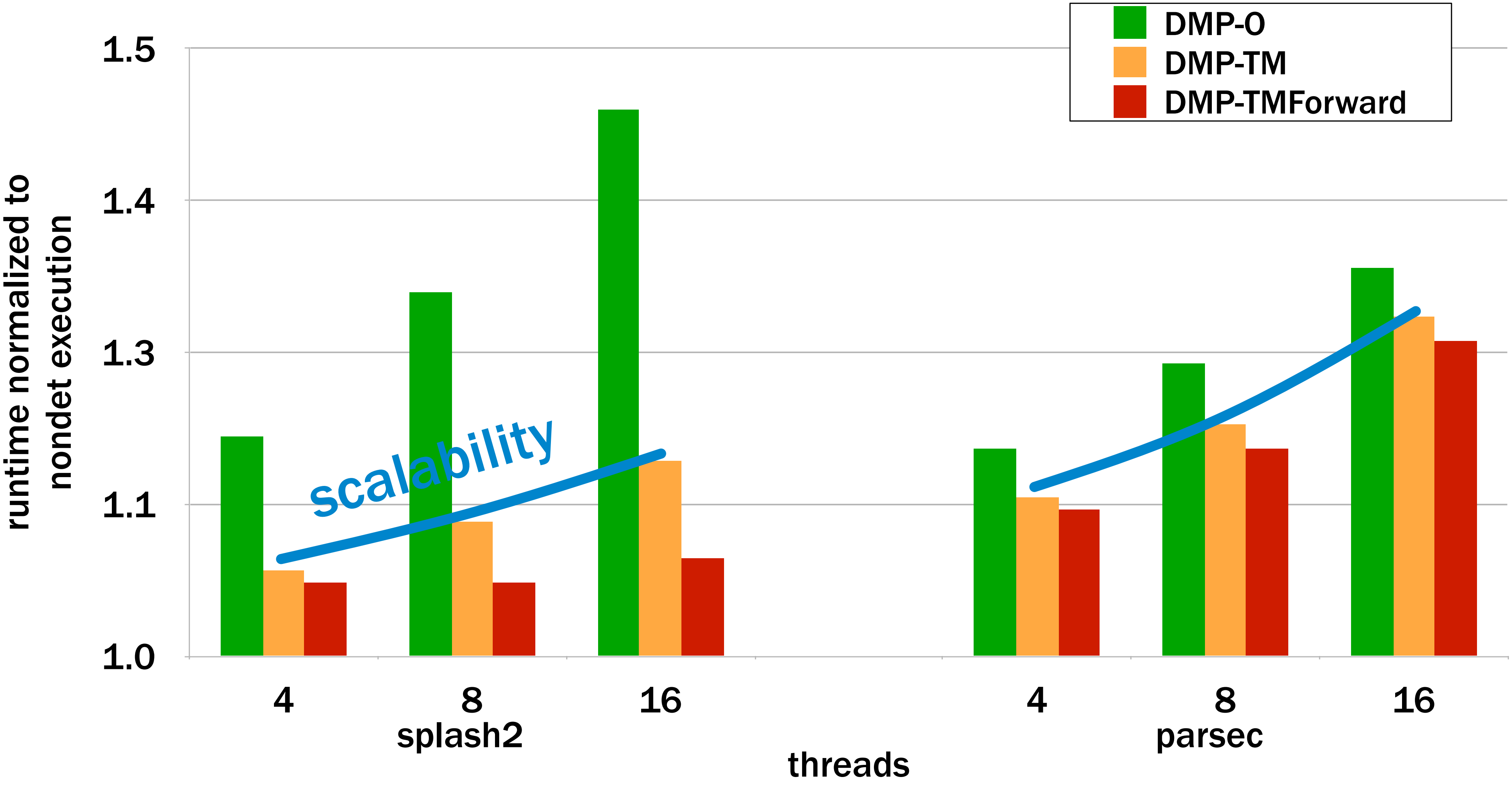
DMP-O and DMP-TM Evaluation

(HW version)



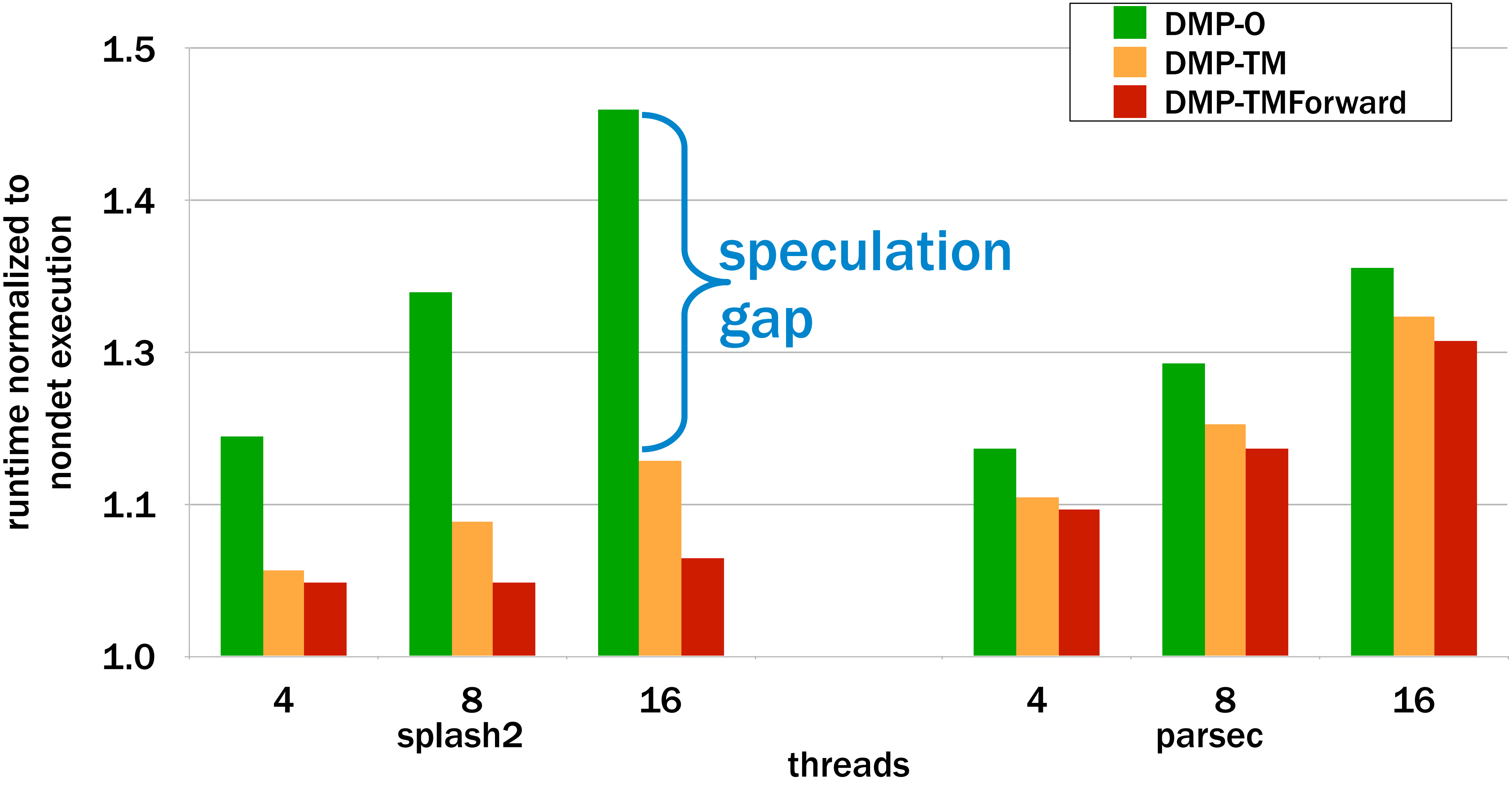
DMP-O and DMP-TM Evaluation

(HW version)



DMP-O and DMP-TM Evaluation

(HW version)



Can we get most of the benefits of speculation *without* the costs of speculation?

Can we get most of the benefits of speculation *without* the costs of speculation?

transactions

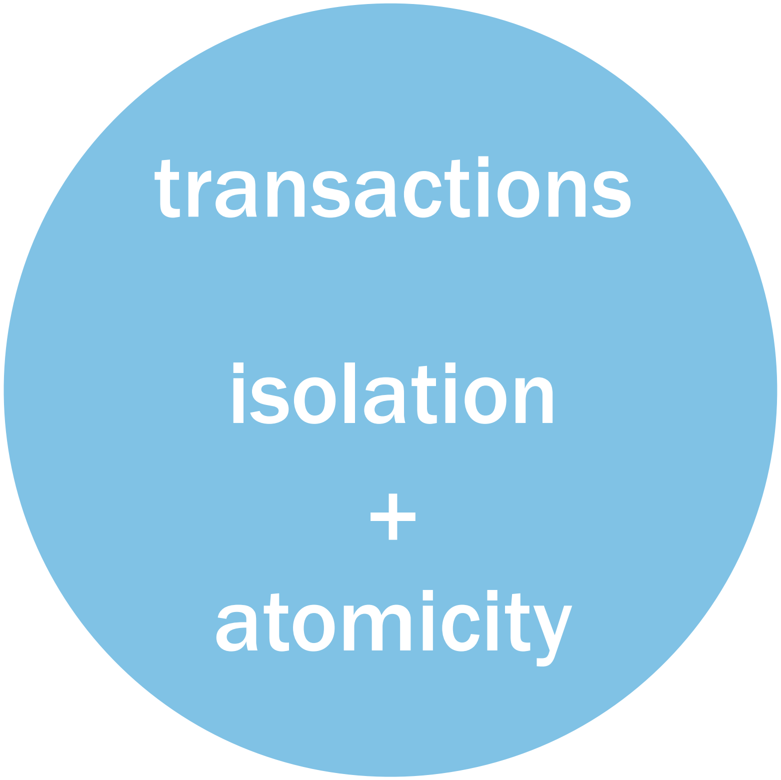
isolation

+

atomicity

Can we get most of the benefits of speculation *without* the costs of speculation?

isolation is sufficient for determinism

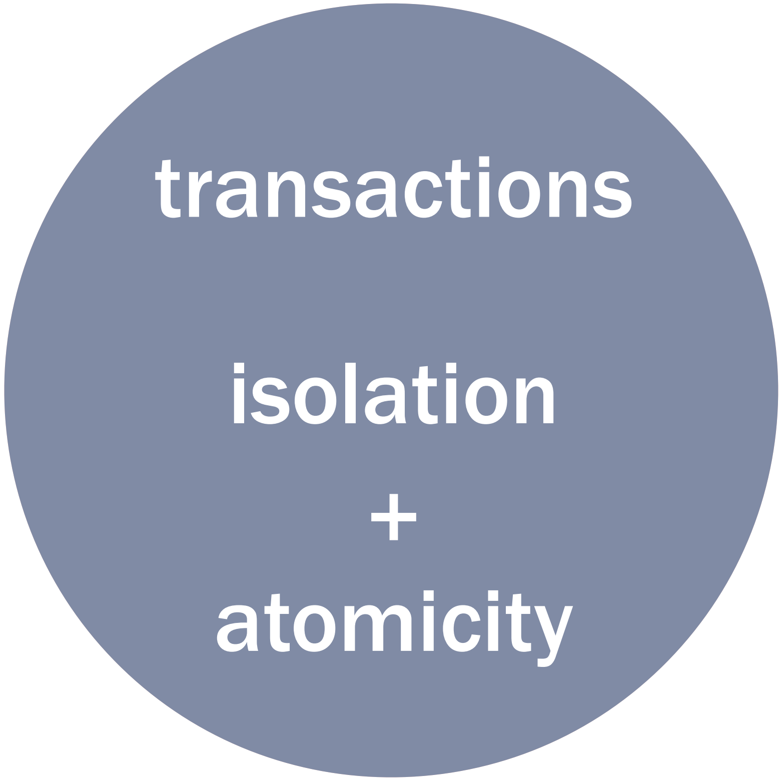


transactions

isolation
+
atomicity

Can we get most of the benefits of speculation *without* the costs of speculation?

isolation is sufficient for determinism



transactions

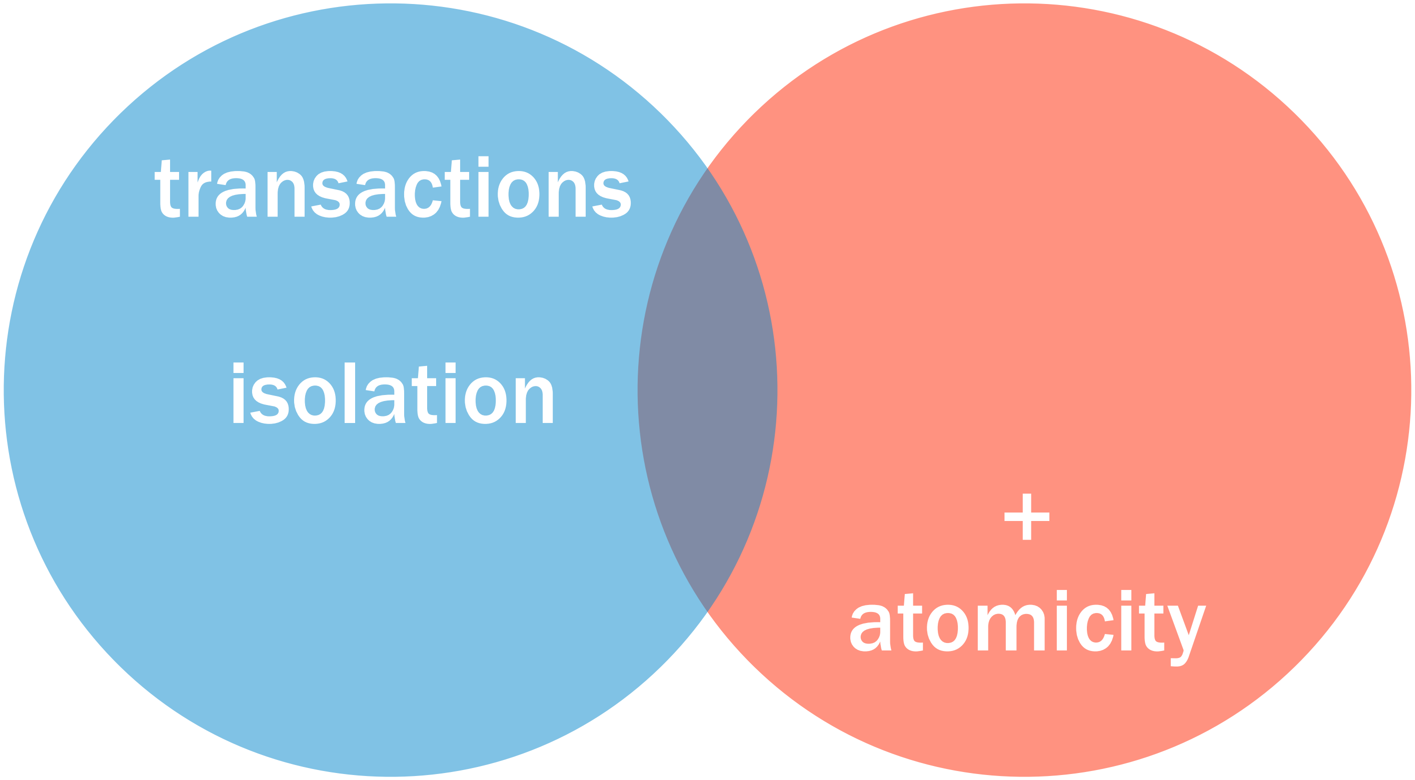
isolation

+

atomicity

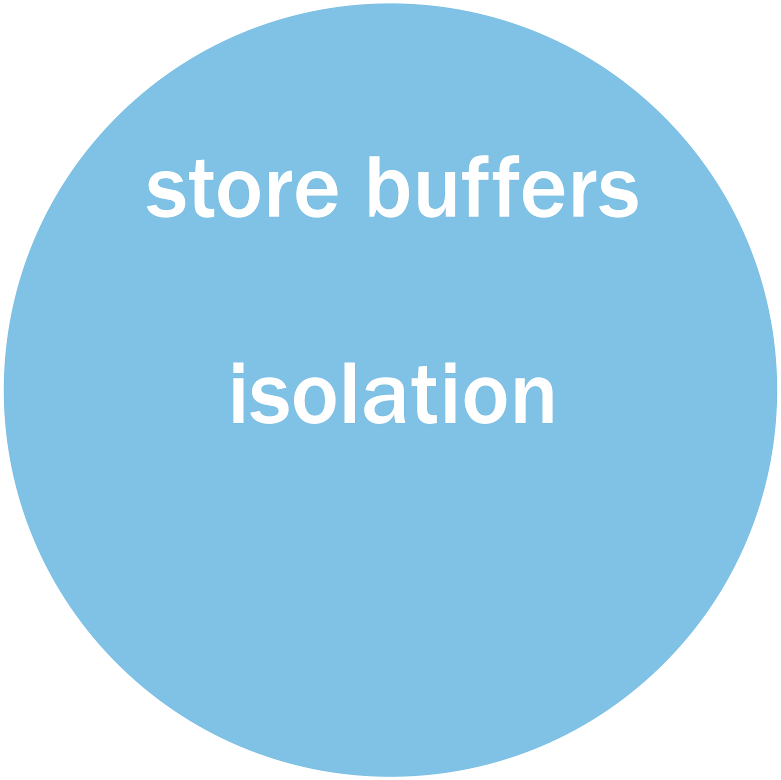
Can we get most of the benefits of speculation *without* the costs of speculation?

isolation is sufficient for determinism



Can we get most of the benefits of speculation *without* the costs of speculation?

isolation is sufficient for determinism



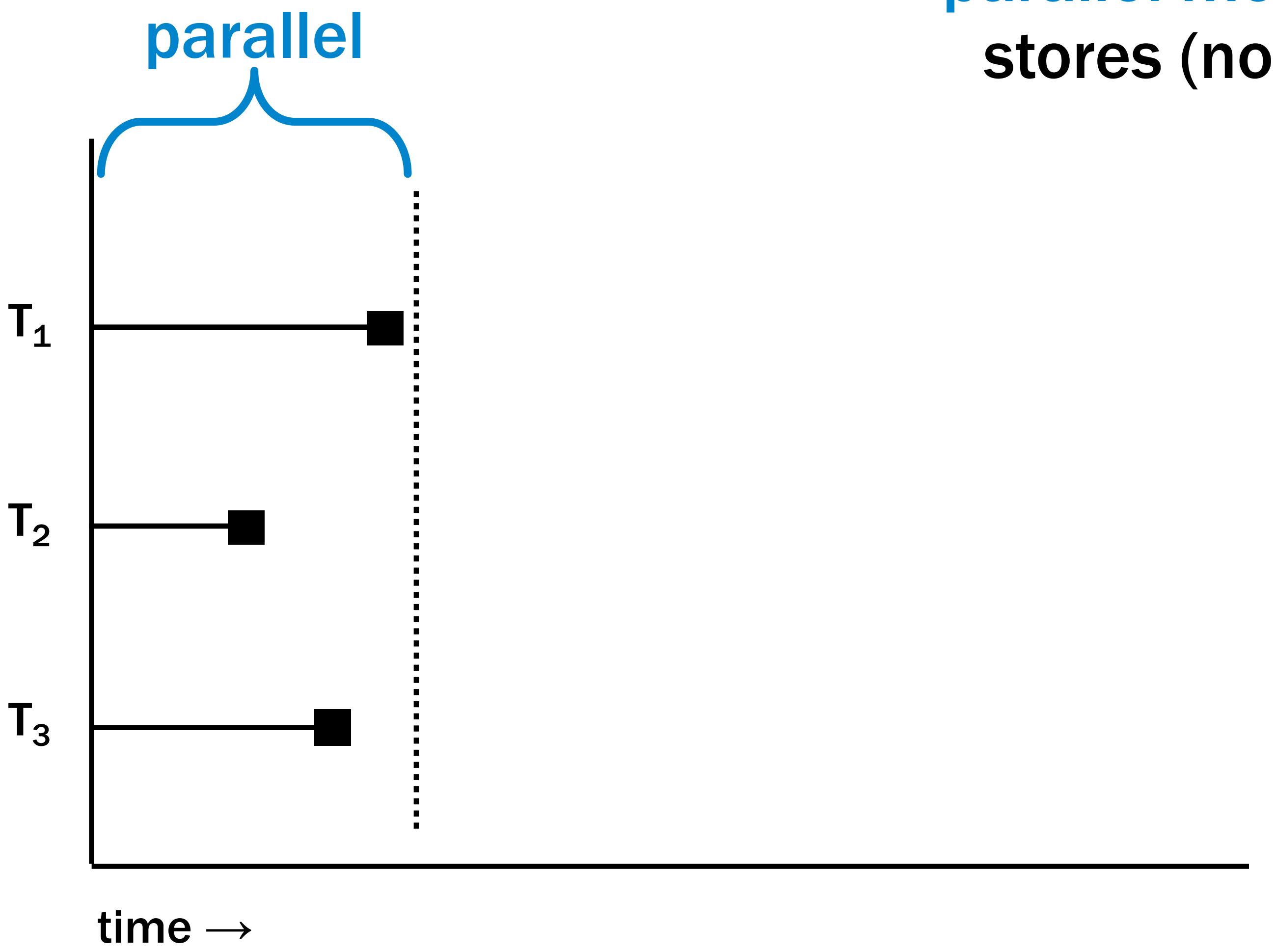
store buffers

isolation

DMP-Buffering: Trading consistency for performance

heard that before? :)

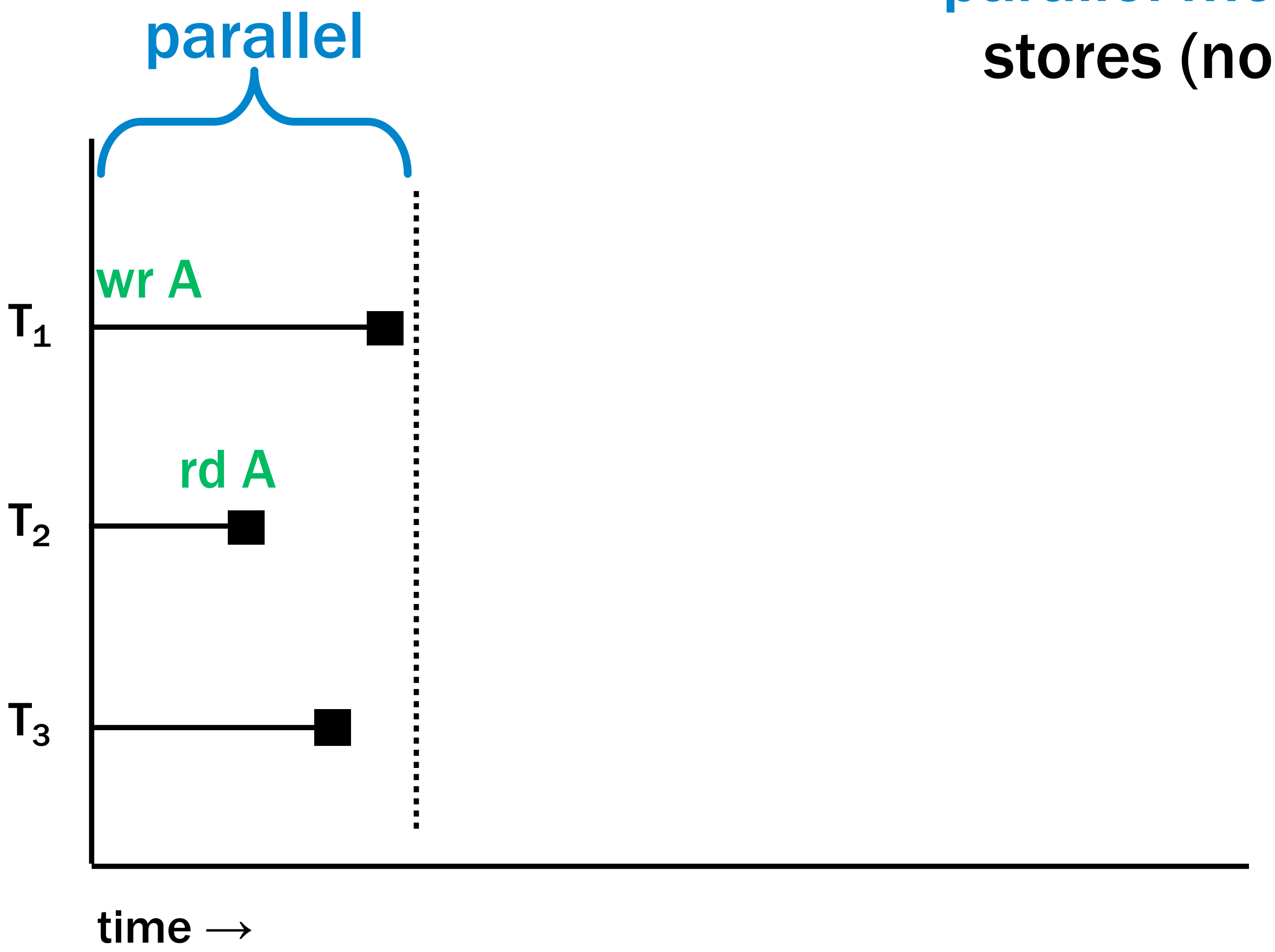
parallel mode: buffer all stores (no communication)



DMP-Buffering: Trading consistency for performance

heard that before? :)

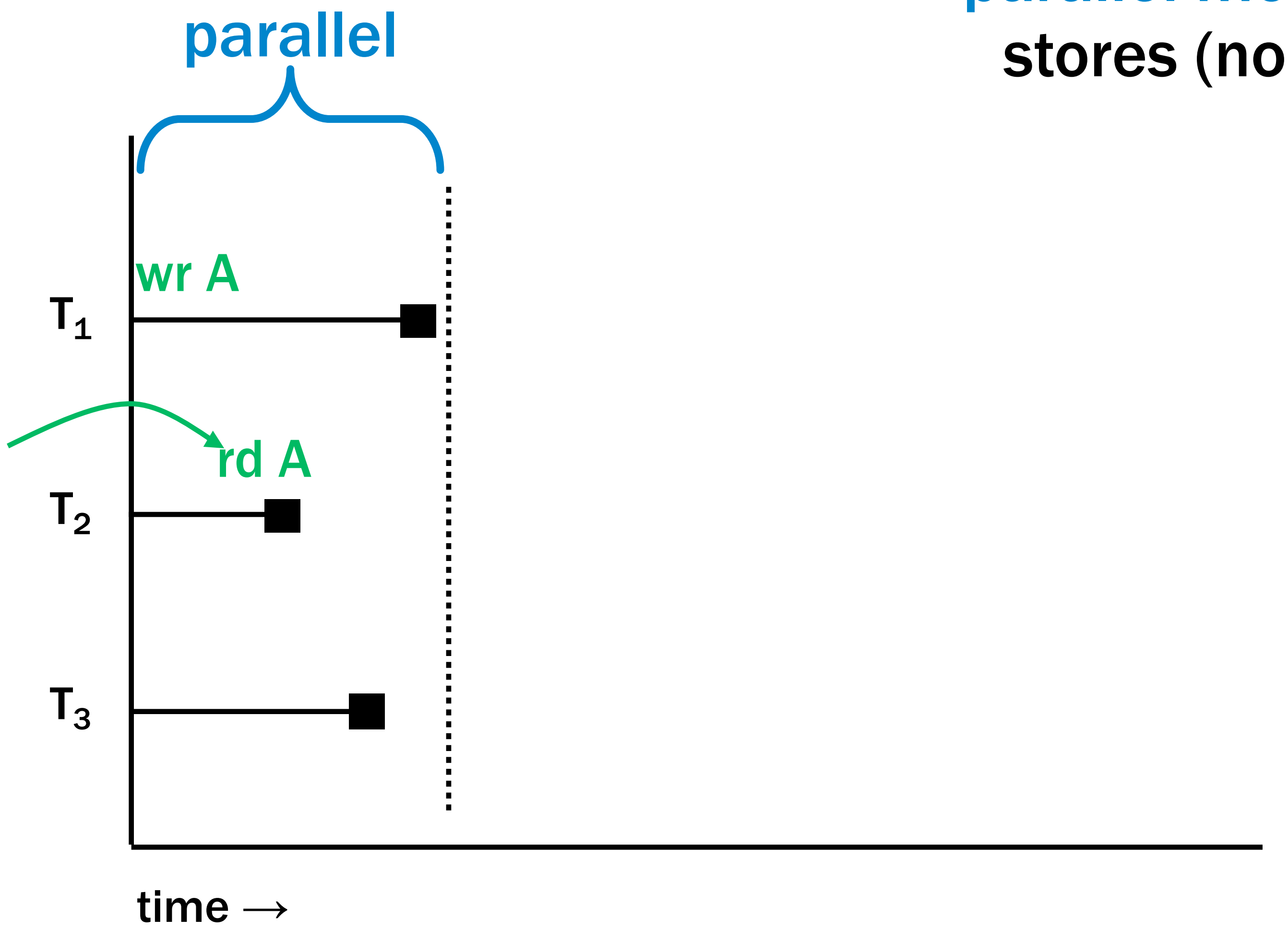
parallel mode: buffer all stores (no communication)



DMP-Buffering: Trading consistency for performance

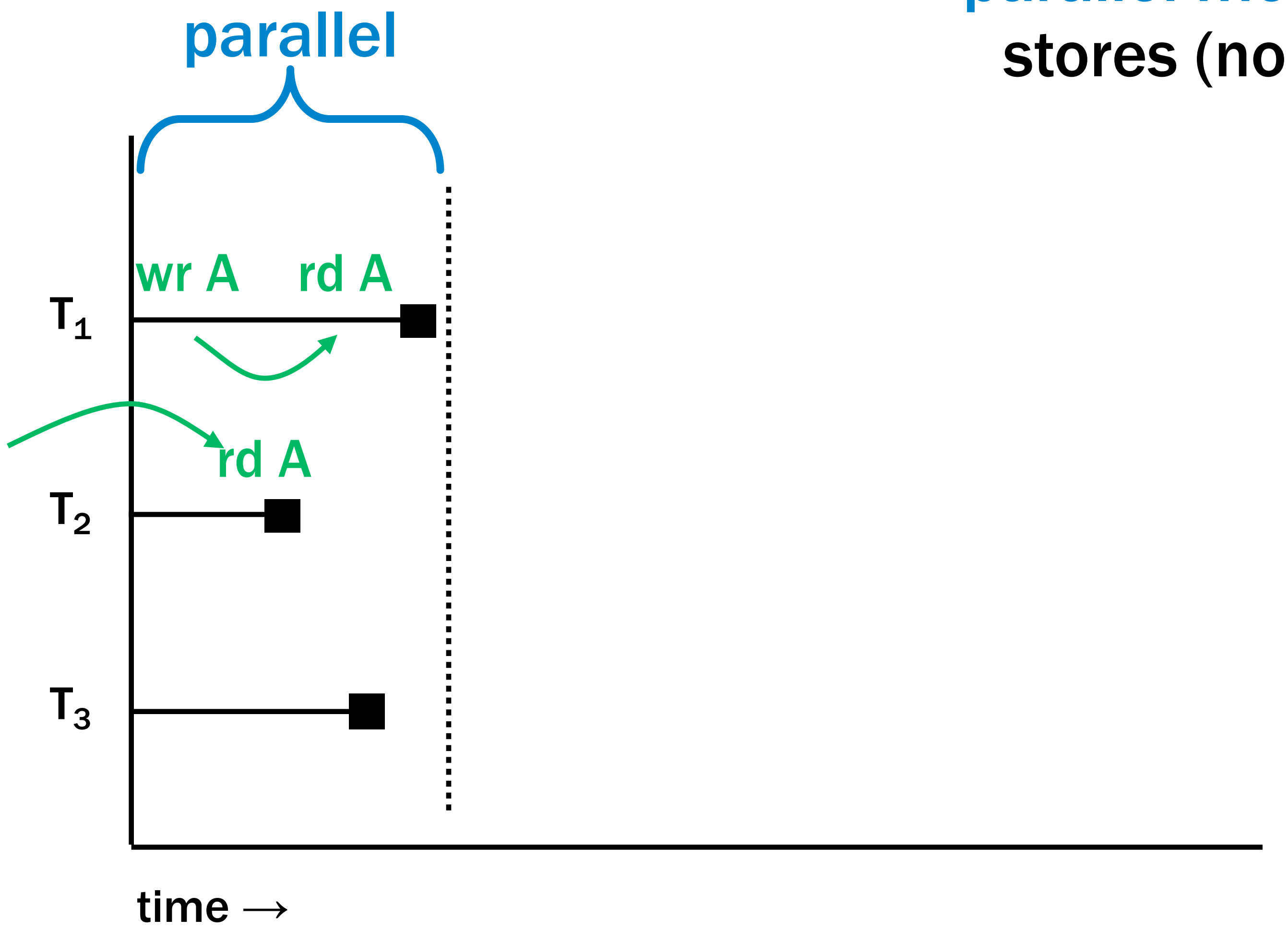
heard that before? :)

parallel mode: buffer all stores (no communication)



DMP-Buffering: Trading consistency for performance

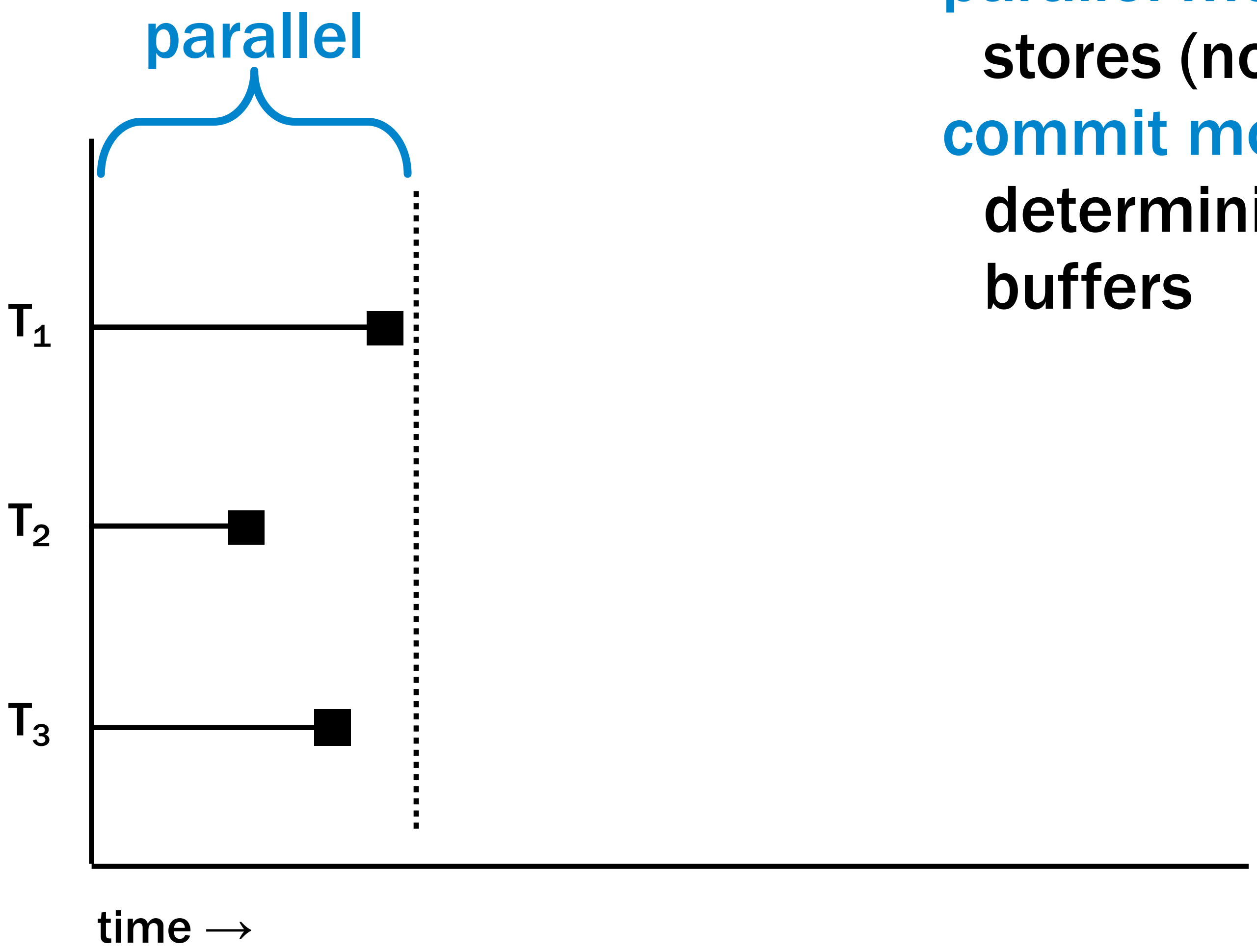
heard that before? :)



parallel mode: buffer all stores (no communication)

DMP-Buffering: Trading consistency for performance

heard that before? :)

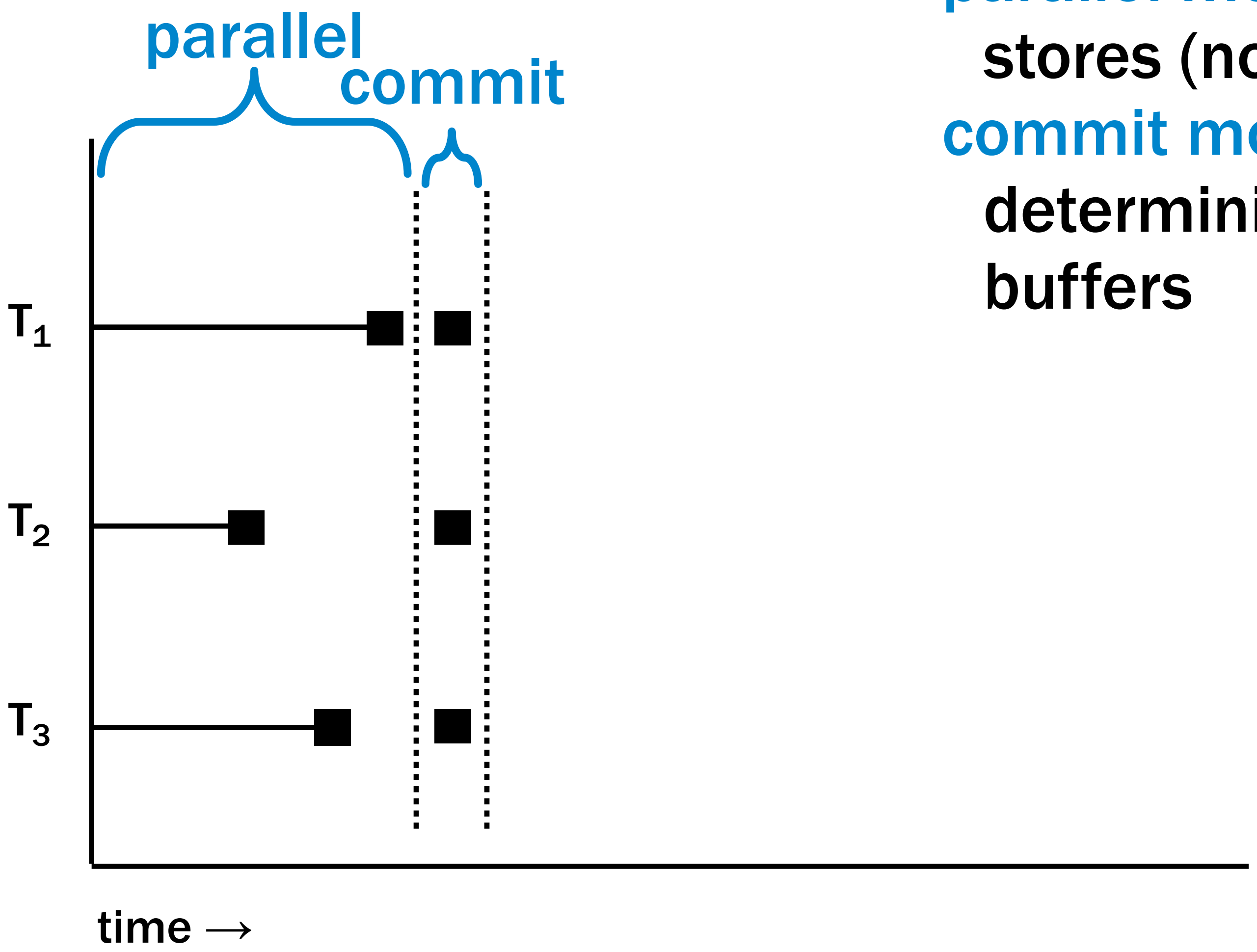


parallel mode: buffer all stores (no communication)

commit mode: deterministically publish buffers

DMP-Buffering: Trading consistency for performance

heard that before? :)

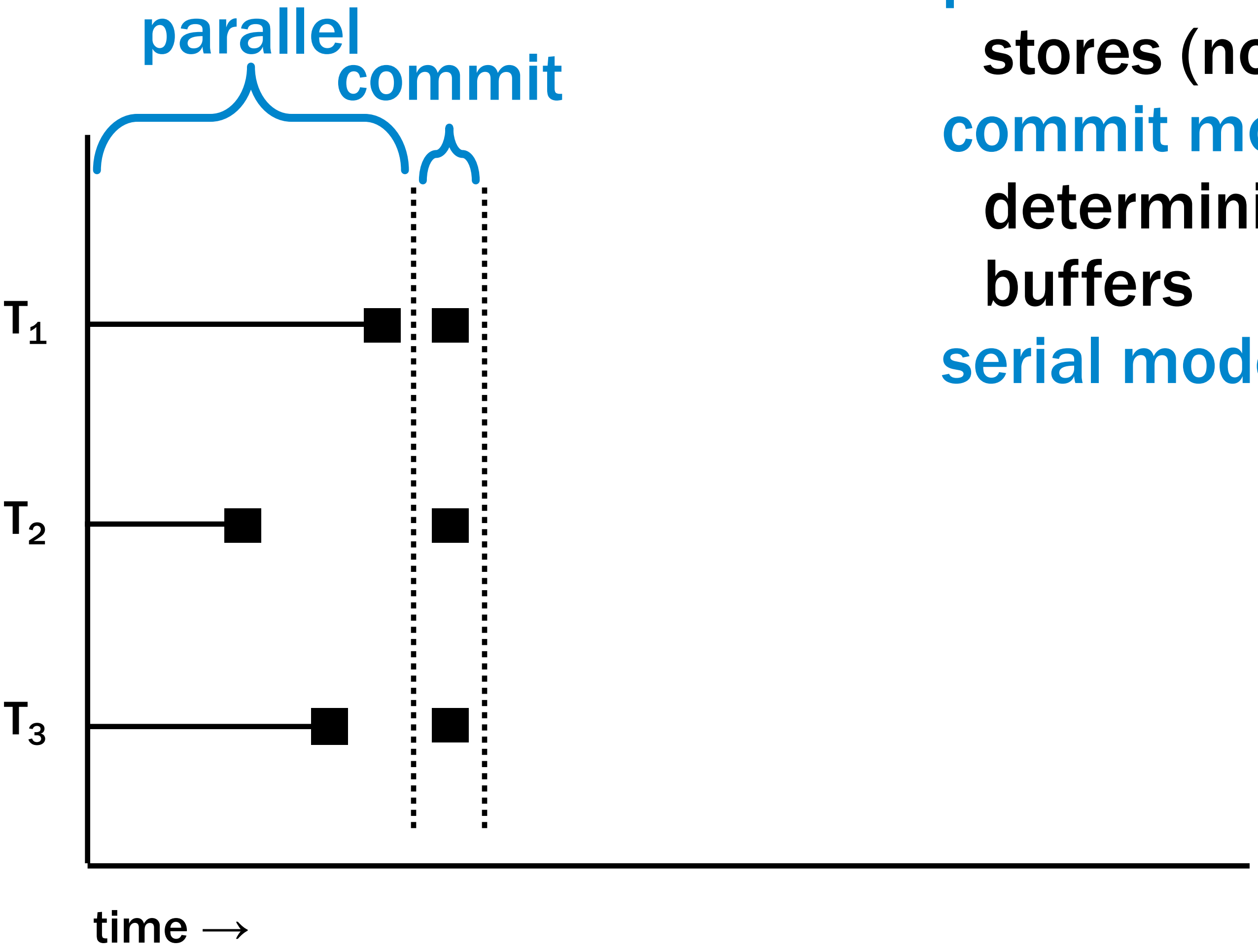


parallel mode: buffer all stores (no communication)

commit mode: deterministically publish buffers

DMP-Buffering: Trading consistency for performance

heard that before? :)



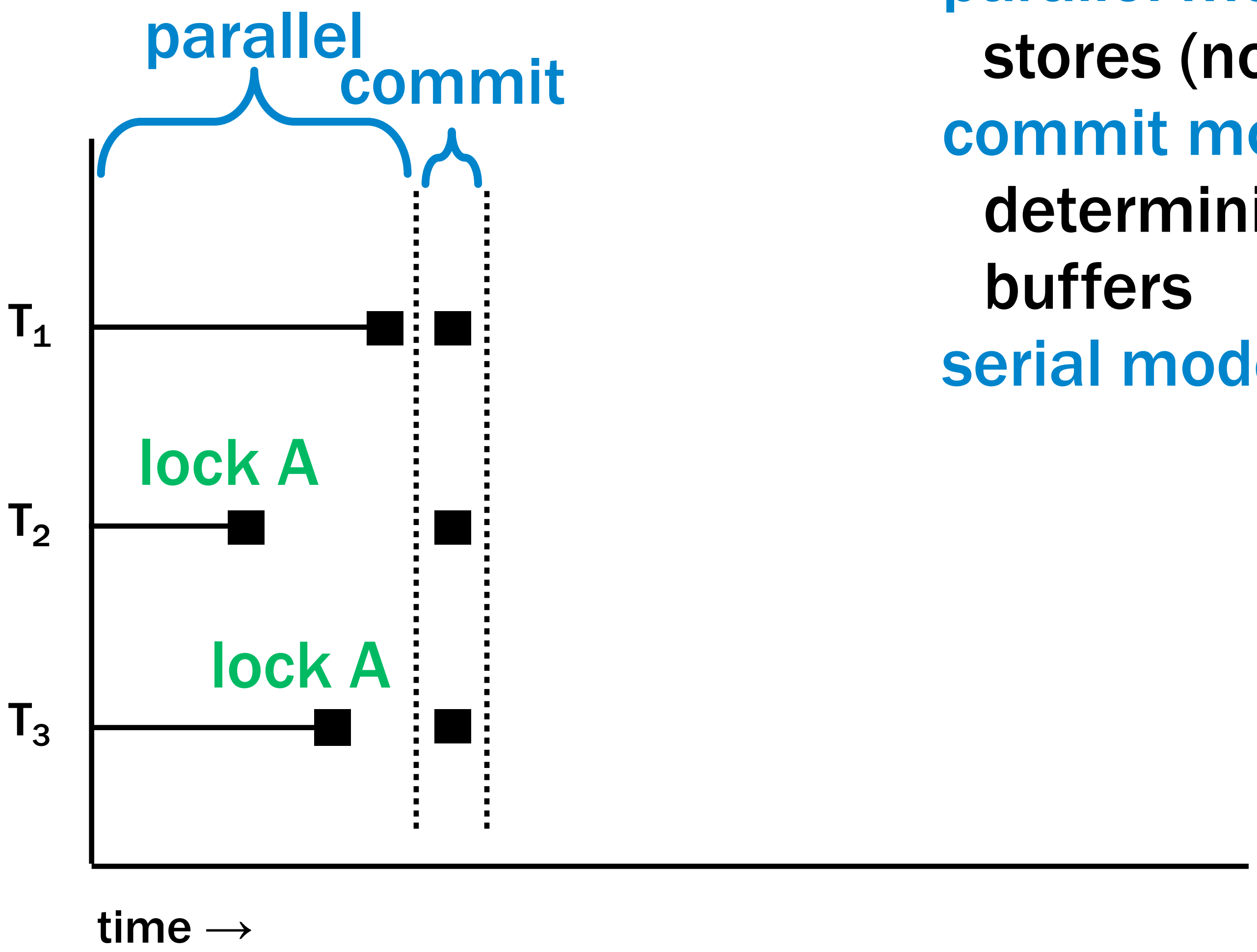
parallel mode: buffer all stores (no communication)

commit mode: deterministically publish buffers

serial mode: for atomic ops

DMP-Buffering: Trading consistency for performance

heard that before? :)



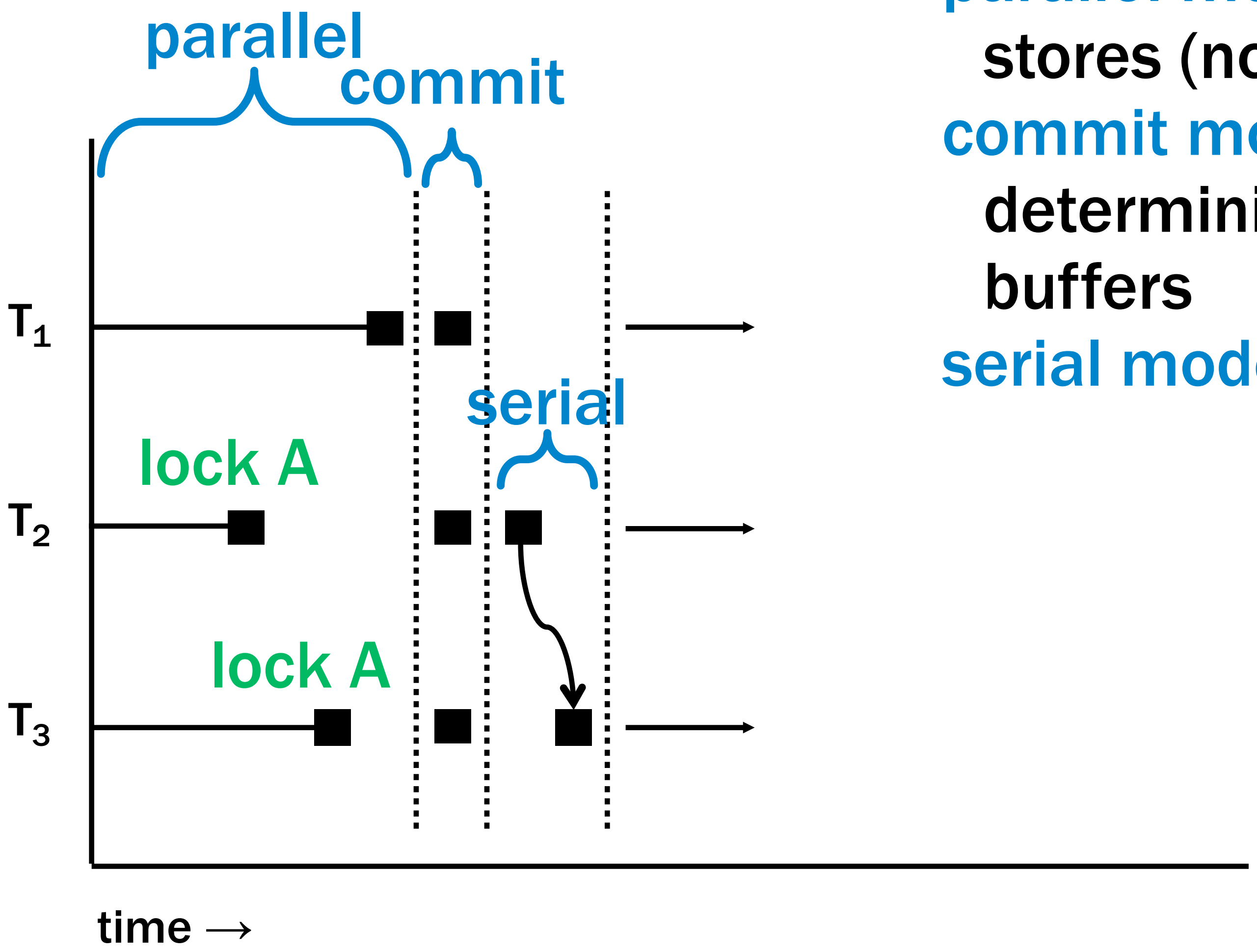
parallel mode: buffer all stores (no communication)

commit mode: deterministically publish buffers

serial mode: for atomic ops

DMP-Buffering: Trading consistency for performance

heard that before? :)

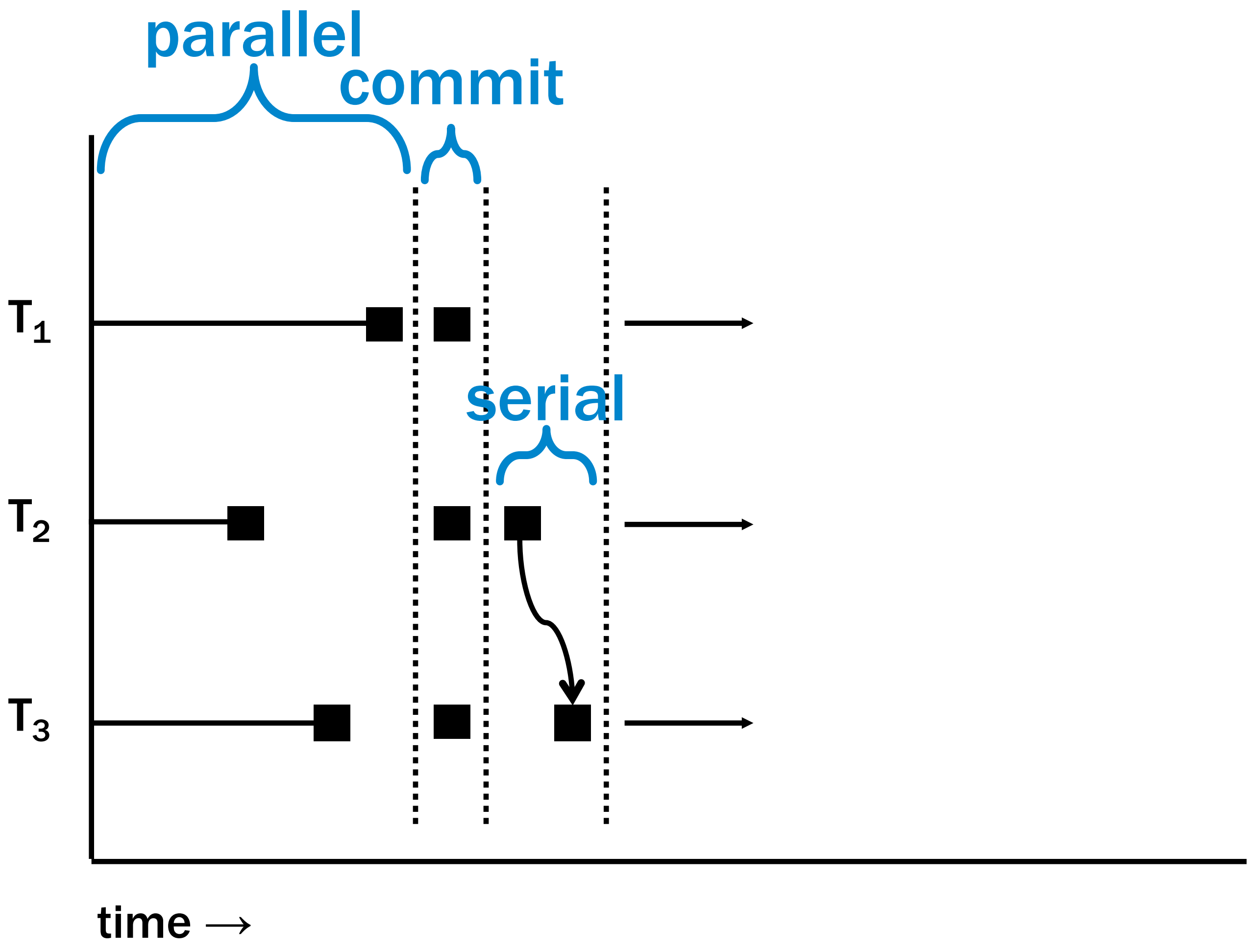


parallel mode: buffer all stores (no communication)

commit mode: deterministically publish buffers

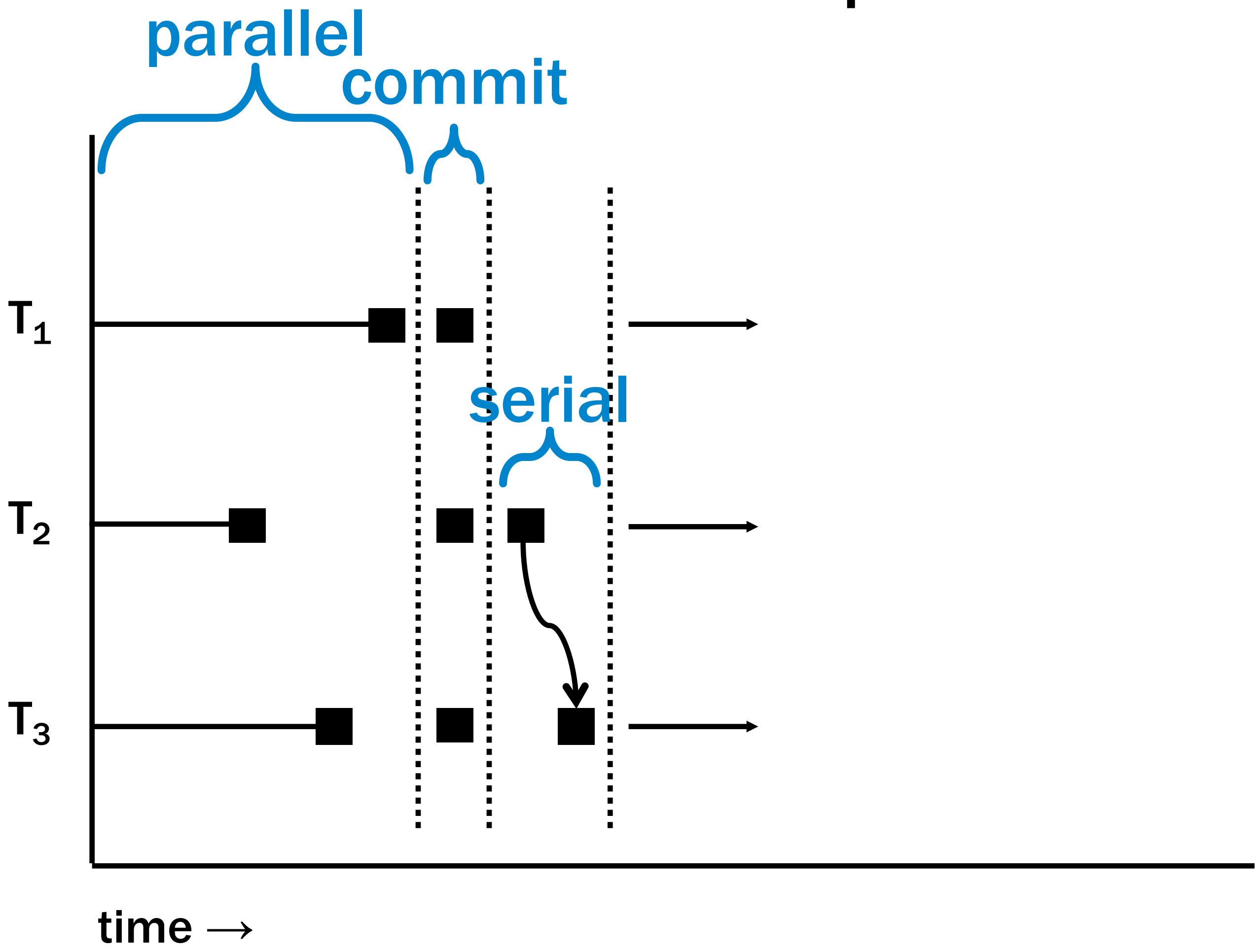
serial mode: for atomic ops

DMP-B “Correctness”

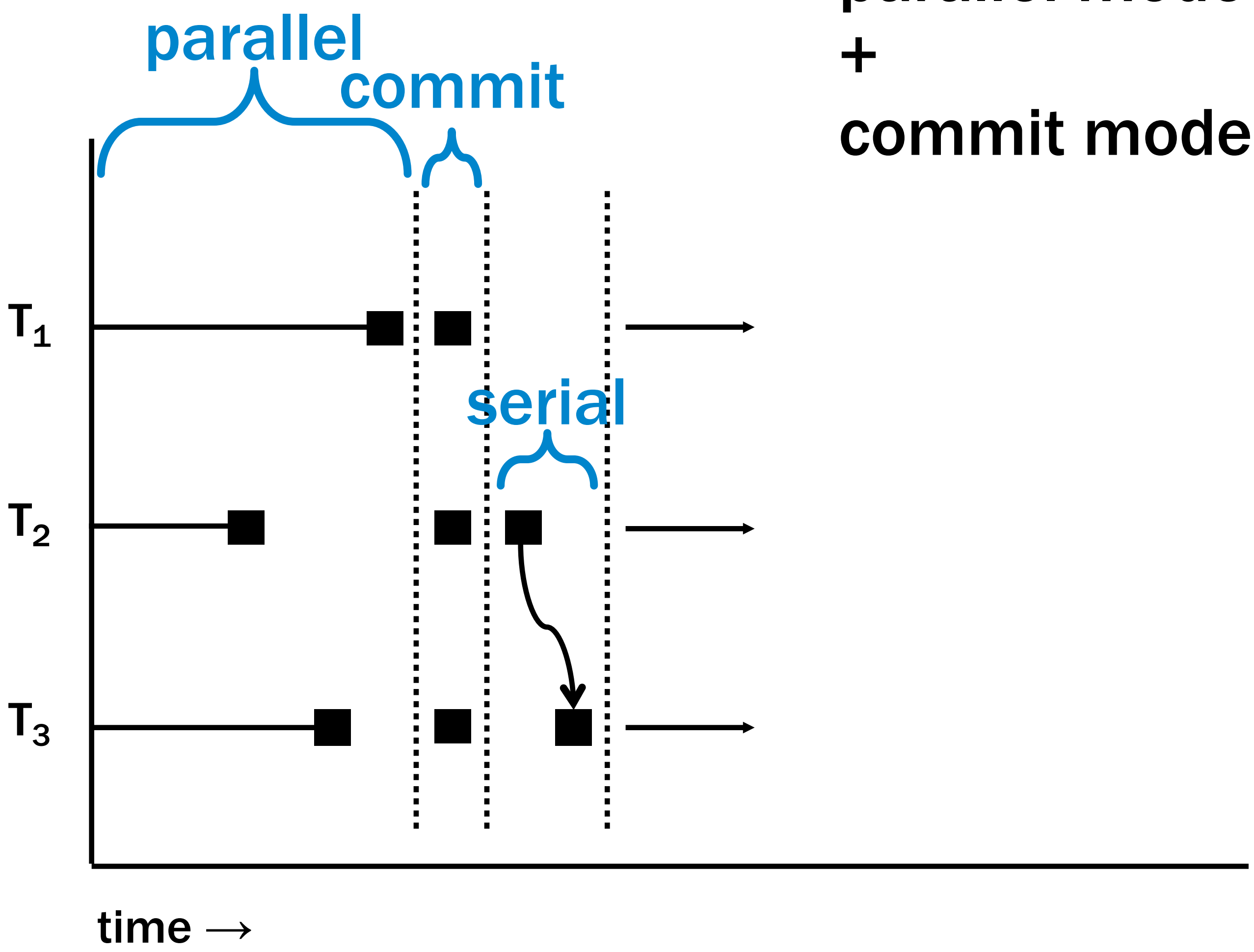


DMP-B “Correctness”

parallel mode (isolated threads)

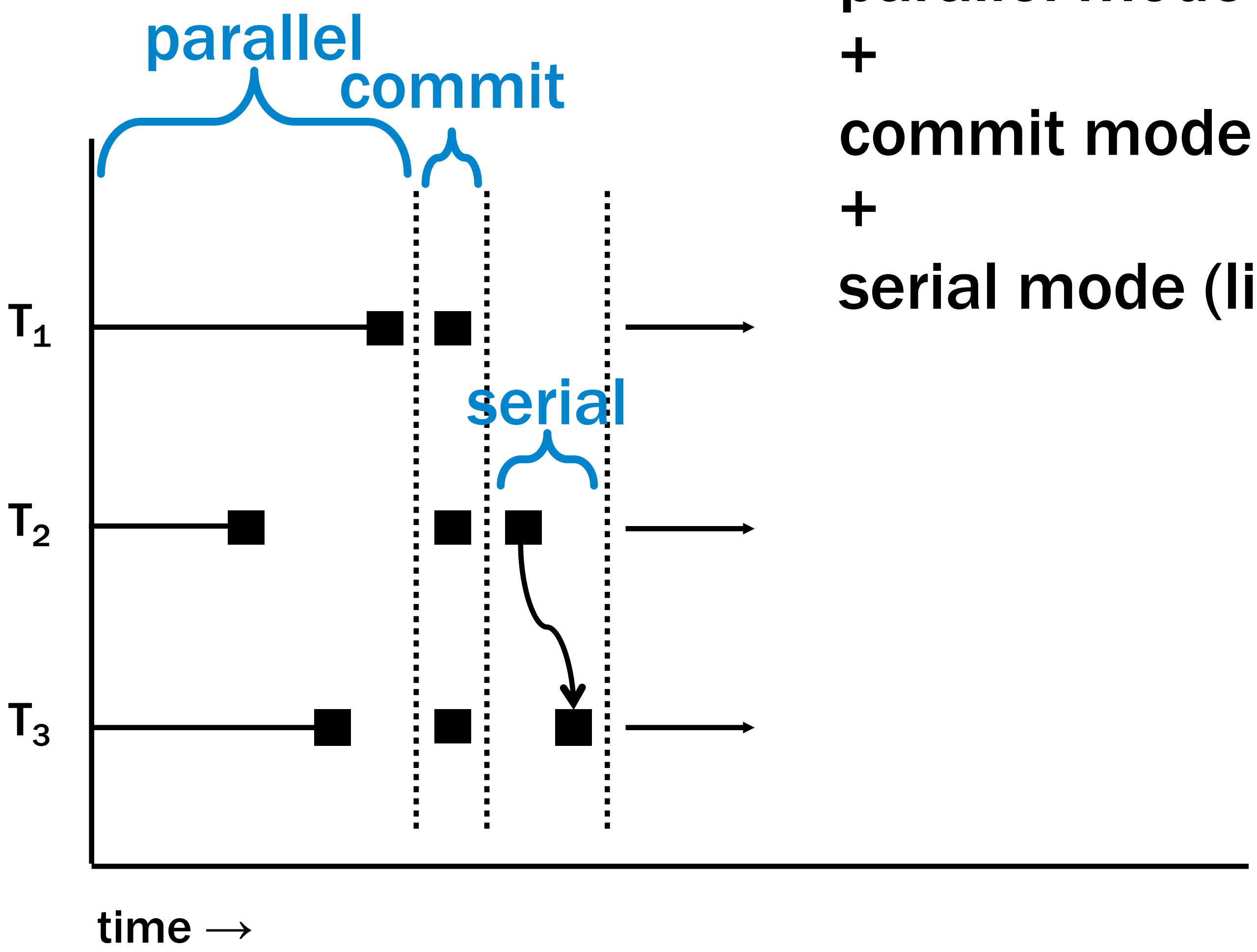


DMP-B “Correctness”



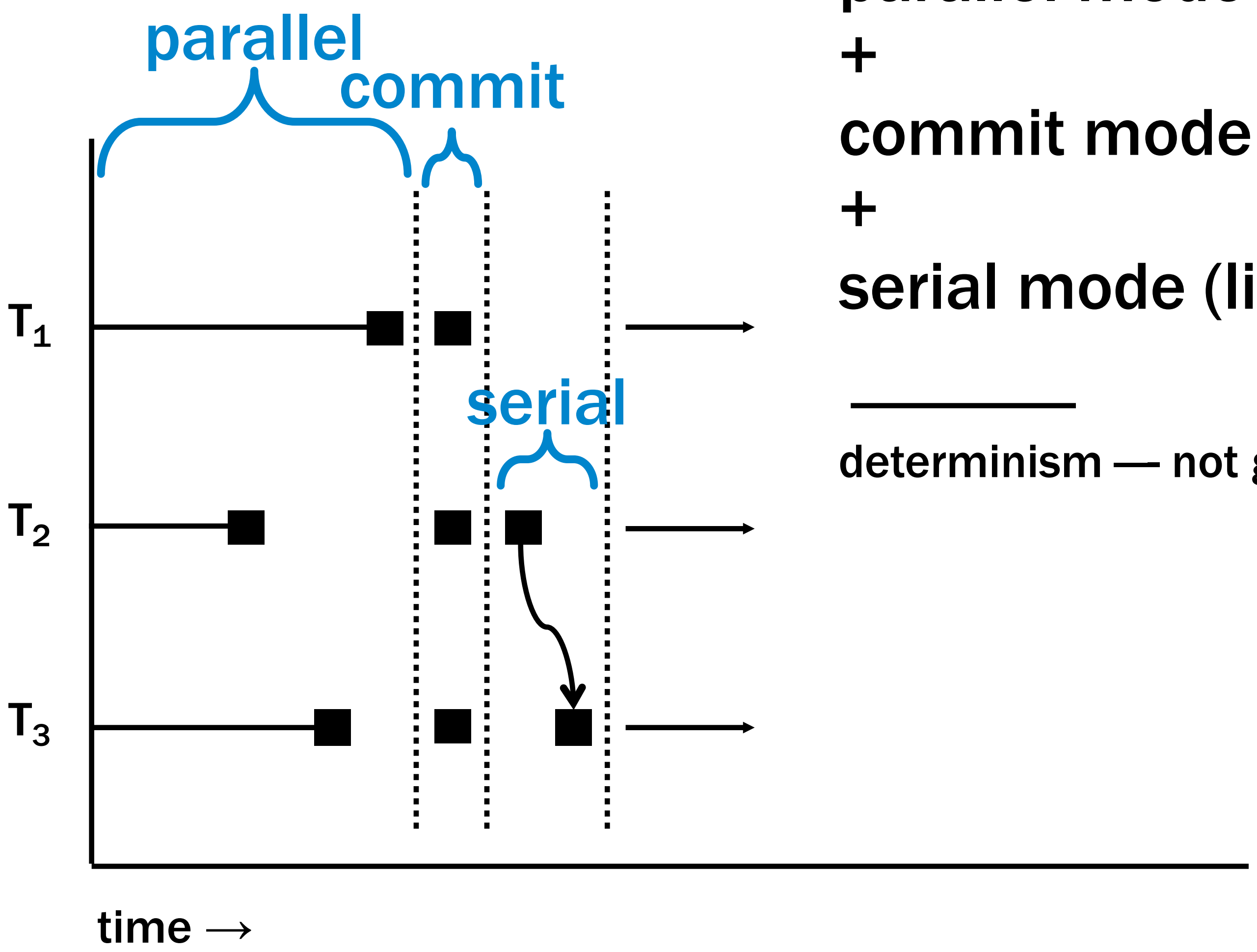
parallel mode (isolated threads)
+
commit mode (logically serial)

DMP-B “Correctness”



parallel mode (isolated threads)
+
commit mode (logically serial)
+
serial mode (like DMP-Serial)

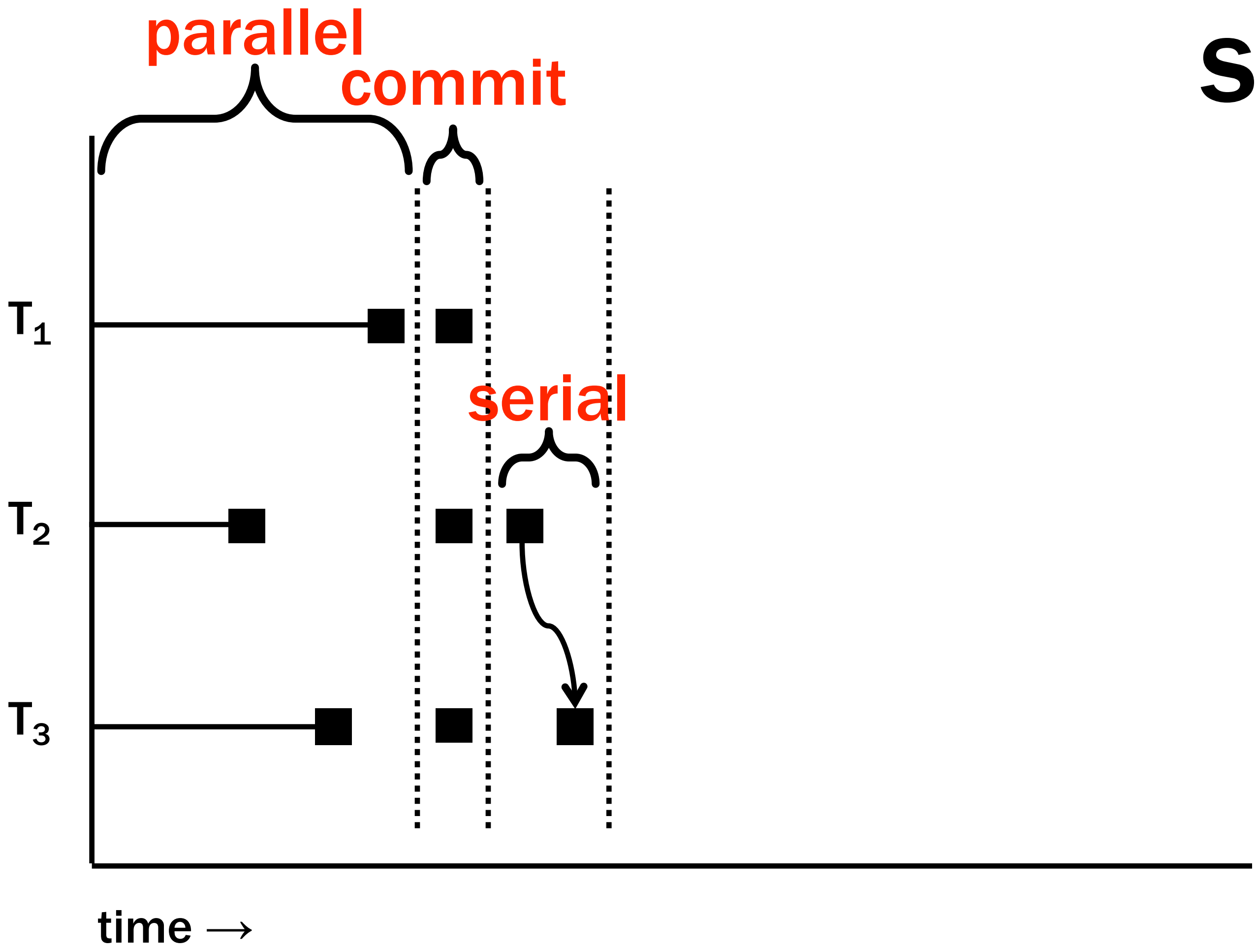
DMP-B “Correctness”



parallel mode (isolated threads)
+
commit mode (logically serial)
+
serial mode (like DMP-Serial)

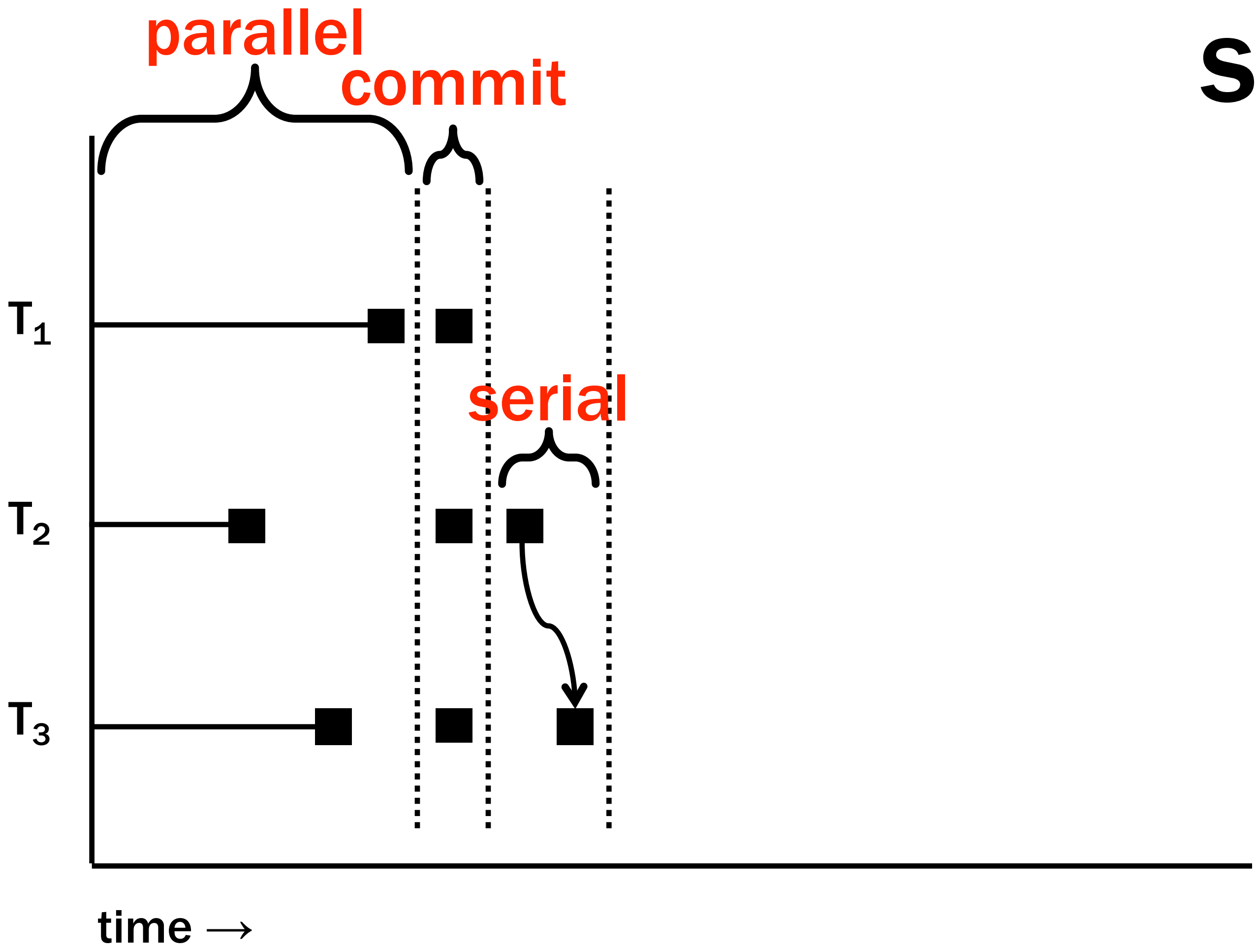
determinism — not guaranteed to be SC

DMP-B Overheads



serial mode

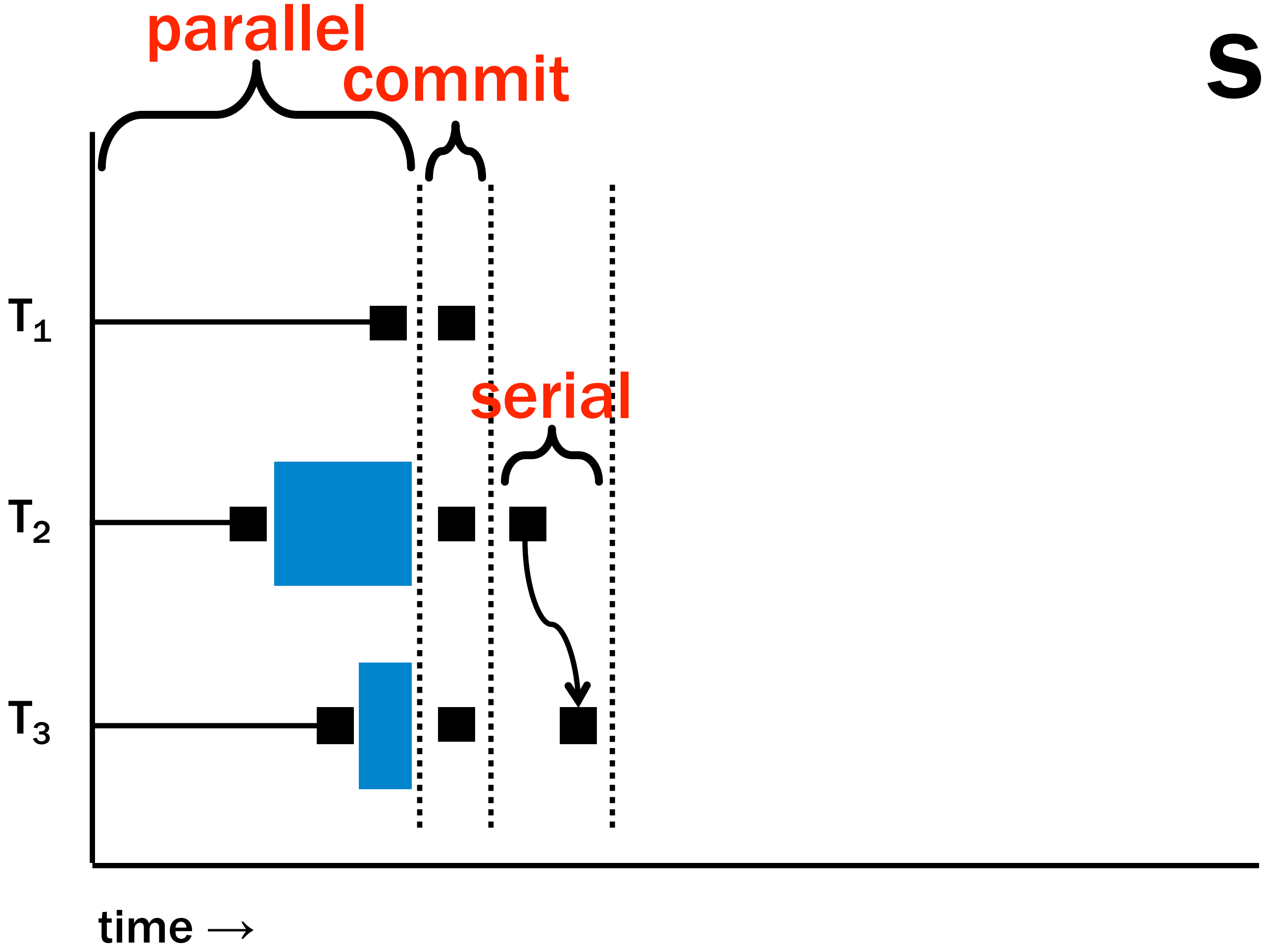
DMP-B Overheads



serial mode

imbalance

DMP-B Overheads



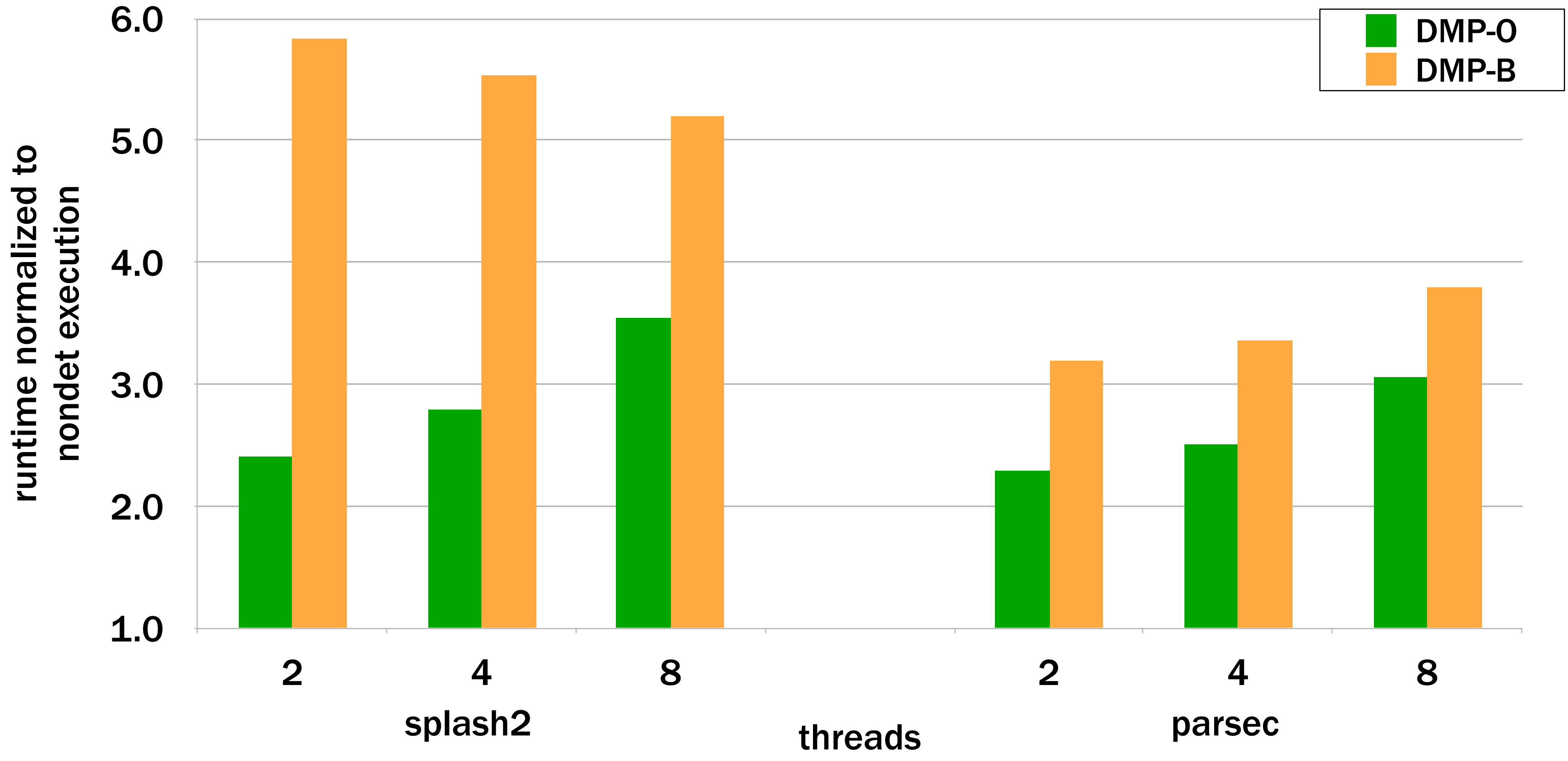
serial mode

imbalance

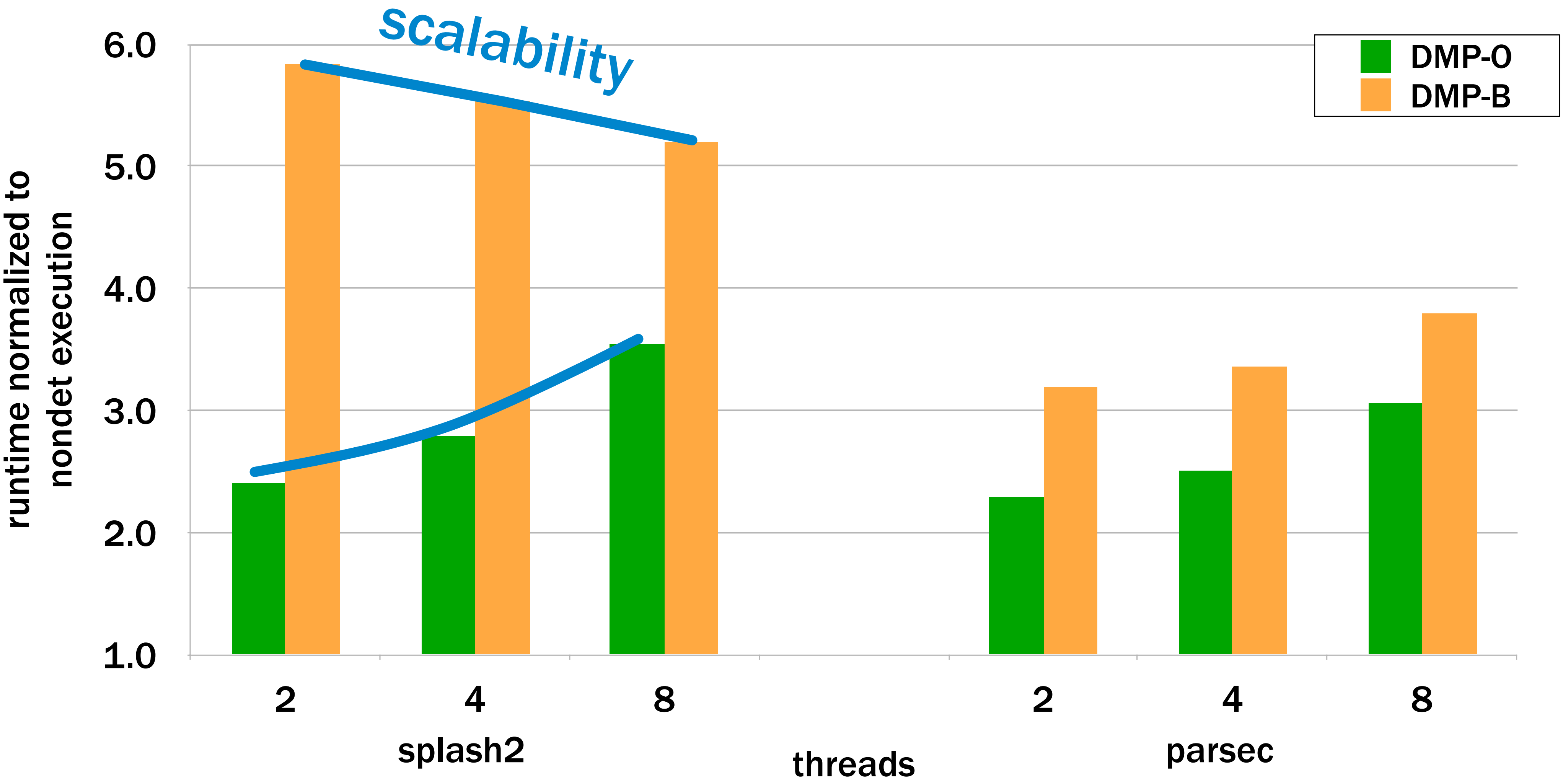
DMP-B Evaluation (1/2)

- C/C++ compiler pass for LLVM
  - yes, the previous results were for a HW-implementation... sorry
- Runtime library that replaces pthreads library, schedules threads and tracks inter-thread communication
- Intel 8-core 2.4GHz Xeon with 10GB RAM, 64-bit Ubuntu 8.10
- SPLASH2 and PARSEC

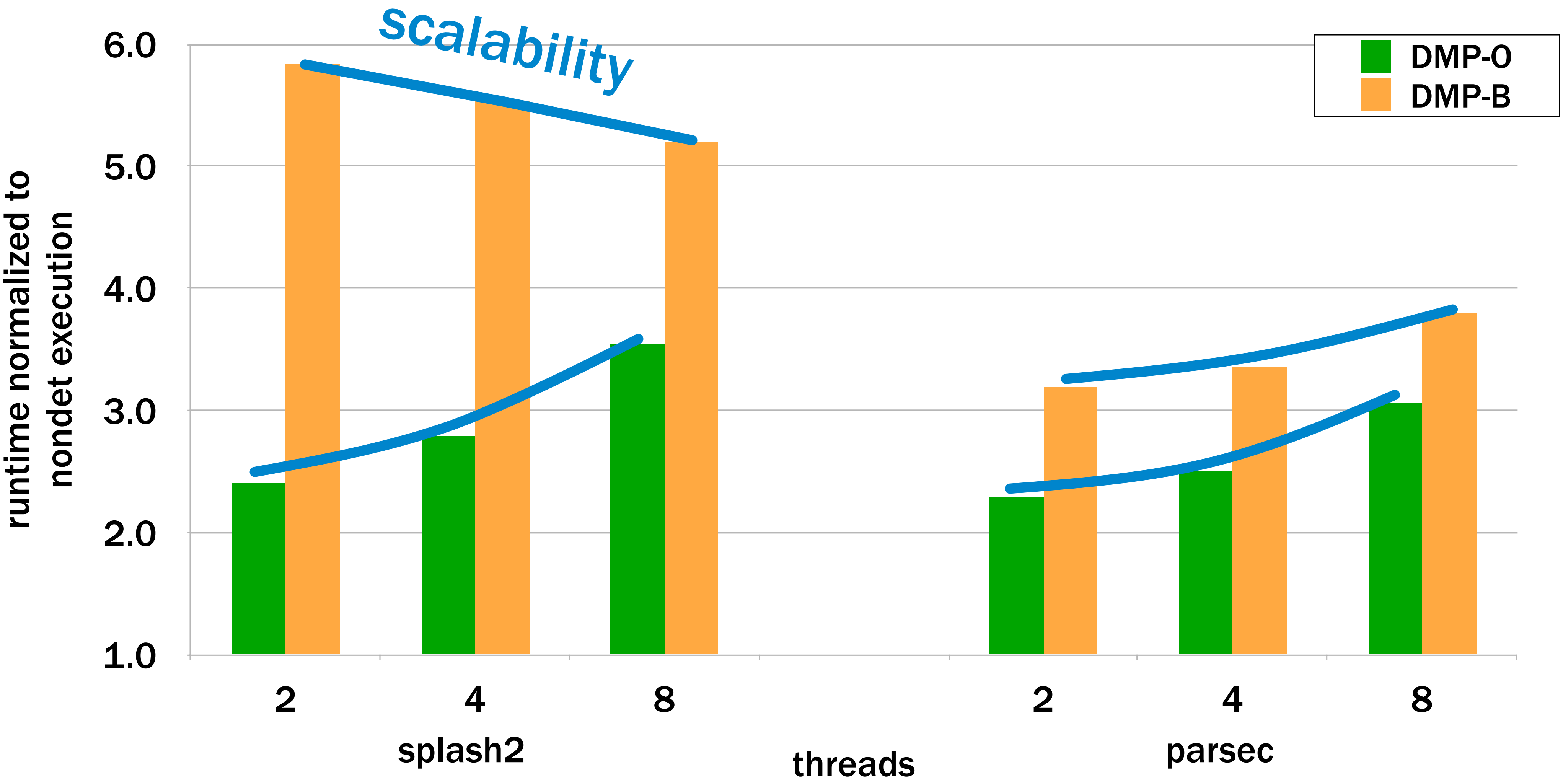
DMP-B Evaluation (2/2)



DMP-B Evaluation (2/2)



DMP-B Evaluation (2/2)



Improving Balance: Better Quantum Building

Improving Balance: Better Quantum Building

- Any deterministic policy will work

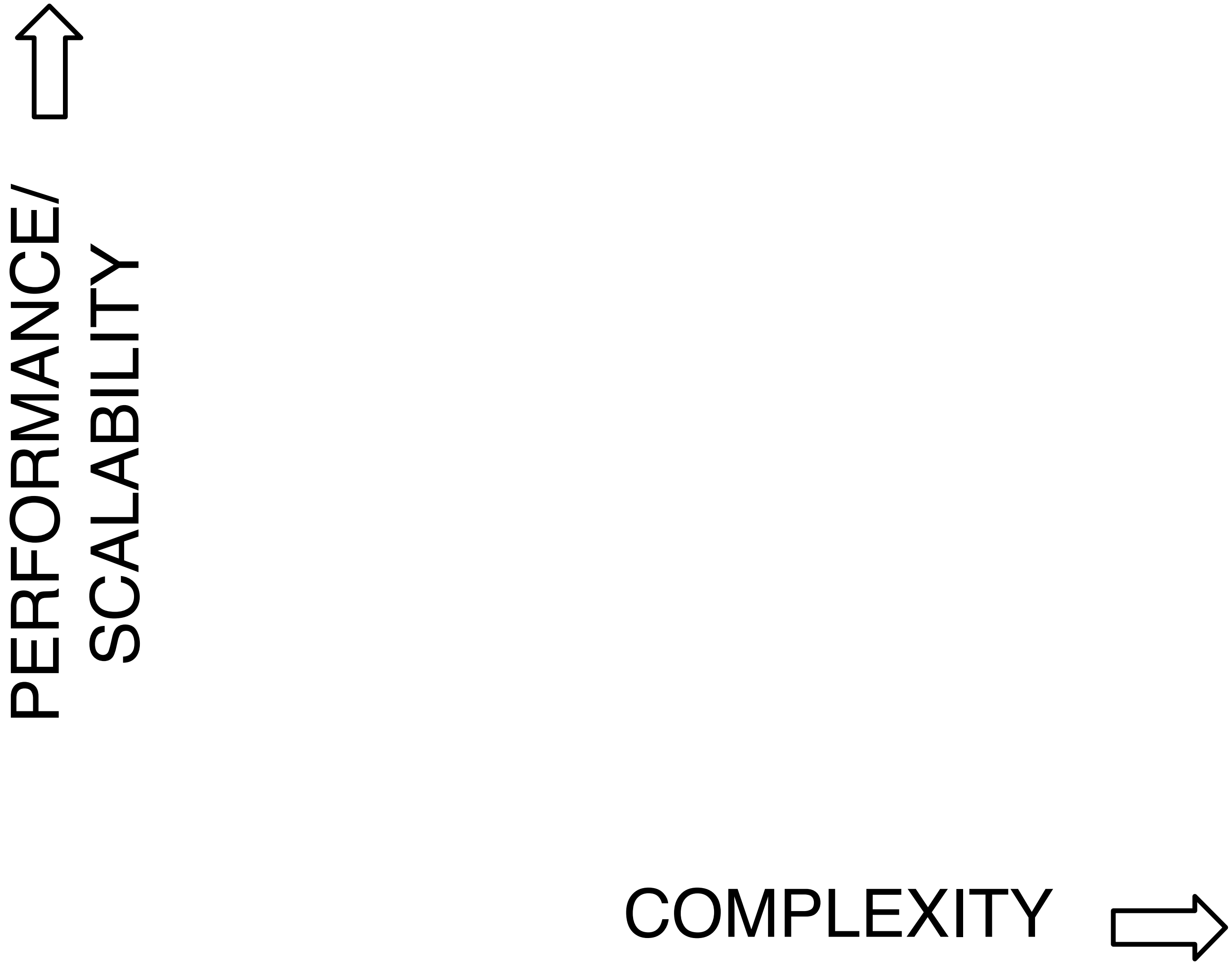
Improving Balance: Better Quantum Building

- Any deterministic policy will work
- We want quanta that are free of communication
  - no communication → no serialization, no rollbacks

Improving Balance: Better Quantum Building

- Any deterministic policy will work
- We want quanta that are free of communication
  - no communication → no serialization, no rollbacks
- Leverage
  - synchronization: end quantum at release points
  - sharing: end quantum after bursty shared accesses
  - program structure (backedges, syscalls, etc...)

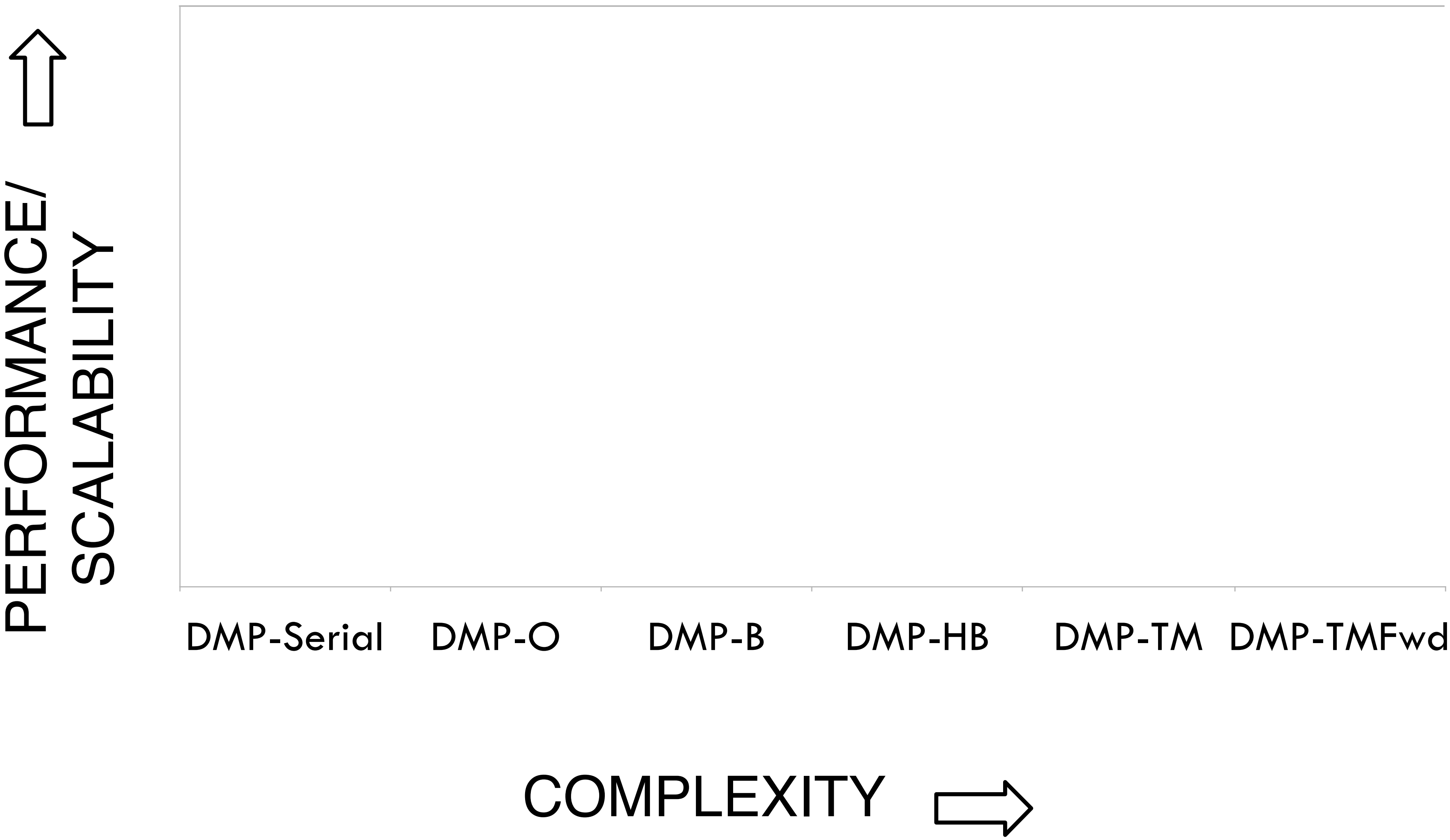
DMP-* Tradeoffs



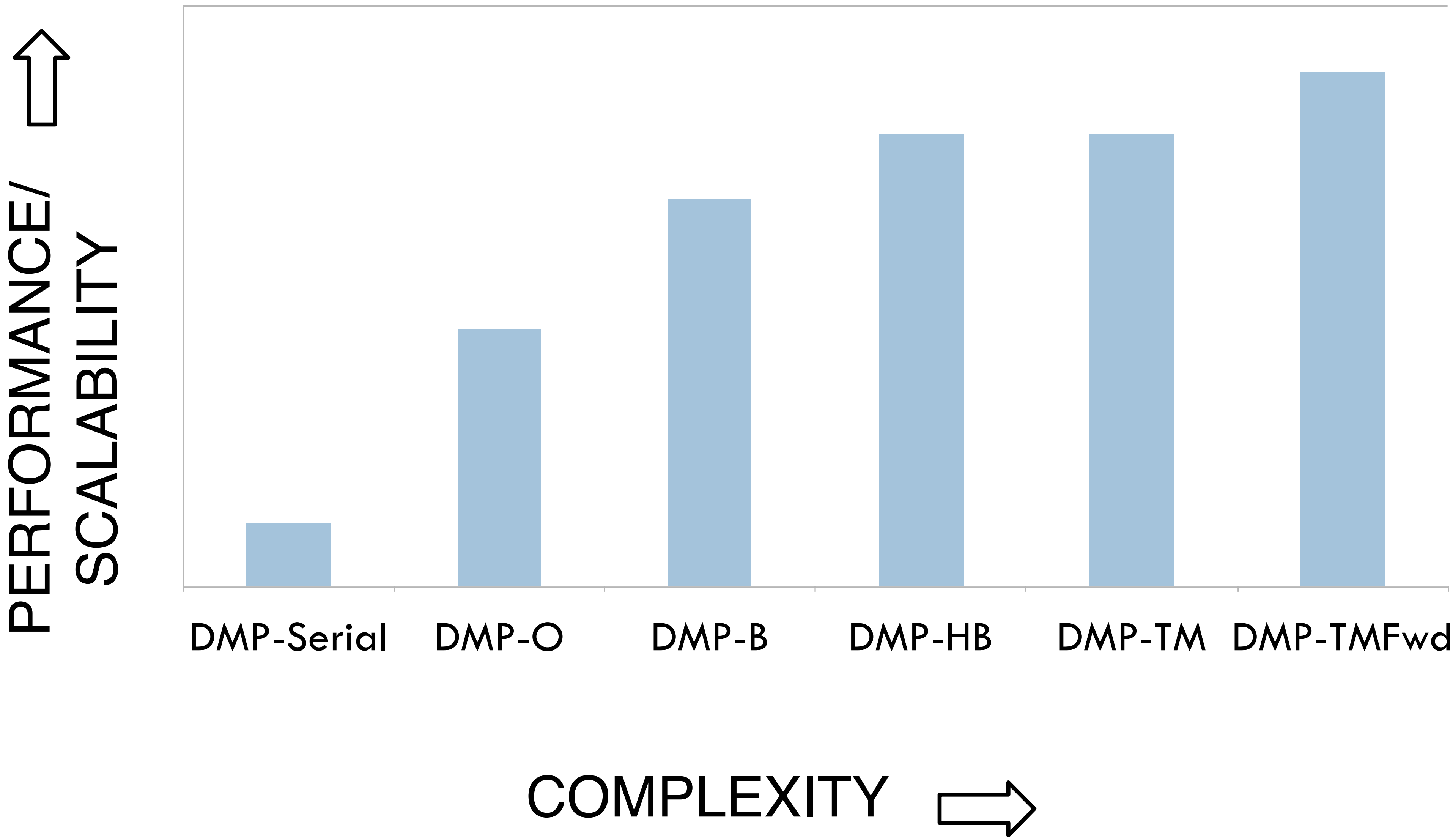
PERFORMANCE/ SCALABILITY

COMPLEXITY

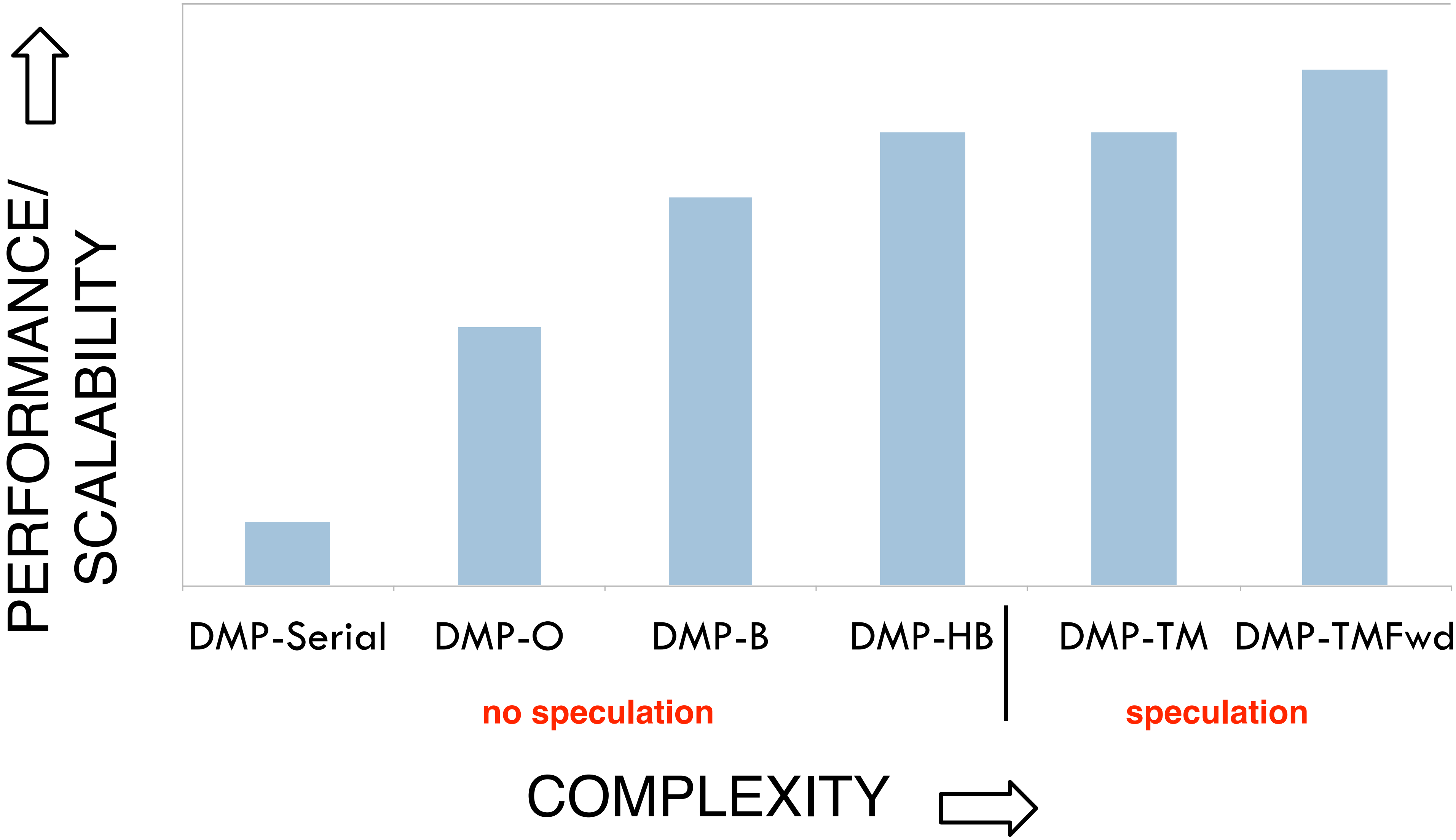
DMP-* Tradeoffs



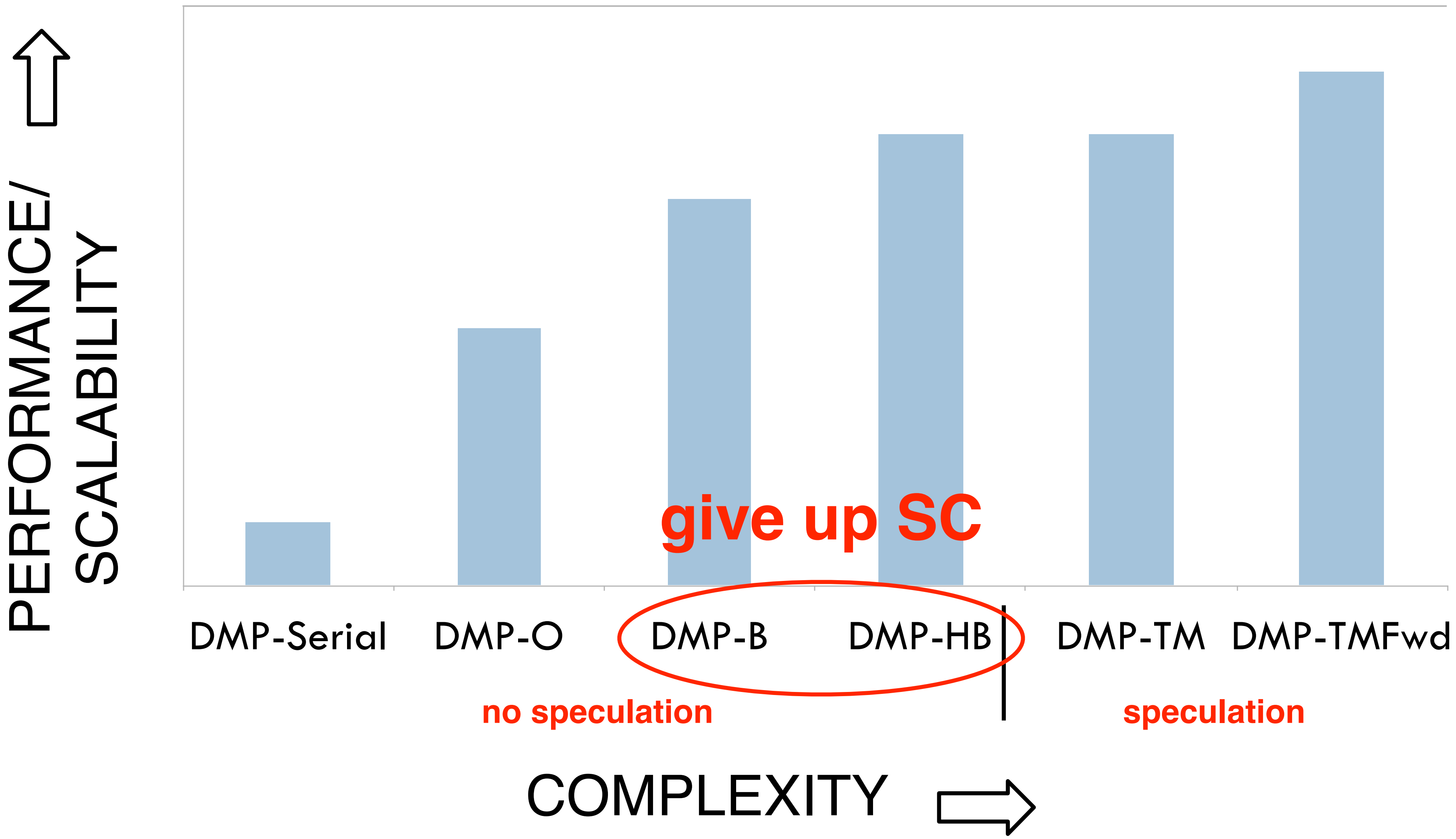
DMP-* Tradeoffs



DMP-* Tradeoffs



DMP-* Tradeoffs



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

Determinism and Fail-stop Races for Sane Multiprocessing

Luis Ceze, *University of Washington*

Safe MultiProcessing Architectures
at the University of Washington

DMP-TSO breaking SC

Thread 1

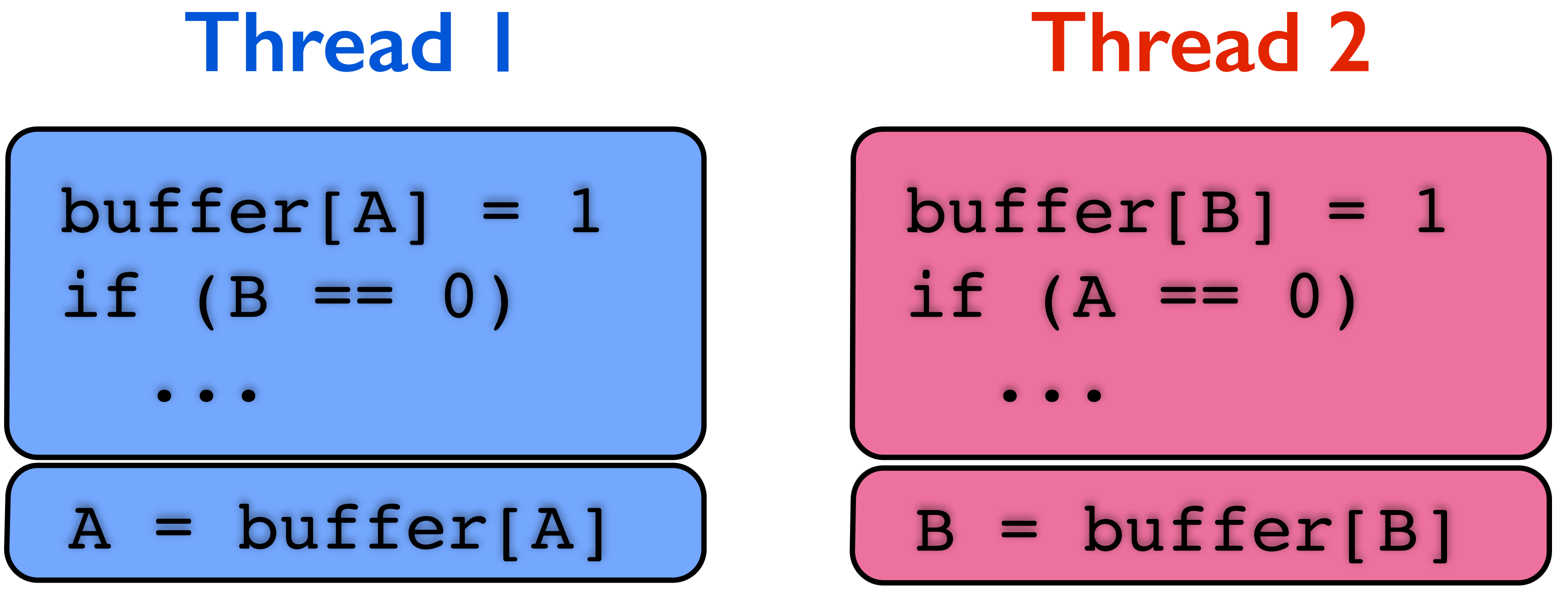
```
A = 1
if (B == 0)
   ...
```

Thread 2

```
B = 1
if (A == 0)
   ...
```

Dekker's Algorithm
(there is a data race)

DMP-TSO breaking SC



Thread 1

```
buffer[A] = 1
if (B == 0)
   ...
```

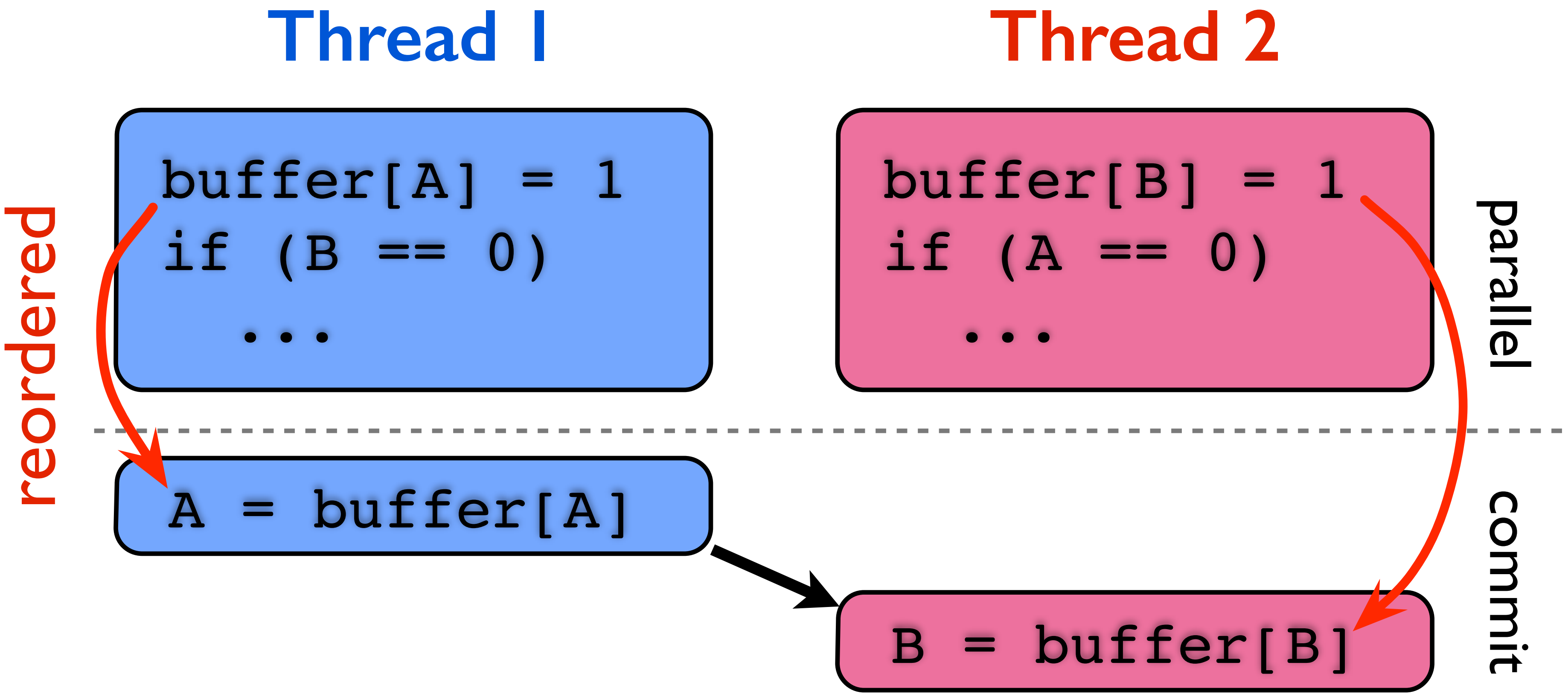
```
A = buffer[A]
```

Thread 2

```
buffer[B] = 1
if (A == 0)
   ...
```

```
B = buffer[B]
```

DMP-TSO breaking SC



Thread 1

```
buffer[A] = 1
if (B == 0)
  ...
```

```
A = buffer[A]
```

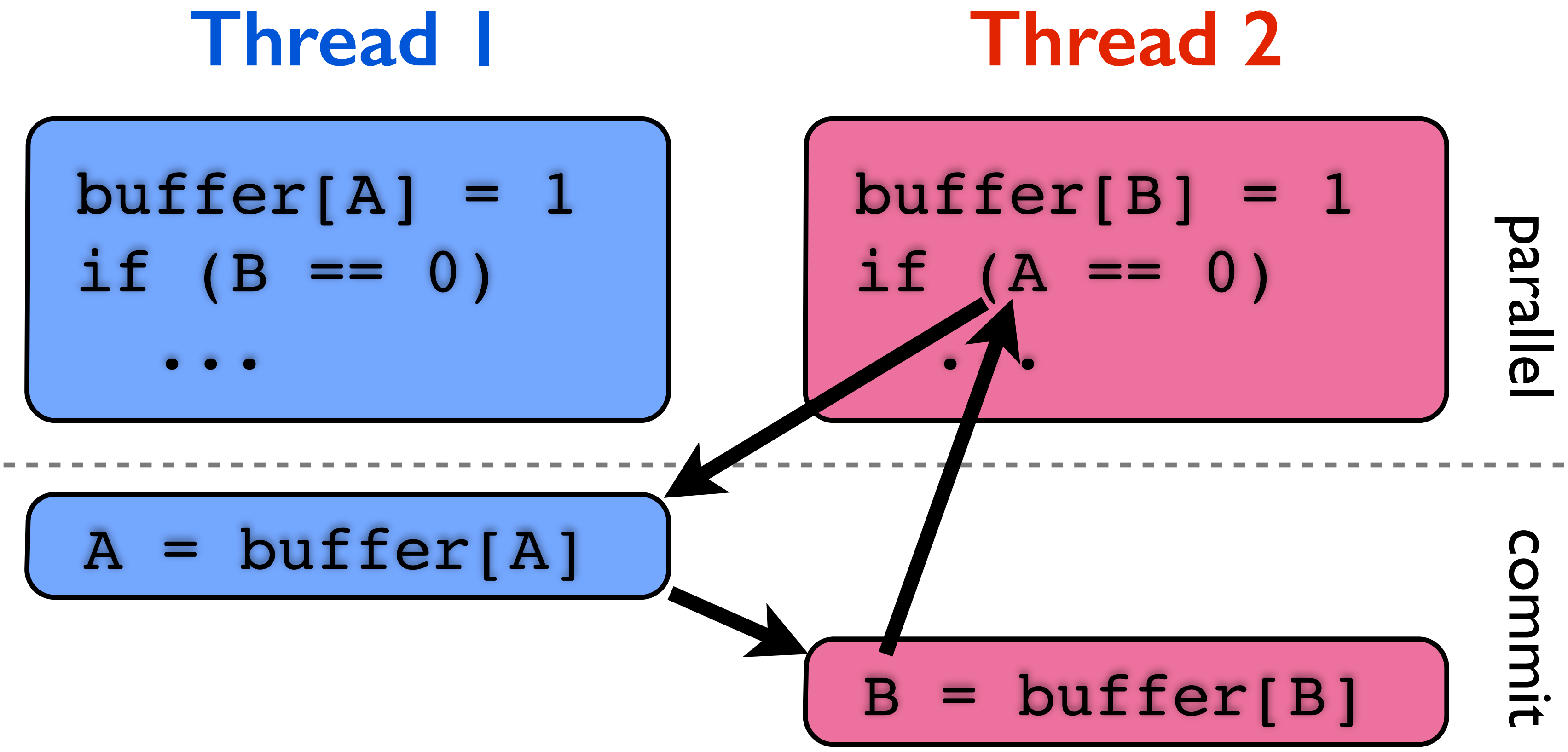
Thread 2

```
buffer[B] = 1
if (A == 0)
  ...
```

```
B = buffer[B]
```

This is deterministic . . .

DMP-TSO breaking SC



Thread 1

```
buffer[A] = 1
if (B == 0)
   ...
```

```
A = buffer[A]
```

Thread 2

```
buffer[B] = 1
if (A == 0)
   ...
```

```
B = buffer[B]
```

parallel

commit

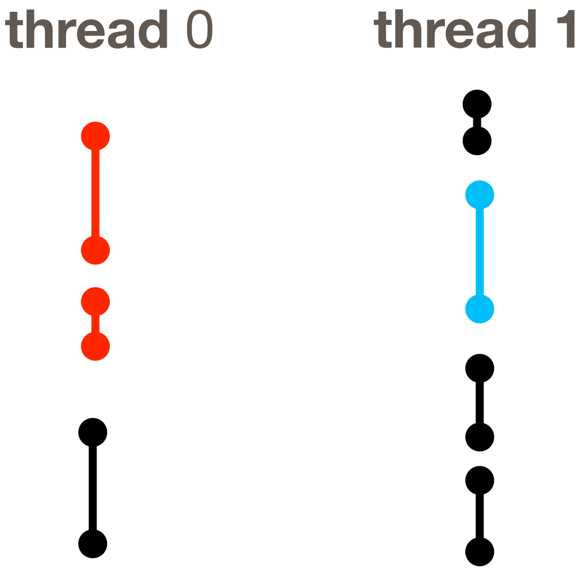
But data race free programs are sequentially consistent (required by C++ and Java memory models)

Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]

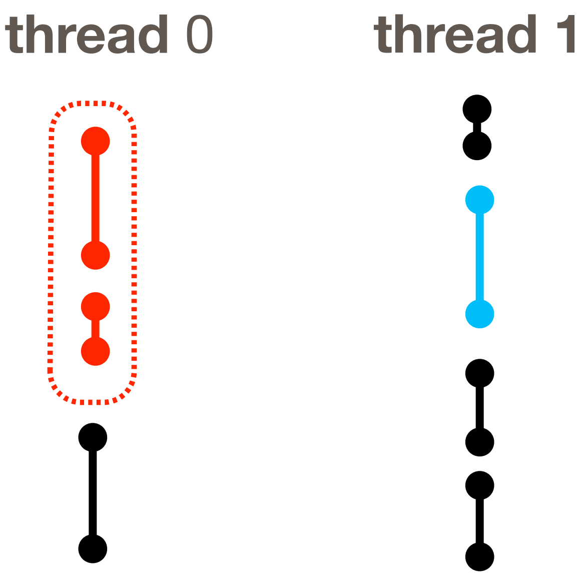
Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]



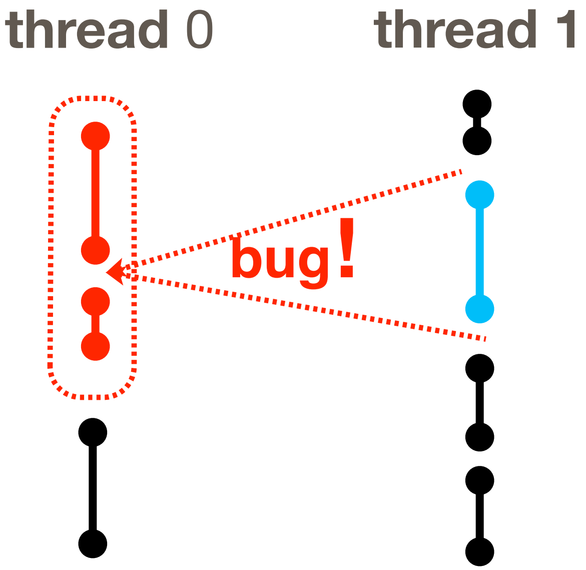
Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]



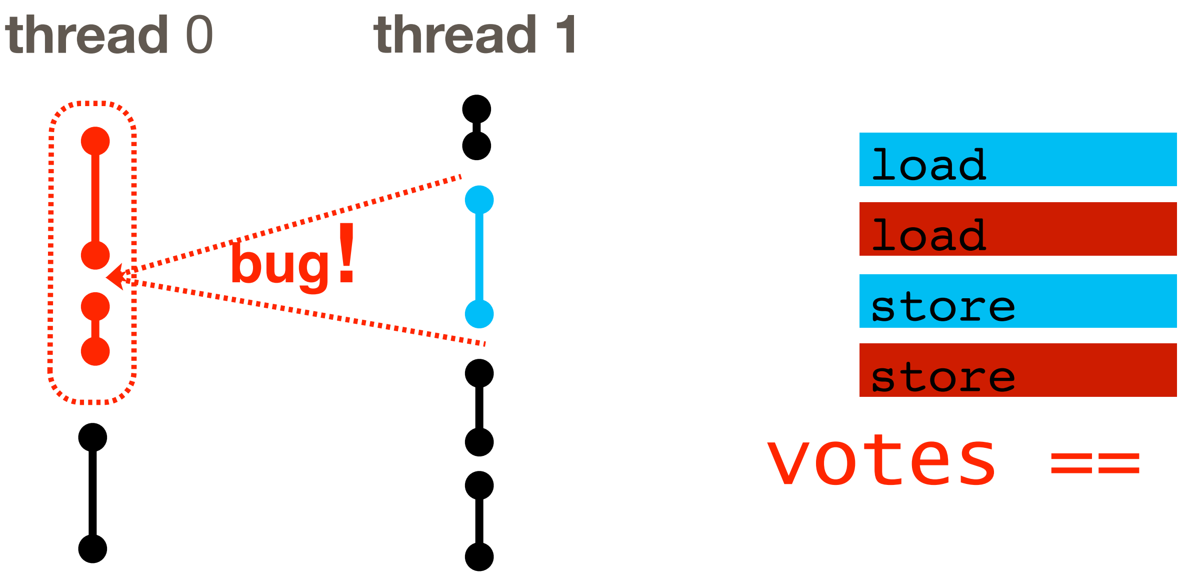
Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]



Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]

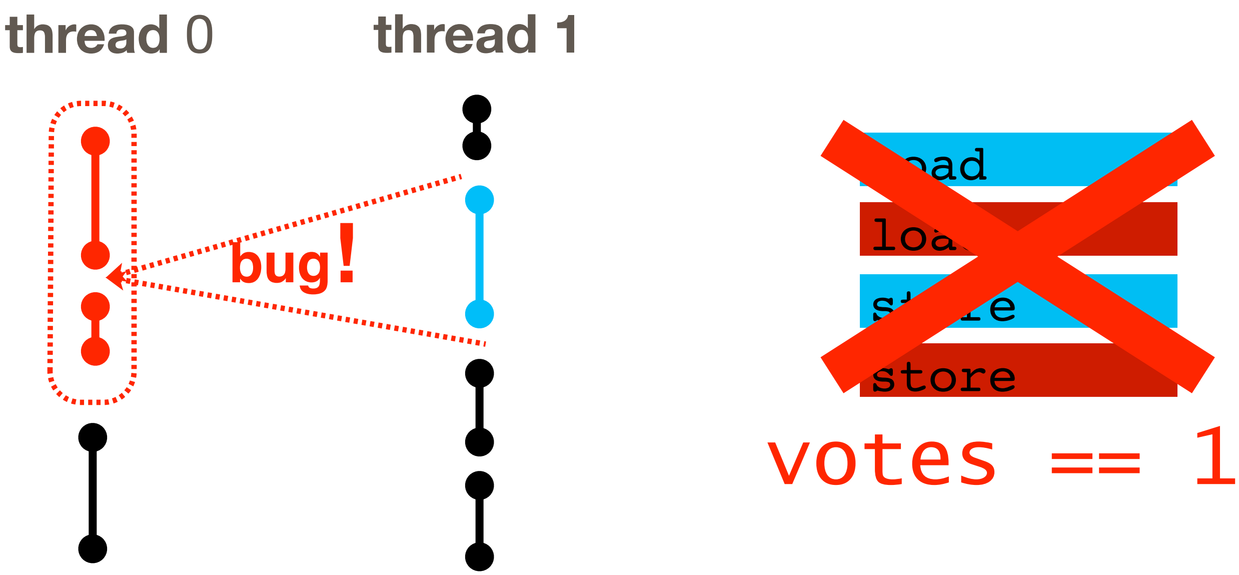


```
load
load
store
store
```

votes == 1

Dynamic Bug Avoidance from 10,000’

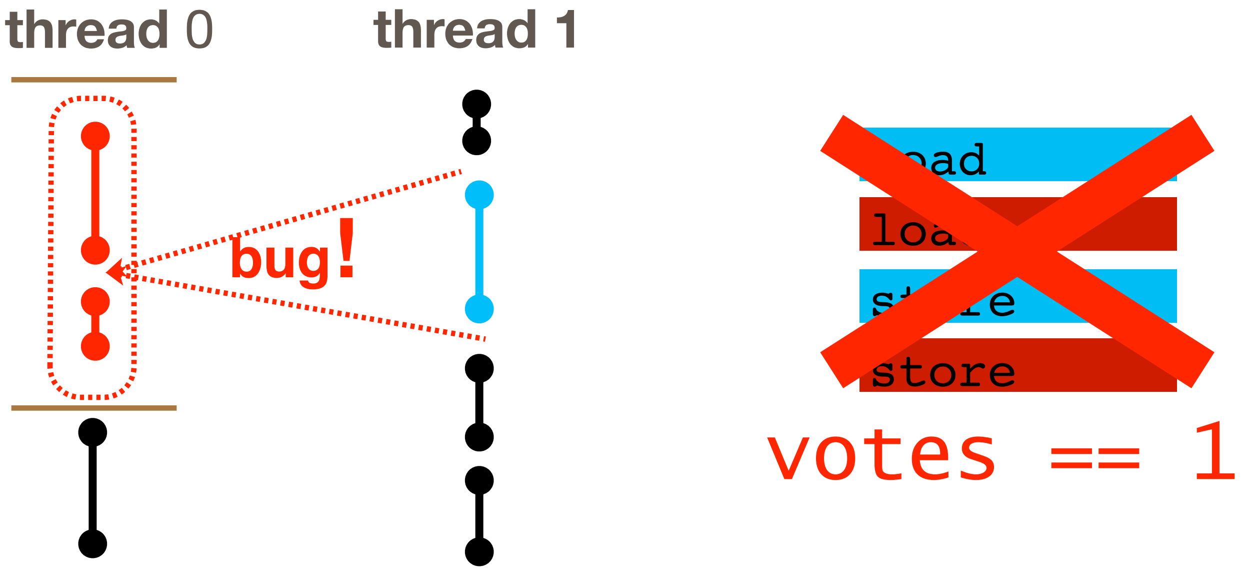
[ISCA’08, ISCA’10]



votes == 1

Dynamic Bug Avoidance from 10,000'

[ISCA'08, ISCA'10]



thread 0

thread 1

bug!

~~load~~

~~load~~

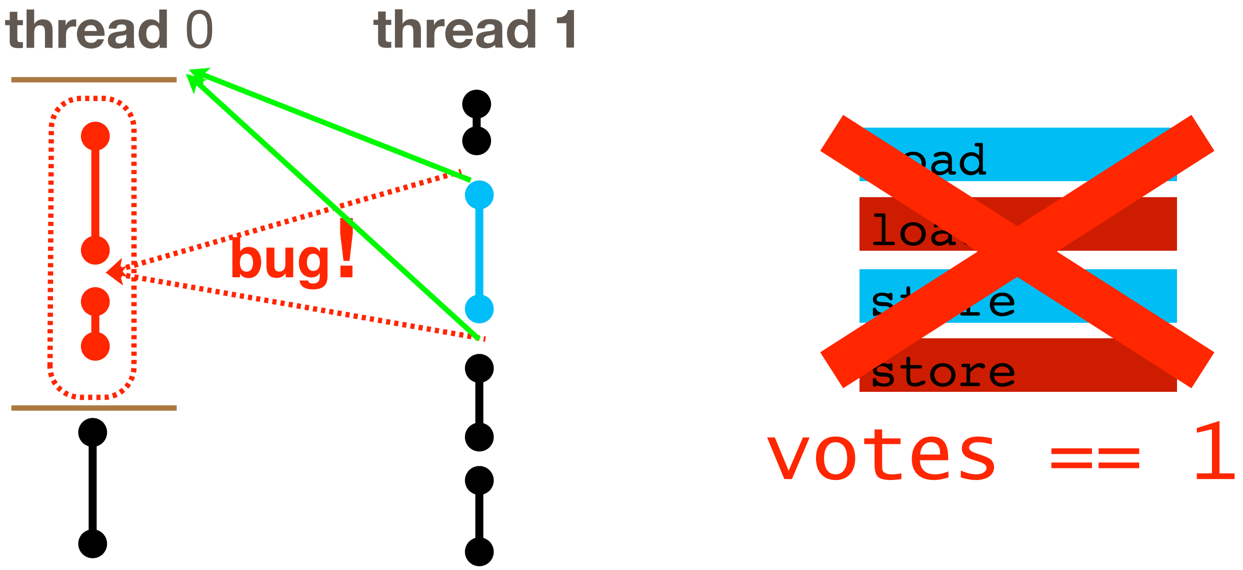
~~store~~

~~store~~

votes == 1

Dynamic Bug Avoidance from 10,000’

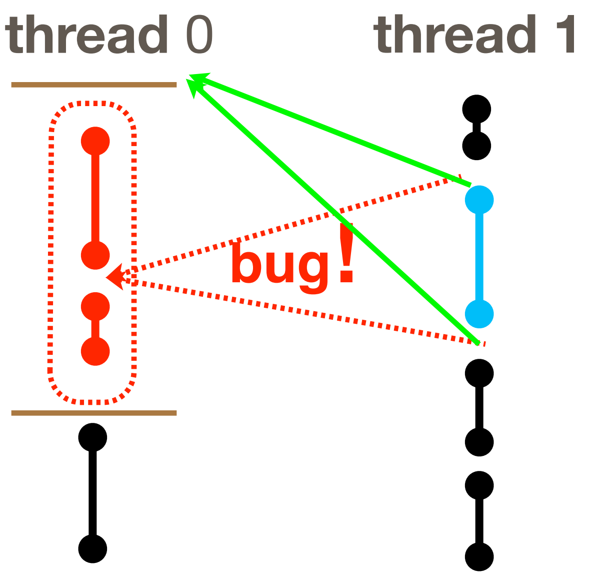
[ISCA’08, ISCA’10]



votes == 1

Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]



~~load~~
~~load~~
~~store~~
~~store~~

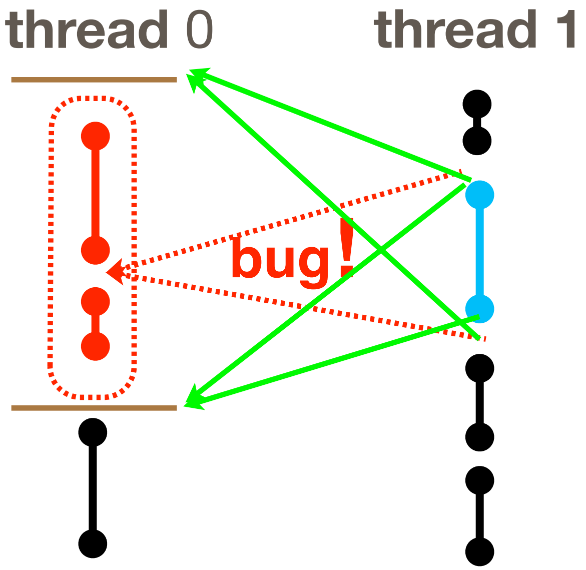
votes == 1

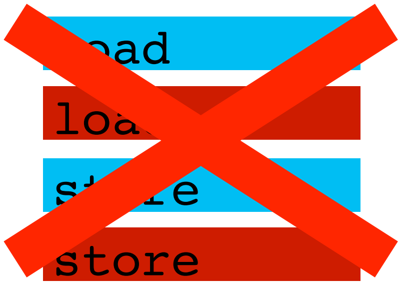
load
store
load
store

votes == 2

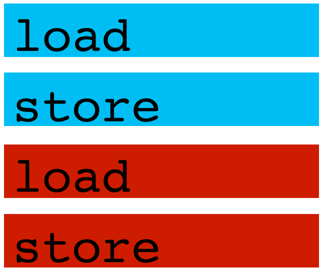
Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]





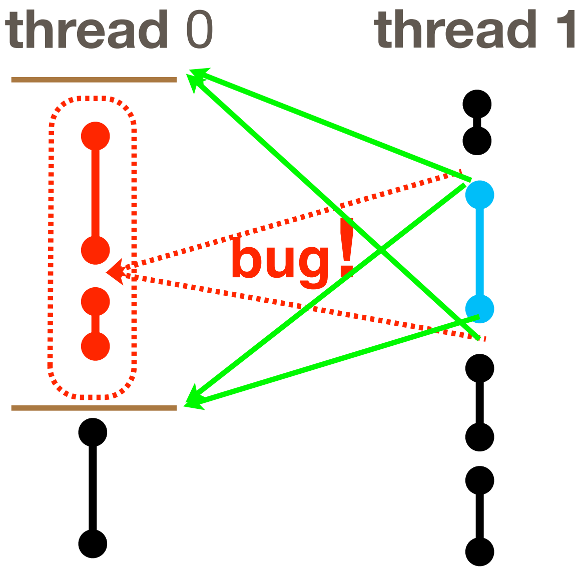
votes == 1



votes == 2

Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]



~~load~~
~~load~~
~~store~~
~~store~~

votes == 1

load
store
load
store

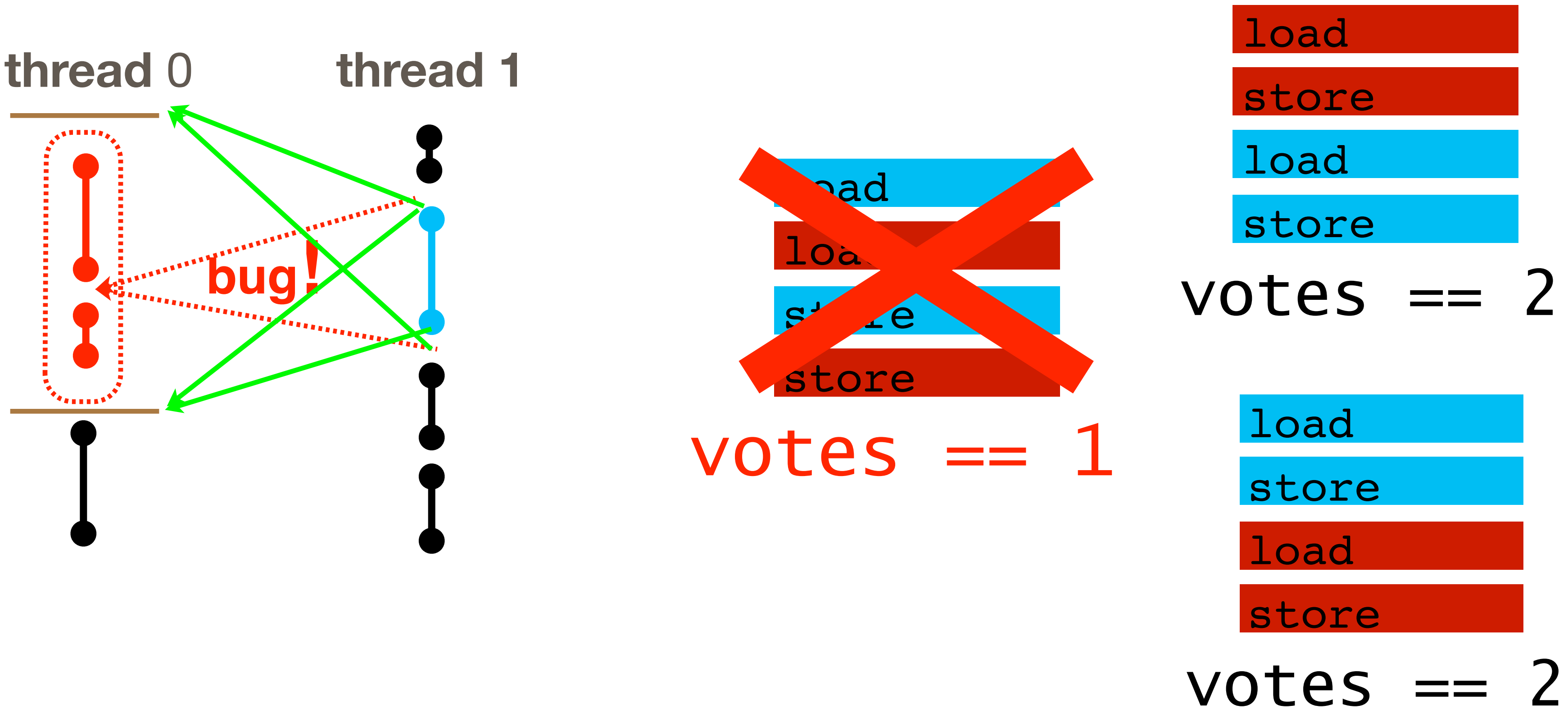
votes == 2

load
store
load
store

votes == 2

Dynamic Bug Avoidance from 10,000’

[ISCA’08, ISCA’10]



- Dynamically detect patterns of buggy interleavings
- Steer the execution away from possibly bad interleavings

What about performance? :)

- 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 apps
  - relaxing memory model helps a lot, even more so than in nondet MPs

- HW implementation: ~5% to 50%
- Compiler implementation: 2x to 3x (instrumentation cost)
- OS (paging): 0% to 10x (false sharing at page gran.)

Ongoing DMP Research

- Support for program instrumentation ***robustness***
  - need to make sure behavior stays the same
- Improving ***scalability***
  - more memory model experiments
- ***Testing***
- Applications to ***distributed systems***

Related work

Approach	Project(s)
Record+Replay	FDR [Xu, ISCA ‘03] ReRun [Hower, ISCA ‘08] Capo [Montesinos, ASPLOS ‘09]
Limited Determinism	Kendo [Olszewski, ASPLOS ‘09] Grace [Berger, OOPSLA ‘09]
Systems Issues	dOS [Bergan, OSDI ‘10] Determinator [Aviram, OSDI ‘10] [Cui, OSDI ‘10]
Deterministic Languages	NESL, JADE CILK, ORCS, DPJ

Corensic DMP Hypervisor.

Or how about Making Errors Failstop?

[ISCA'10]

Fail-Stop Semantics for Data-Races

Semantics are clear and simple

Better data-race debugging

Safety: races can't cause problems

When a data-race occurs, throw an exception

The ***Guarantee***:

Exception-Thrown? There was a data-race.

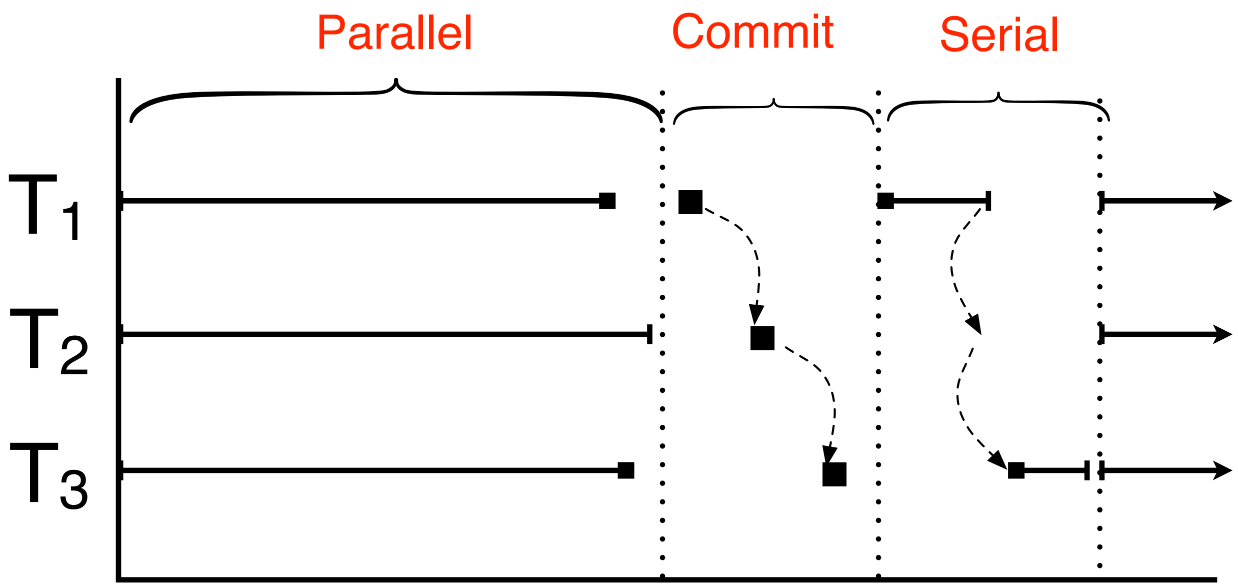
Exception-Free? Sequential Consistency.

CoreDet: Compiler and Runtime System

[ASPLOS'10]

- An implementation of DMP in software
  - DMP-Ownership: simple, reasonable overheads, but poor scalability
- Our goal with this implementation: preserve scalability
- New DMP technique: DMP-Buffering
  - better scalability, but more overheads
  - no speculation (easier to implement than DMP-TM)
  - **key insight: relaxed memory consistency** (specifically, TSO)
    - yes, deterministic relaxed consistency :)

DMP-Buffering



Parallel mode: buffer stores locally

- ends at *synchronization* (*atomic ops* and *fences*), and *quantum boundaries*

Commit mode: publish local store buffers

- happens semantically in serial for determinism
- executes in parallel for performance

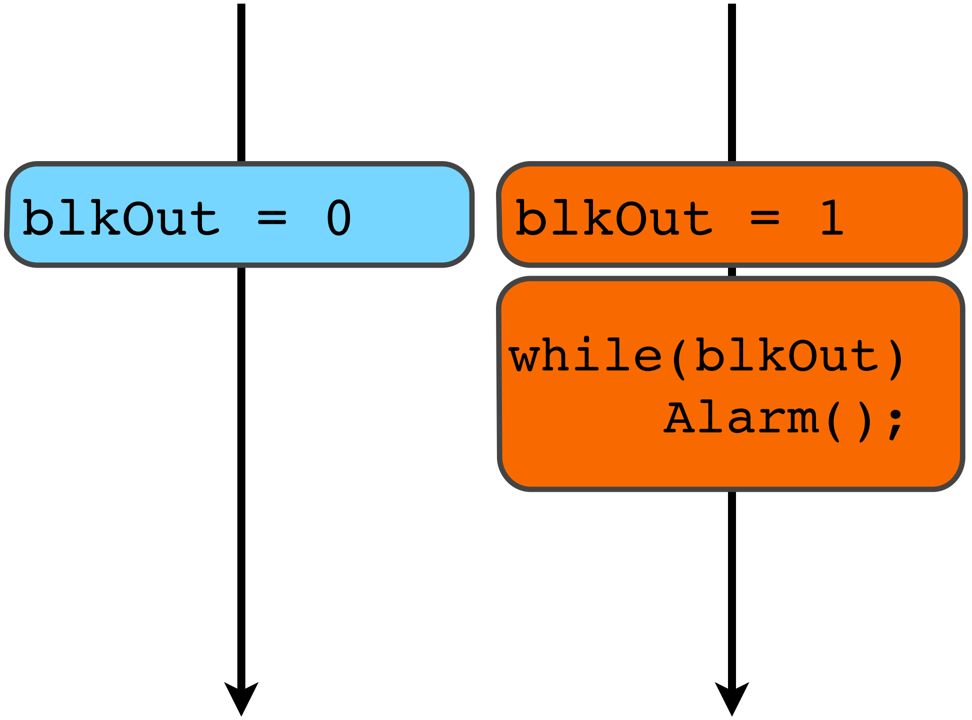
Serial mode: used for synchronization (*e.g.* atomic ops)

CoreDet: Implementation

- A **compiler**
  - instruments the code with calls to the runtime
  - static optimizations to remove instrumentation
- A **runtime library**
  - scheduling threads
  - tracks inter-thread communication
  - deterministic wrappers for: pthreads, malloc, etc...

Bugaboo

From Interleavings To Communication



```
blkOut = 0
```

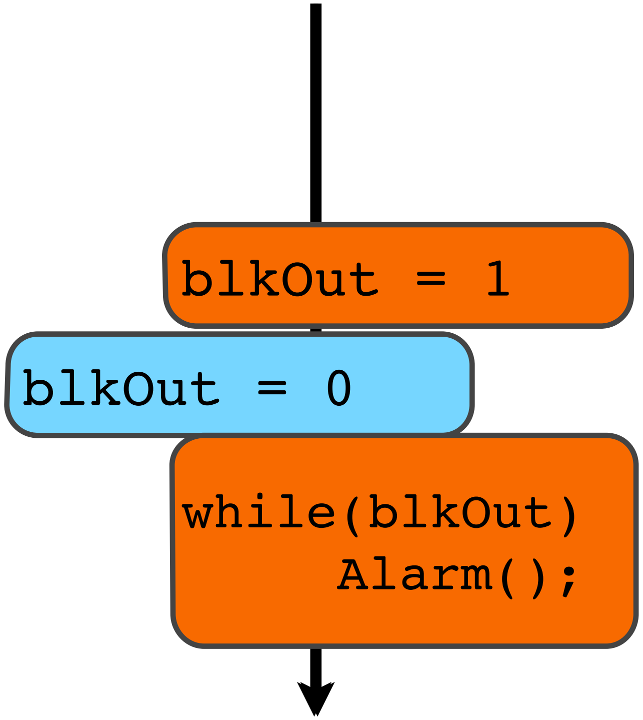
```
blkOut = 1
```

```
while(blkOut)
      Alarm();
```

Interleaving

*http://www.availabilitydigest.com/private/0203/northeast_blackout.pdf

From Interleavings To Communication



```
blkOut = 1
```

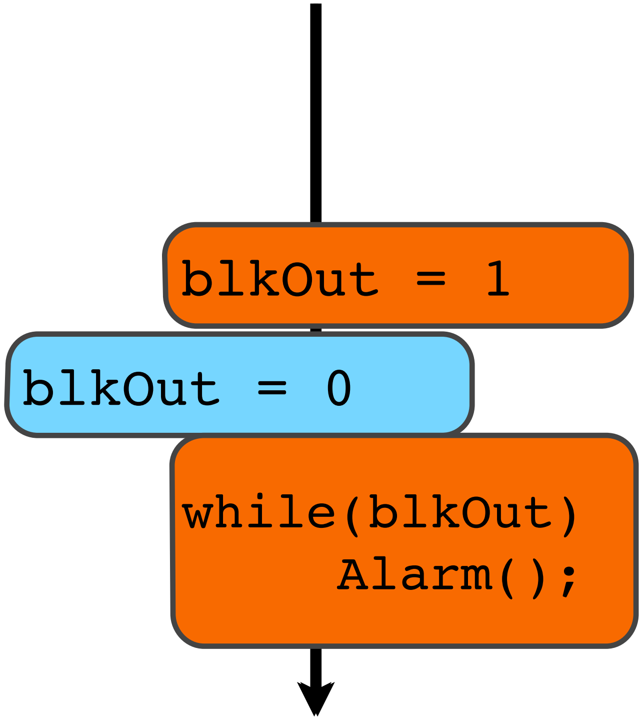
```
blkOut = 0
```

```
while(blkOut)
     Alarm();
```

Interleaving

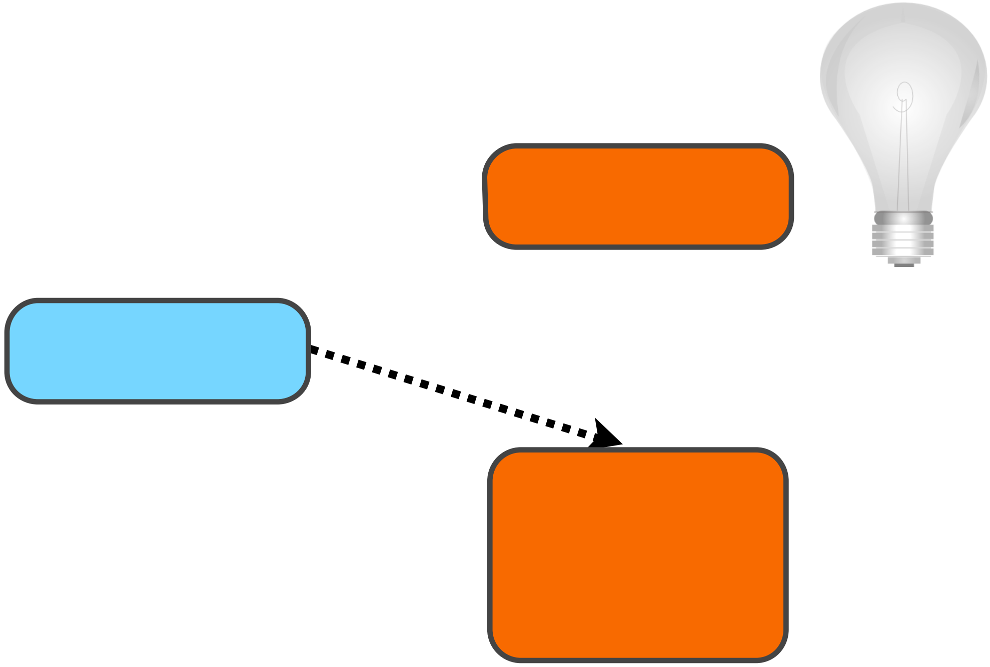
*http://www.availabilitydigest.com/private/0203/northeast_blackout.pdf

From Interleavings To Communication



```
blkOut = 1
blkOut = 0
while(blkOut)
     Alarm();
```

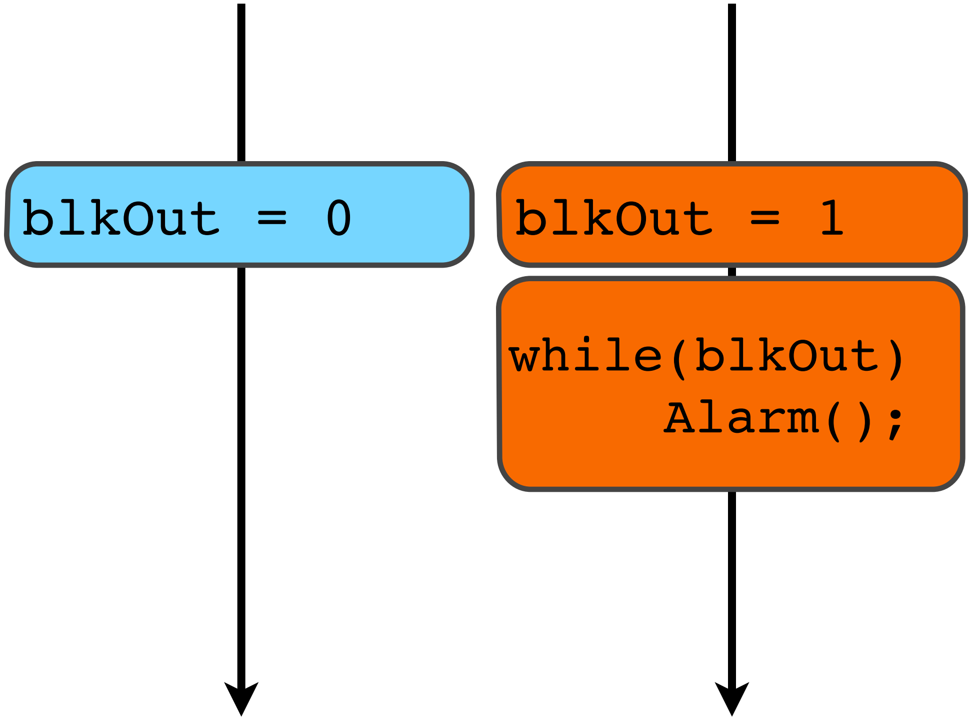
Interleaving



Communication
(via blkOut)

*http://www.availabilitydigest.com/private/0203/northeast_blackout.pdf

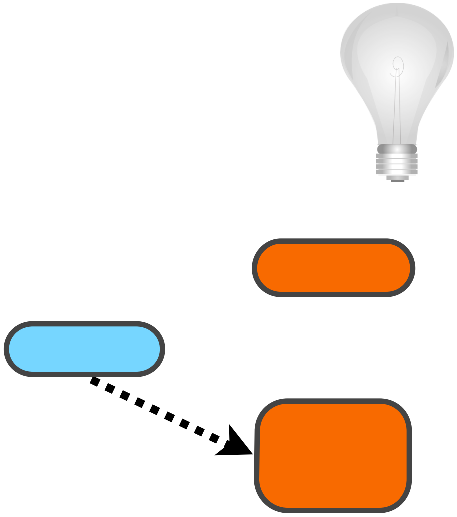
Finding Bugs with Communication Graphs



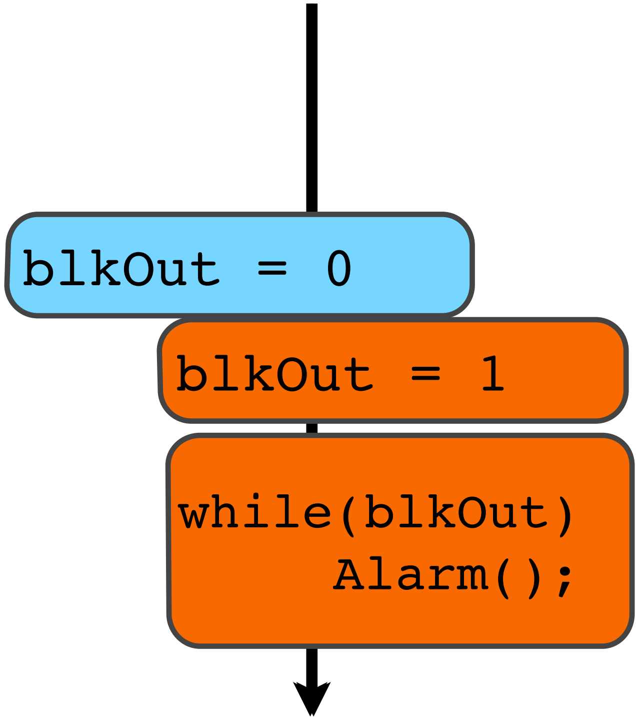
```
blkOut = 0
```

```
blkOut = 1
```

```
while(blkOut)
      Alarm();
```



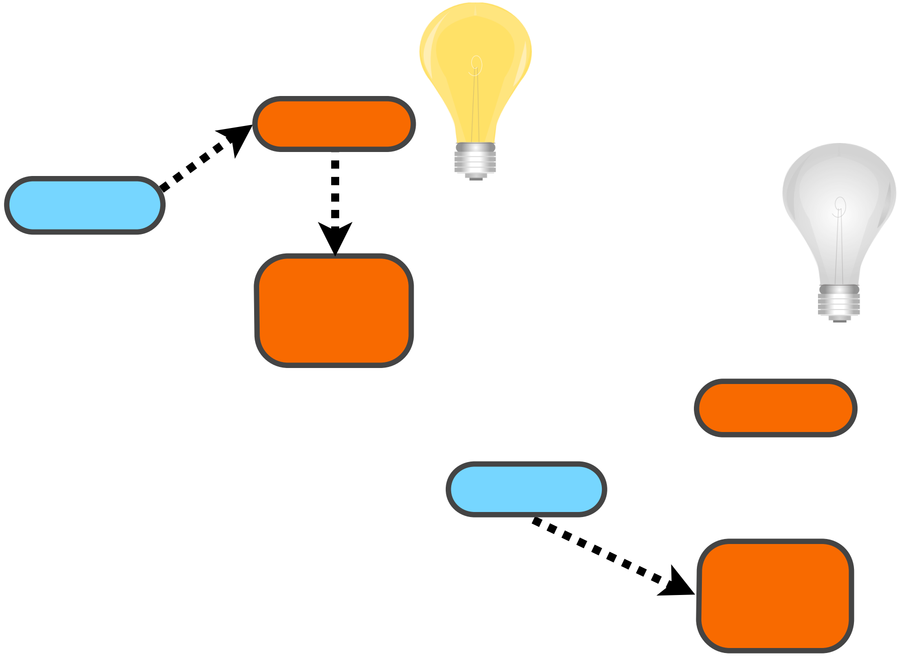
Finding Bugs with Communication Graphs



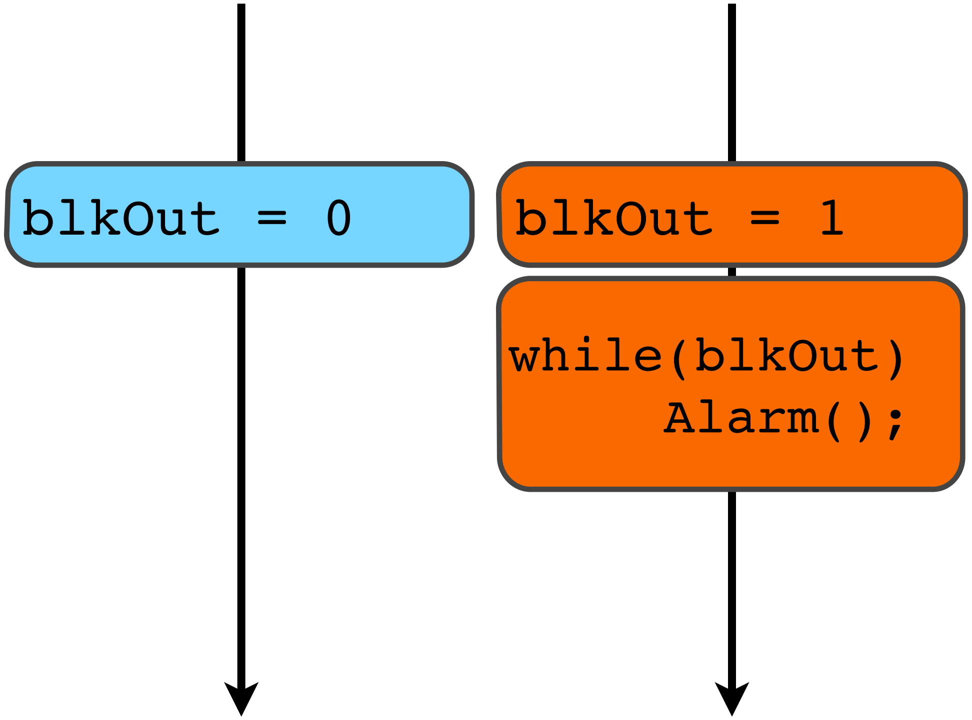
```
blkOut = 0
```

```
blkOut = 1
```

```
while(blkOut)
      Alarm();
```



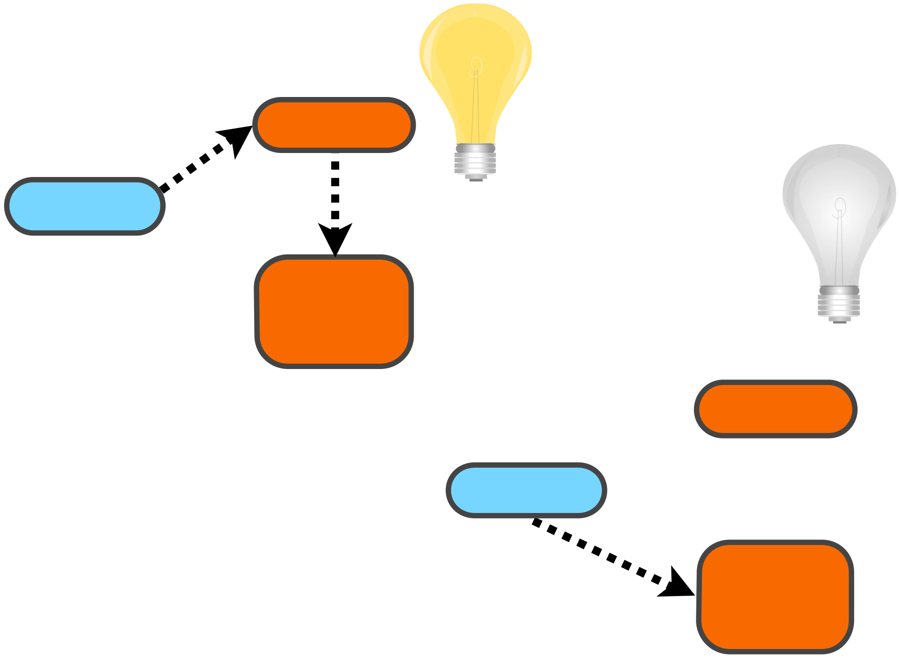
Finding Bugs with Communication Graphs



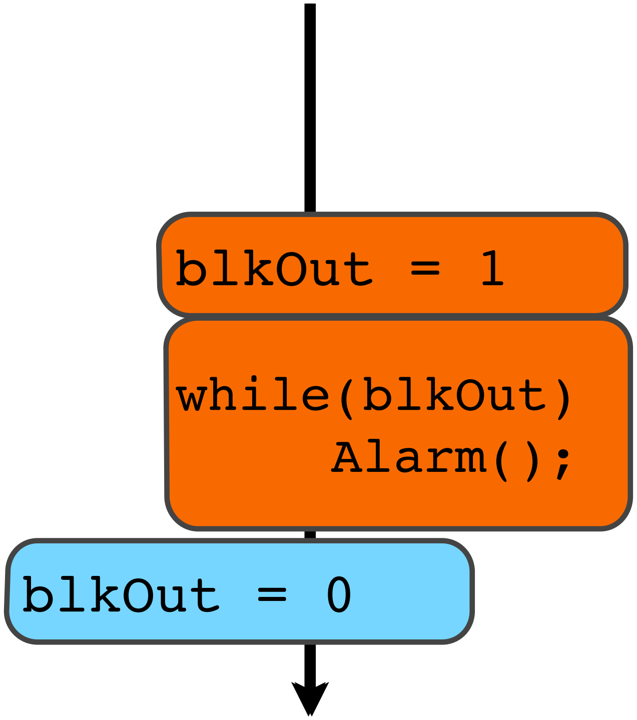
```
blkOut = 0
```

```
blkOut = 1
```

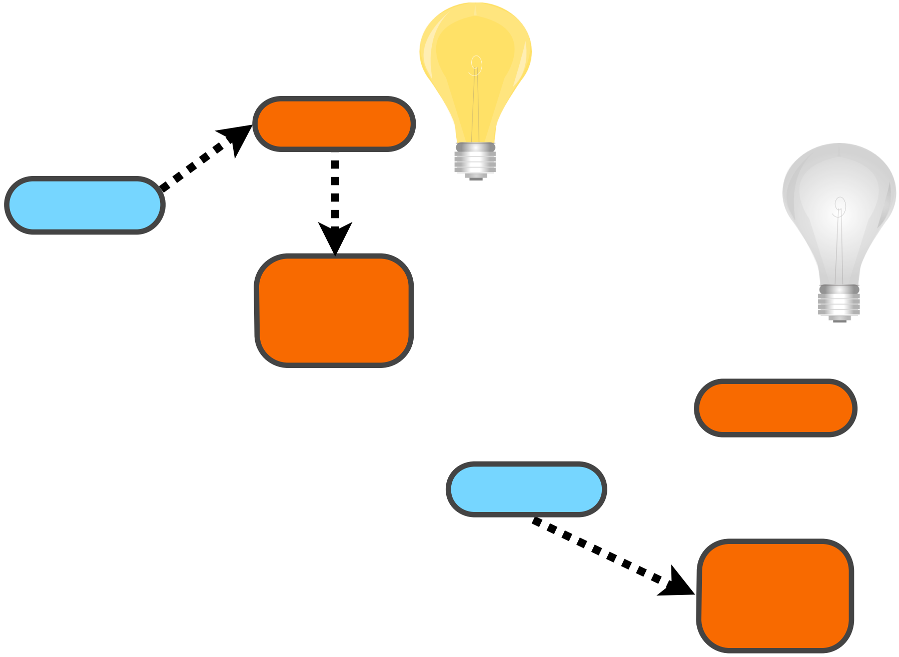
```
while(blkOut)
      Alarm();
```



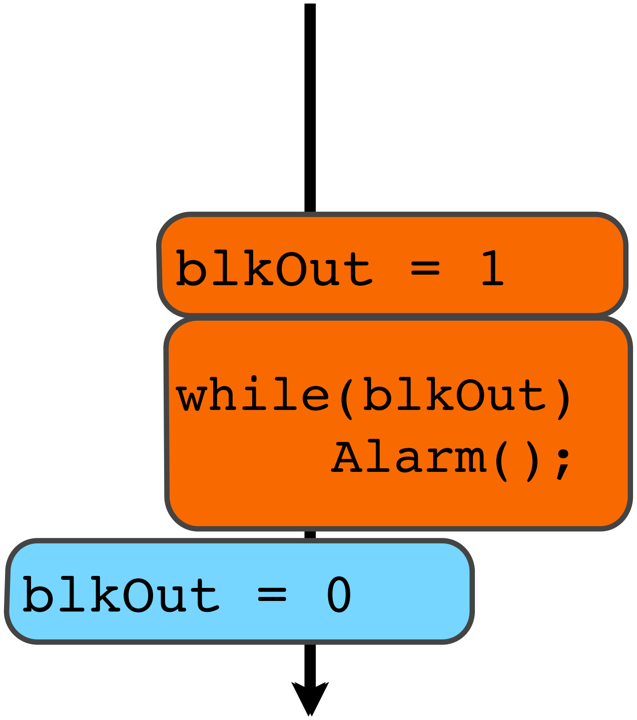
Finding Bugs with Communication Graphs



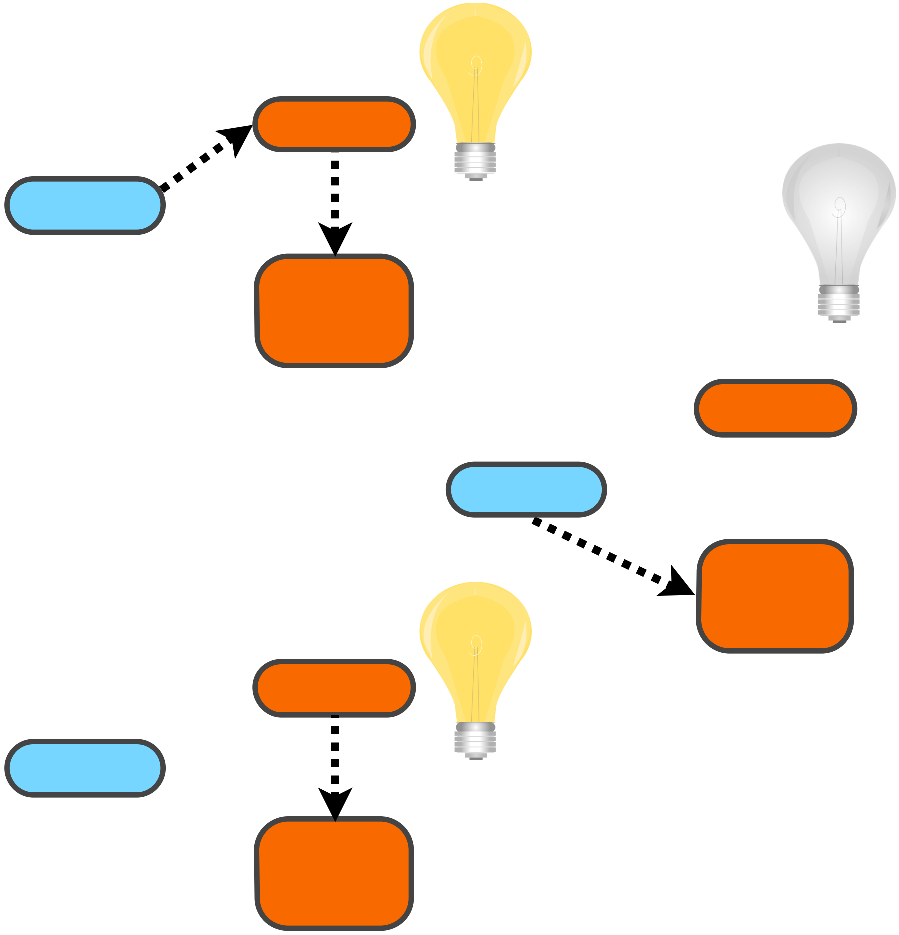
```
blkOut = 1

while(blkOut)
      Alarm();

blkOut = 0
```



Finding Bugs with Communication Graphs

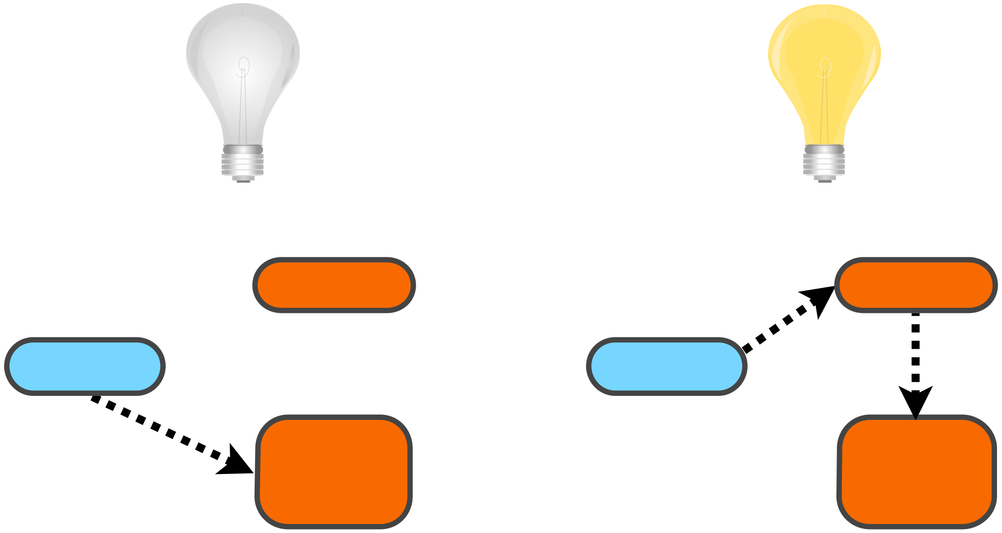


```
blkOut = 1
while(blkOut)
      Alarm();
blkOut = 0
```



Debugging With Communication Graphs From 10,000’

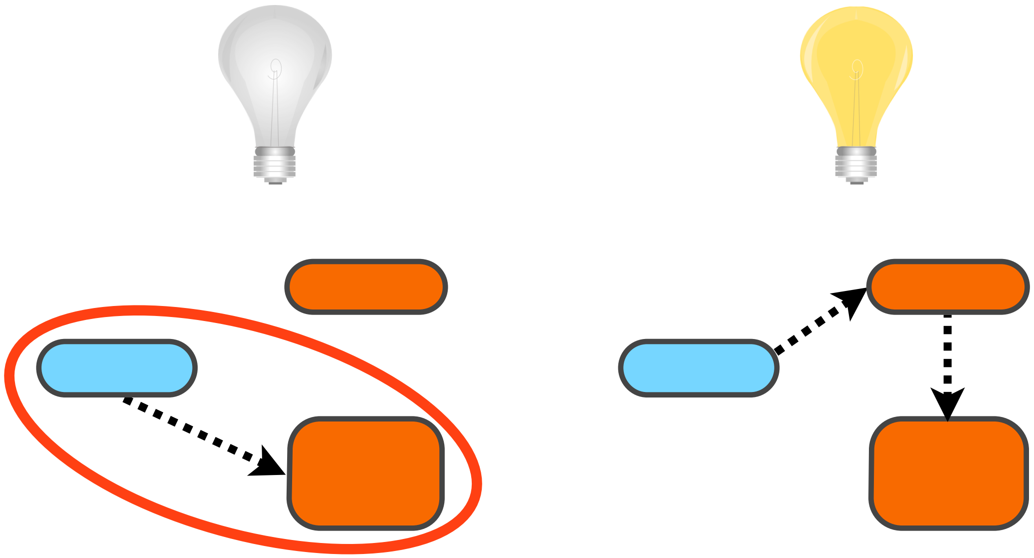
1. Collect communication graphs, and label them as Buggy or Correct



Debugging With Communication Graphs From 10,000’

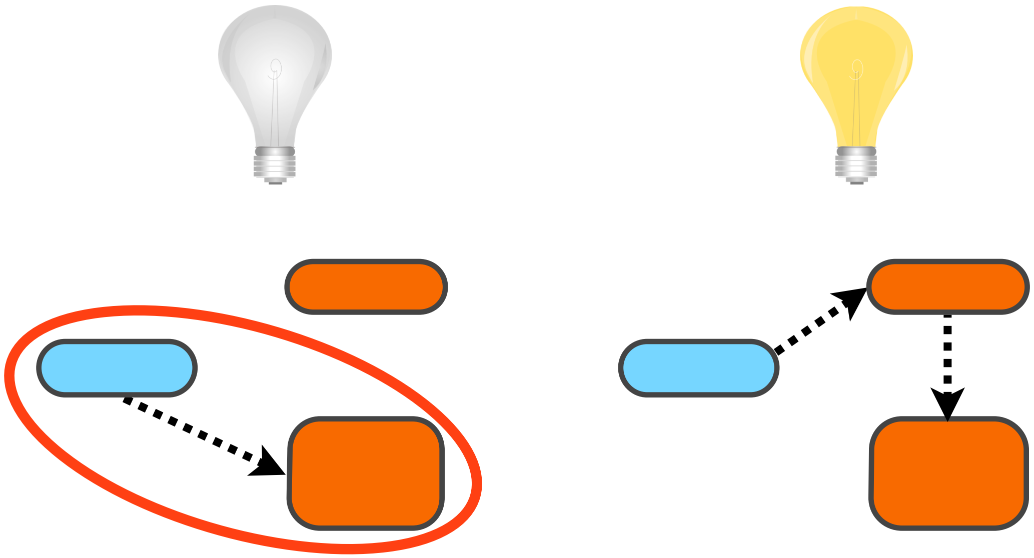
1. Collect communication graphs, and label them as Buggy or Correct

2. Identify edges in Buggy graphs, but not in Correct graphs



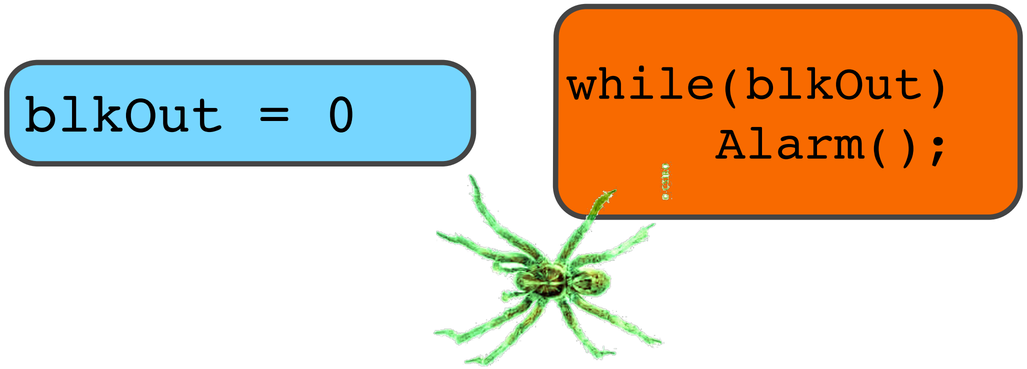
Debugging With Communication Graphs From 10,000’

1. Collect communication graphs, and label them as Buggy or Correct



2. Identify edges in Buggy graphs, but not in Correct graphs

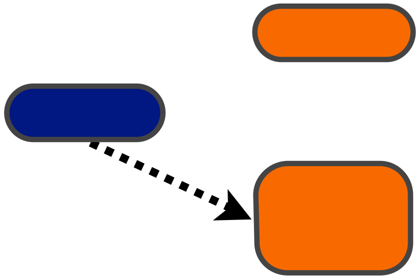
3. Inspect code involved in Buggy-only edges



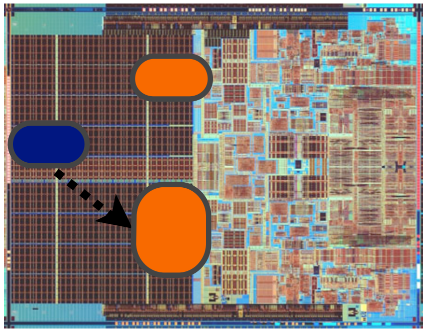
```
blkOut = 0
```

```
while(blkOut)
     Alarm();
```

System Design Requirements

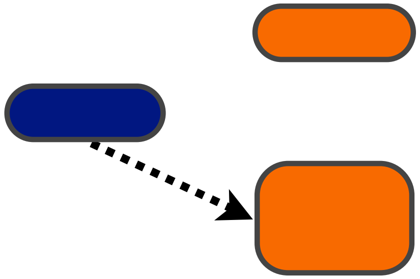


Graphs Must Encode Enough Information to Identify Buggy Communication



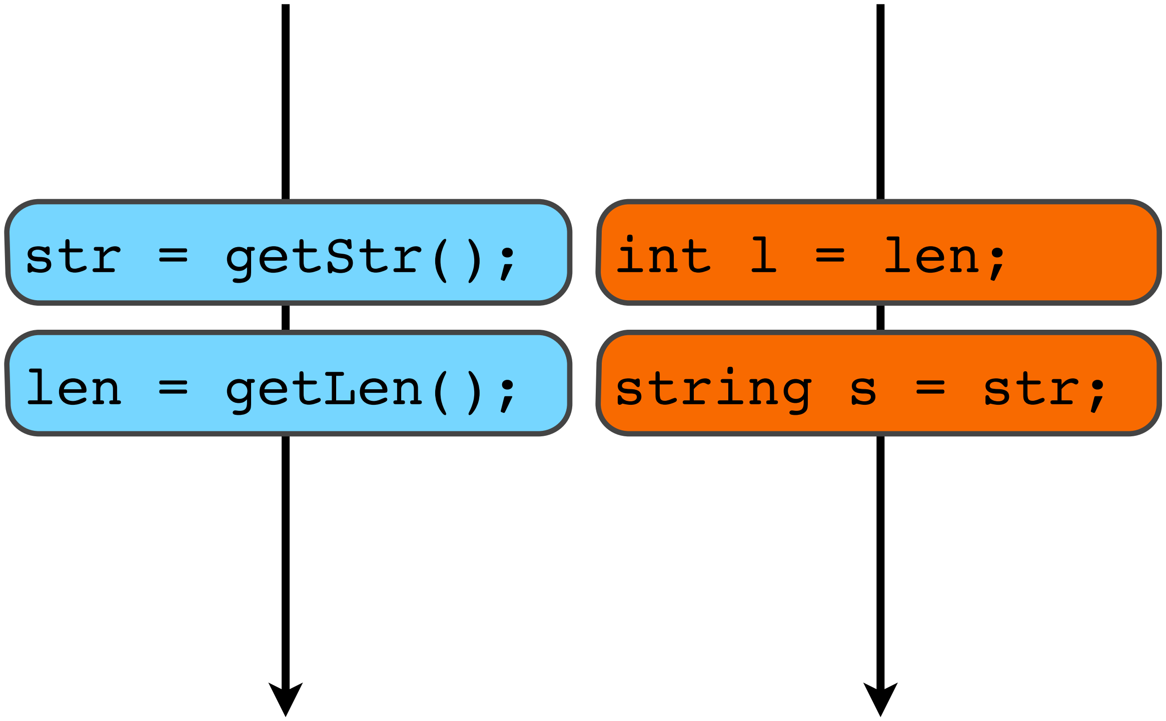
Graph Collection Must be Cheap

Debugging Must Be Simple



Making Useful Communication Graphs

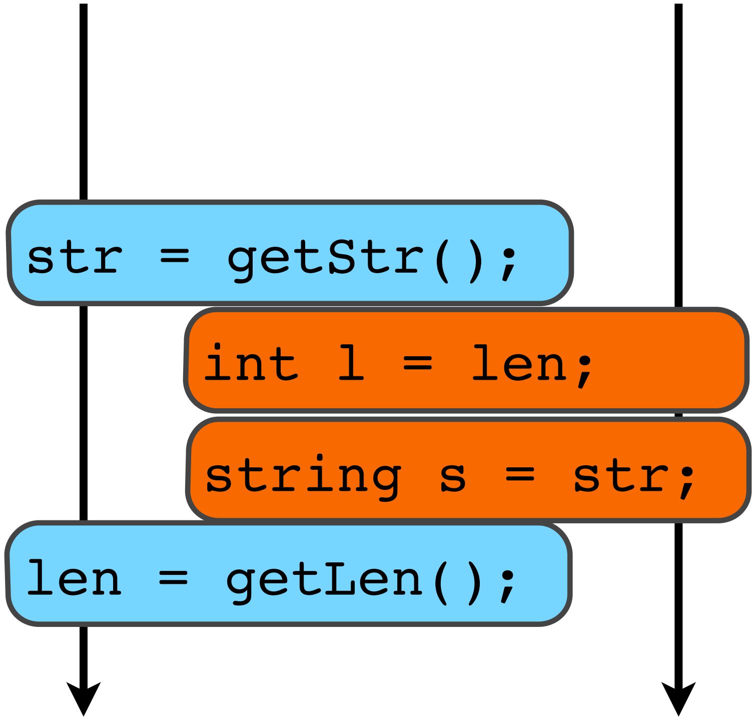
A More Interesting Example



```
str = getStr();
len = getLen();
```

```
int l = len;
string s = str;
```

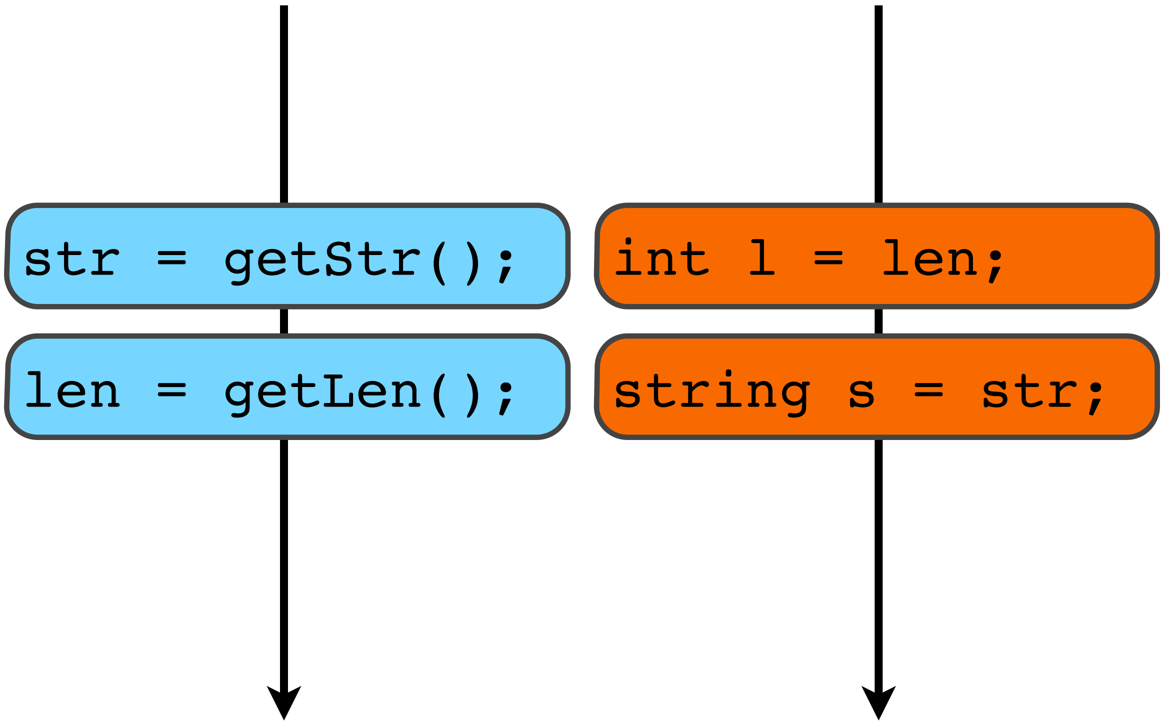
A More Interesting Example



```
str = getStr();
        int l = len;
        string s = str;
len = getLen();
```

Multi-Variable Atomicity Violation can result in reads of inconsistent `str` and `len`

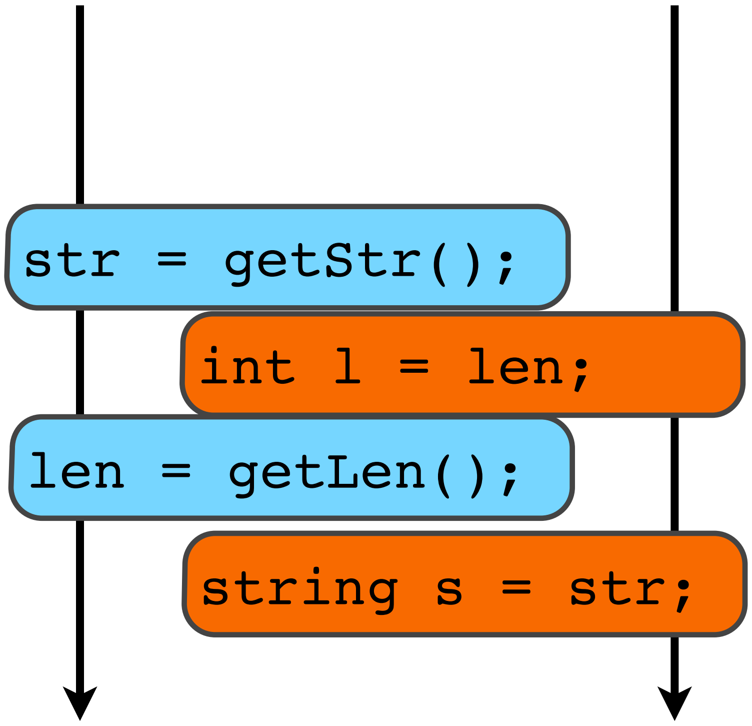
Communication Alone Is Insufficient



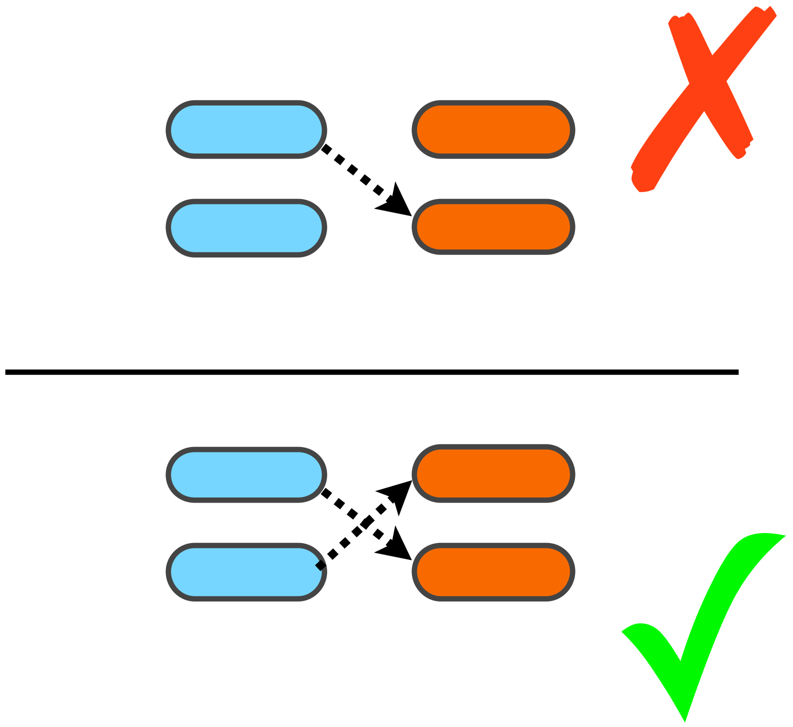
```
str = getStr();
len = getLen();
```

```
int l = len;
string s = str;
```

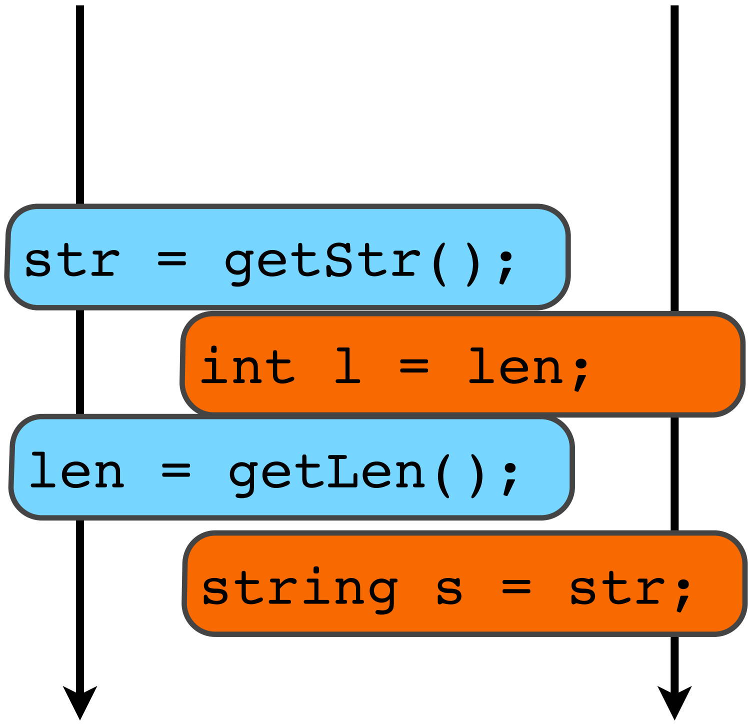
Communication Alone Is Insufficient



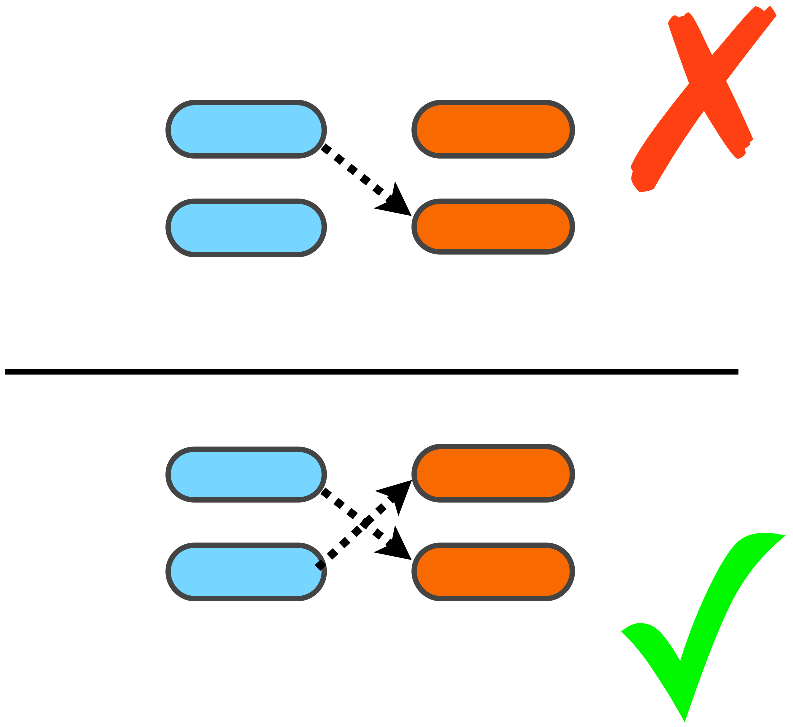
```
str = getStr();
int l = len;
len = getLen();
string s = str;
```



Communication Alone Is Insufficient

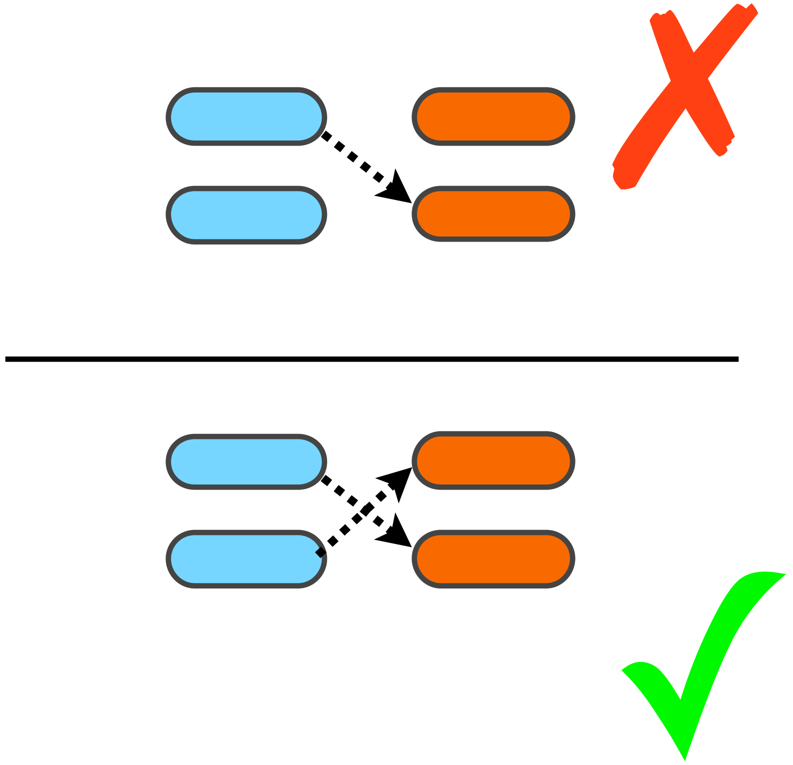


```
str = getStr();
        int l = len;
len = getLen();
        string s = str;
```



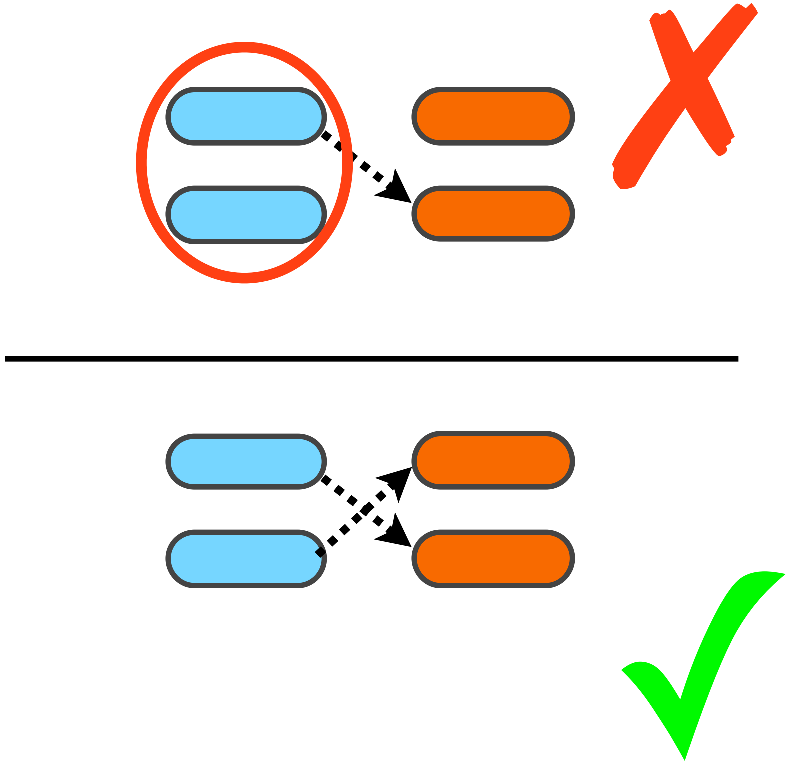
There is no edge in the Buggy graph that isn't in the Correct graph!

Adding Context to Graphs



Adding Context to Graphs

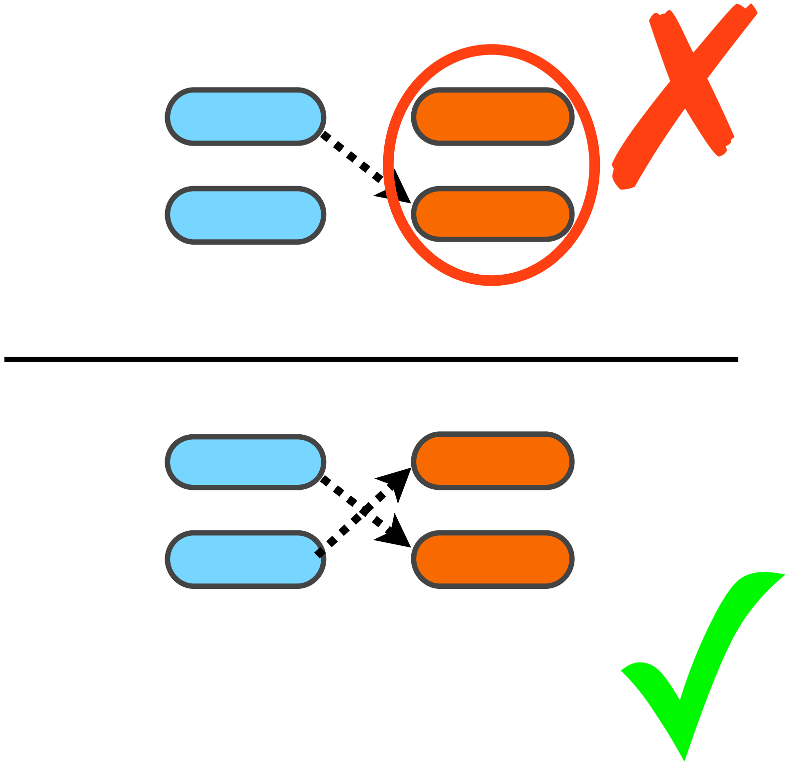
These writes should not be interleaved...



Adding Context to Graphs

These writes should not be interleaved...

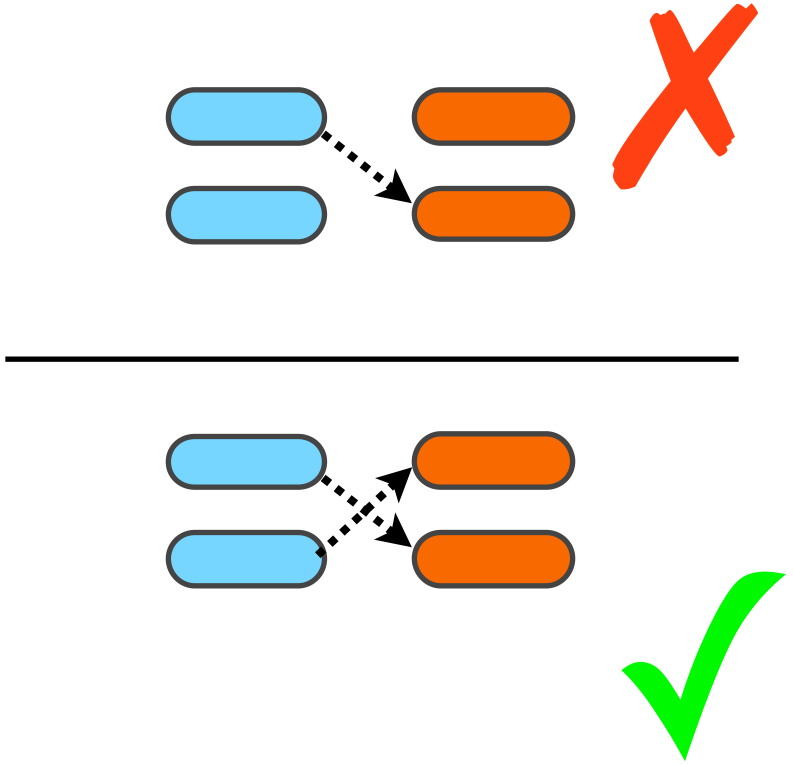
...so these instructions should be ***ordered*** before, or after ***both*** writes



Adding Context to Graphs

These writes should not be interleaved...

...so these instructions should be ***ordered*** before, or after ***both*** writes

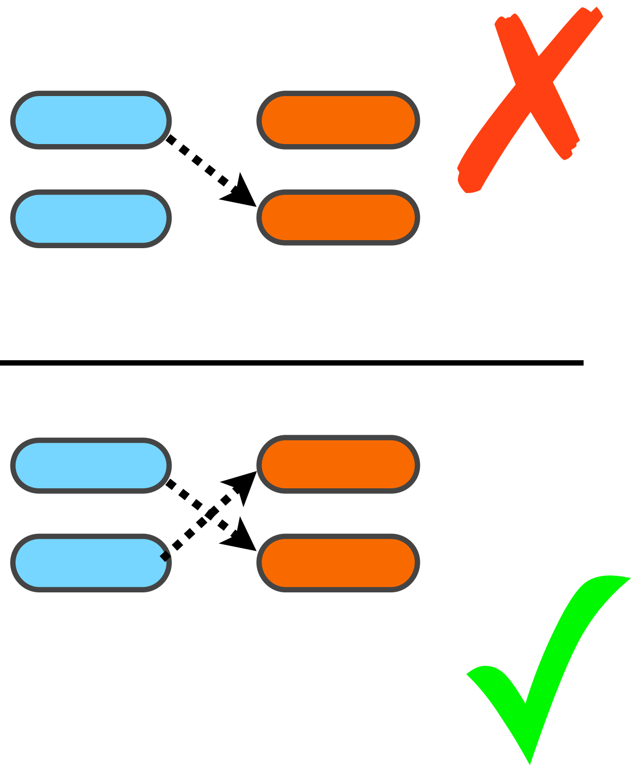


Adding Context to Graphs

These writes should not be interleaved...

...so these instructions should be ***ordered*** before, or after ***both*** writes

Communication graphs do not encode ***relative ordering*** of communications

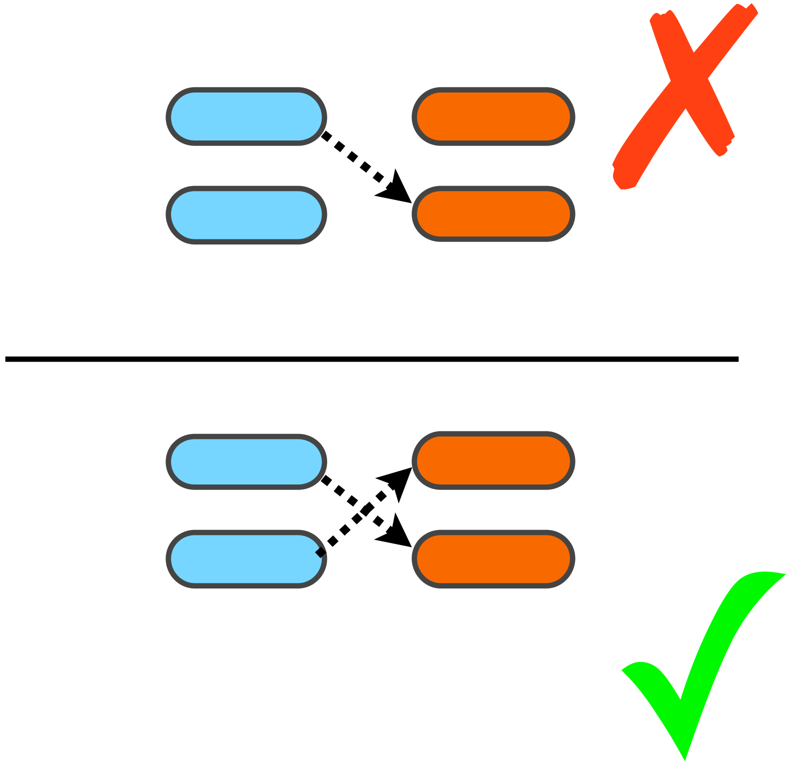


Adding Context to Graphs

These writes should not be interleaved...

...so these instructions should be ***ordered*** before, or after ***both*** writes

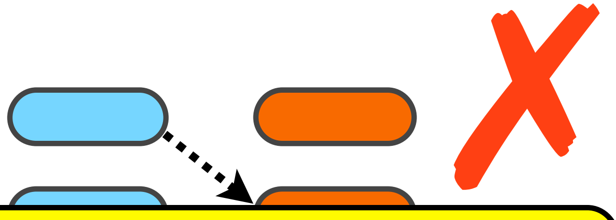
Communication graphs do not encode ***relative ordering*** of communications



Communication Context is a short history of preceding communication events added to each node

Adding Context to Graphs

These writes should not be interleaved...



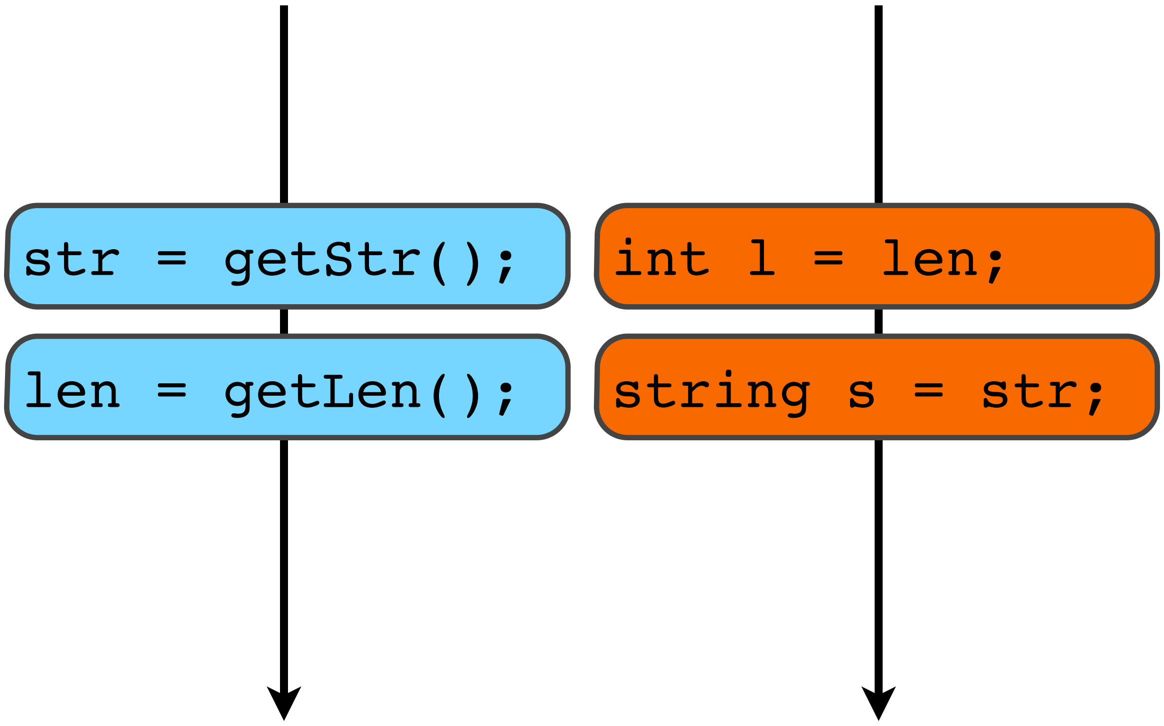
Context encodes ordering amongst communication events, enabling more general bug detection

relative ordering of communications

✓

Communication Context is a short history of preceding communication events added to each node

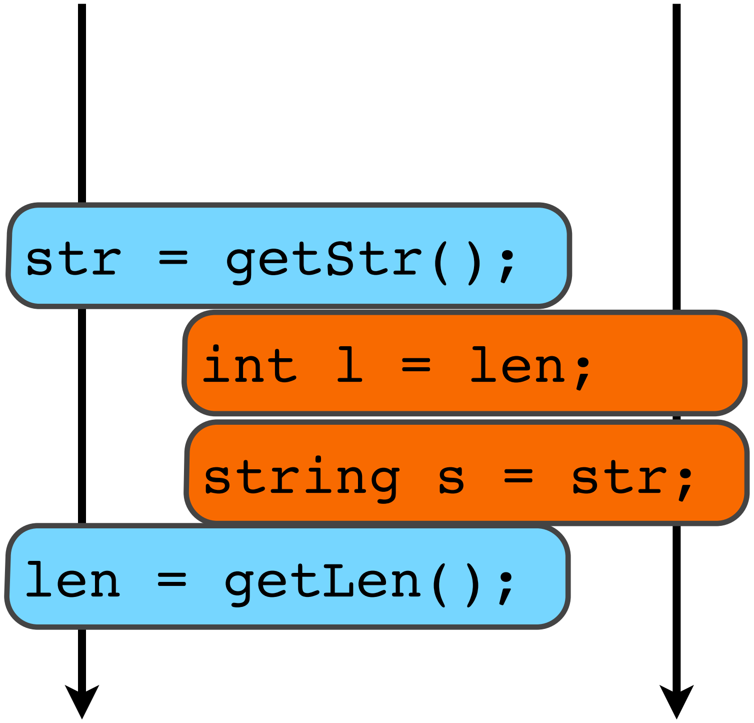
Context-Aware Communication Graphs



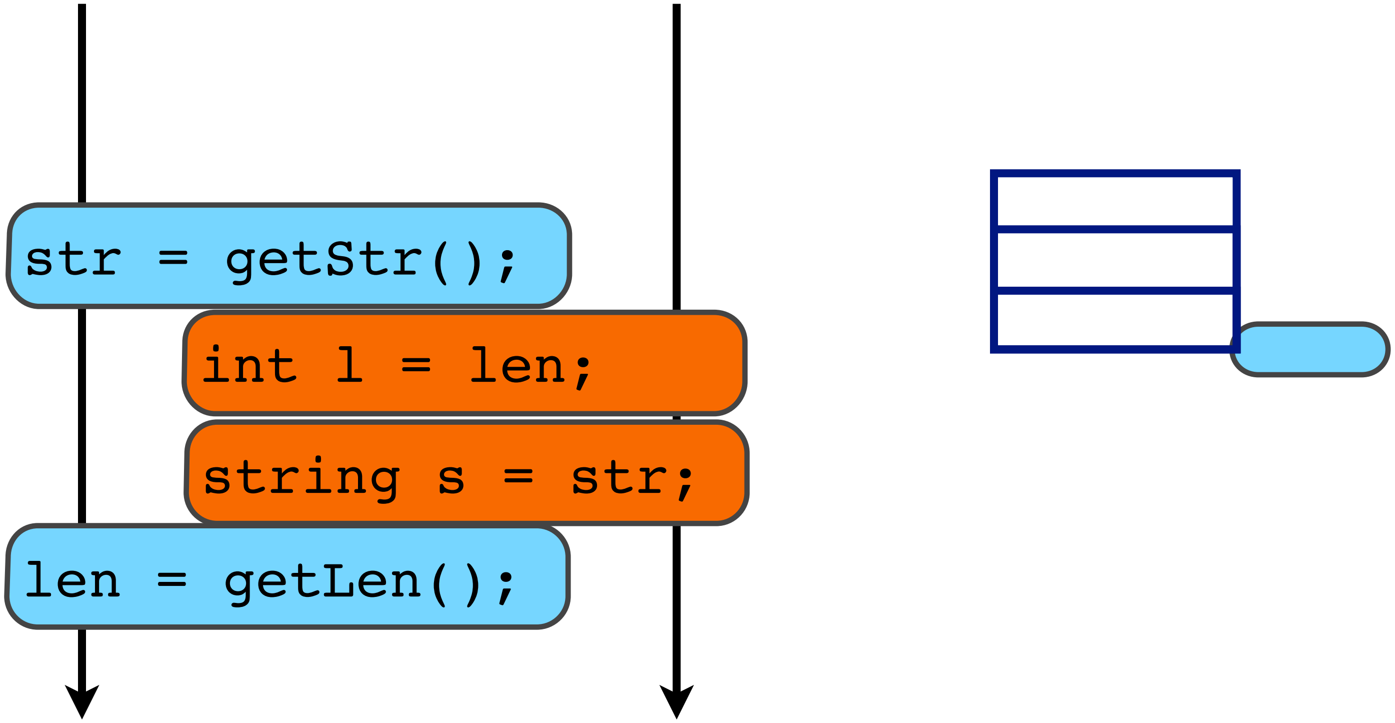
```
str = getStr();
len = getLen();
```

```
int l = len;
string s = str;
```

Context-Aware Communication Graphs

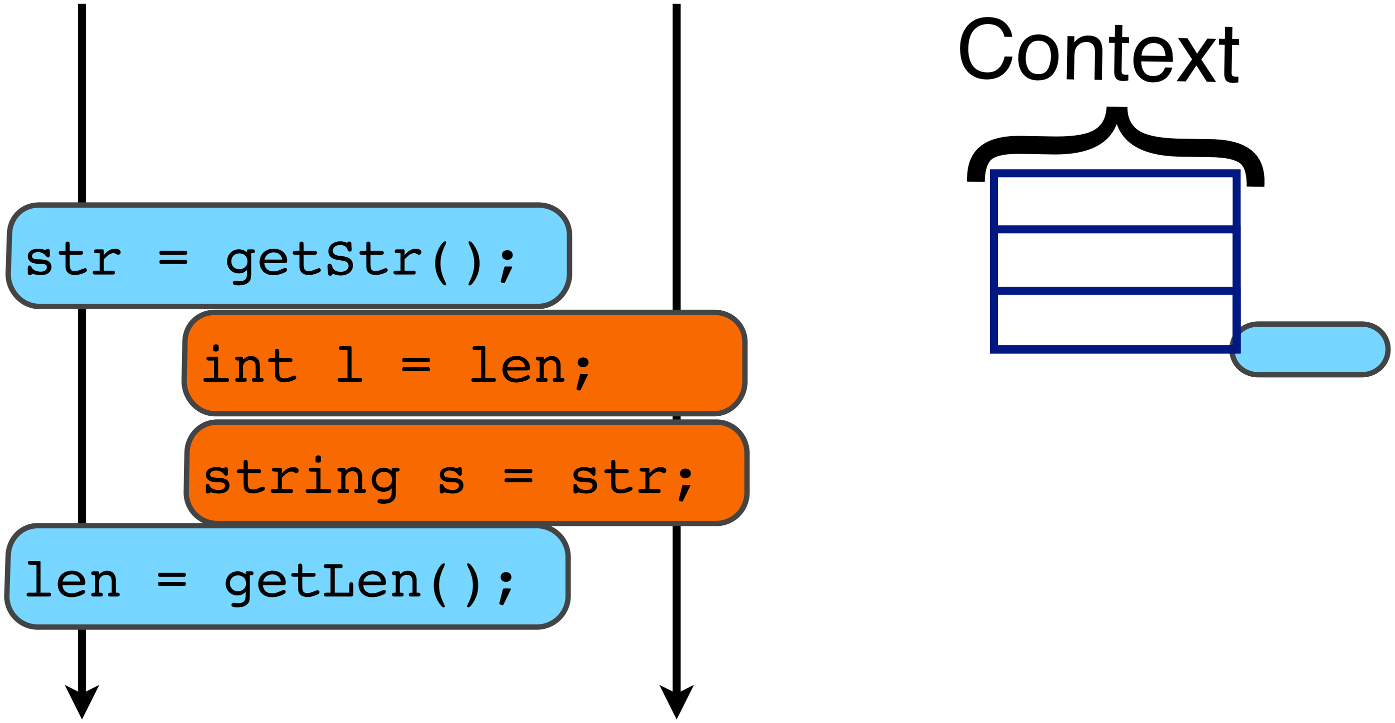


Context-Aware Communication Graphs



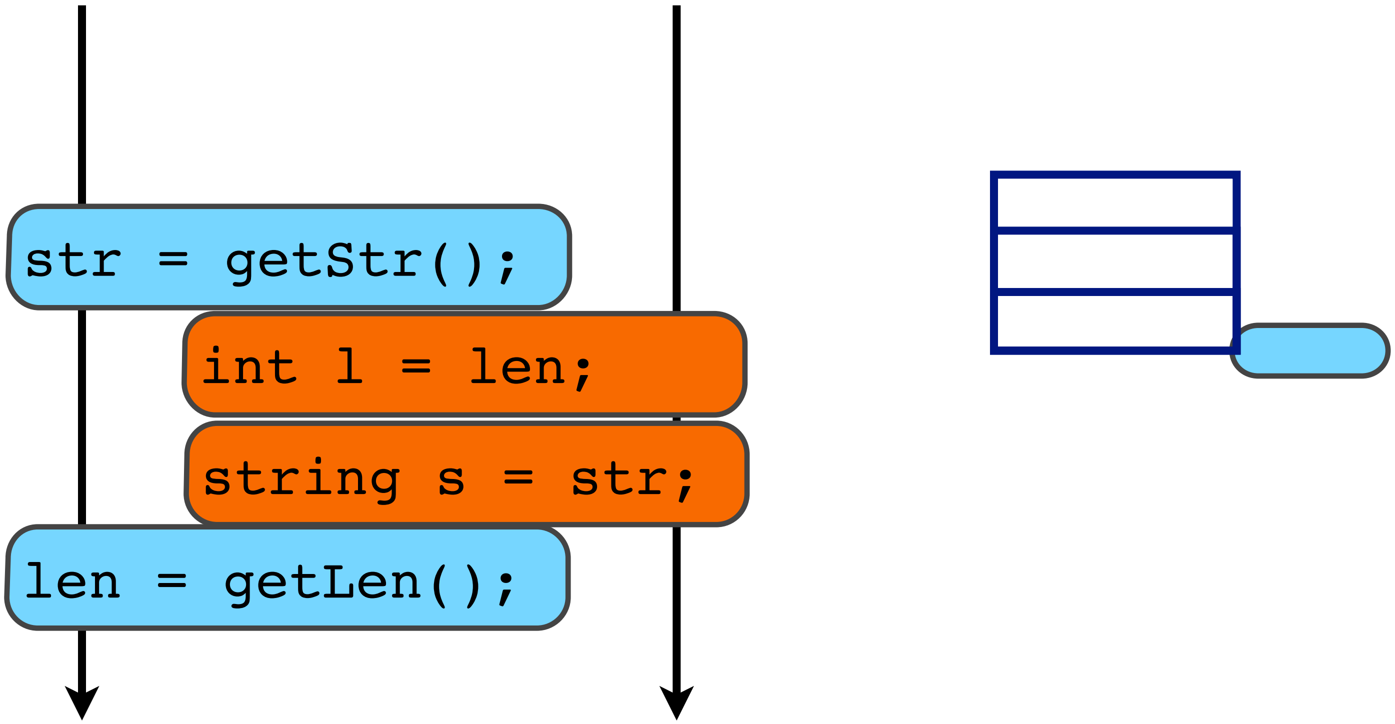
```
str = getStr();
int l = len;
string s = str;
len = getLen();
```

Context-Aware Communication Graphs



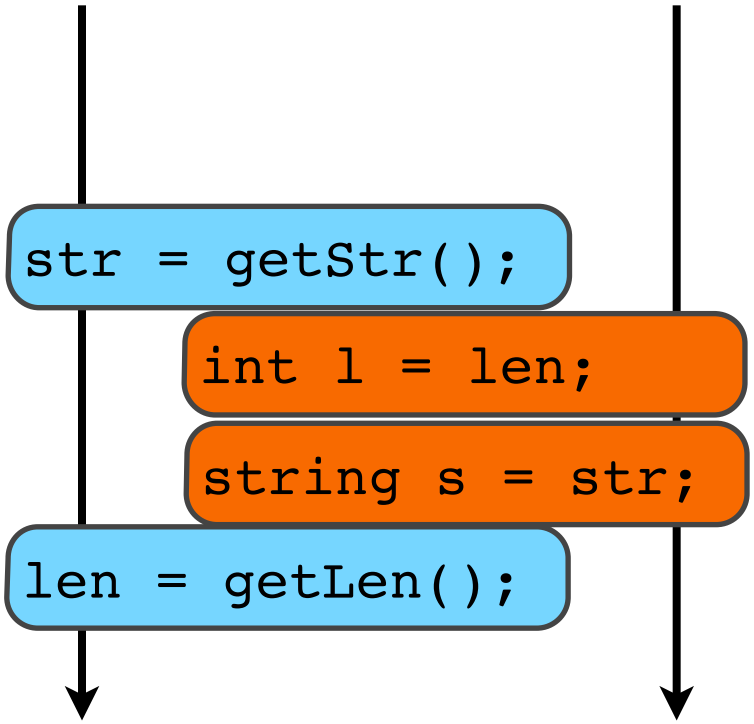
```
str = getStr();
int l = len;
string s = str;
len = getLen();
```

Context-Aware Communication Graphs

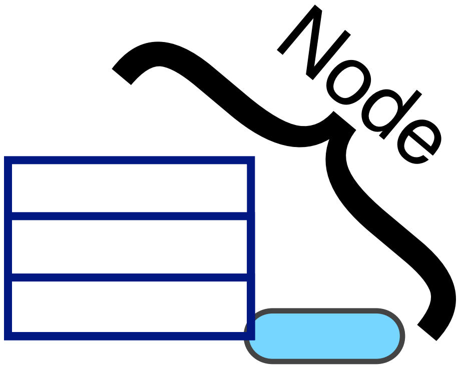


```
str = getStr();
int l = len;
string s = str;
len = getLen();
```

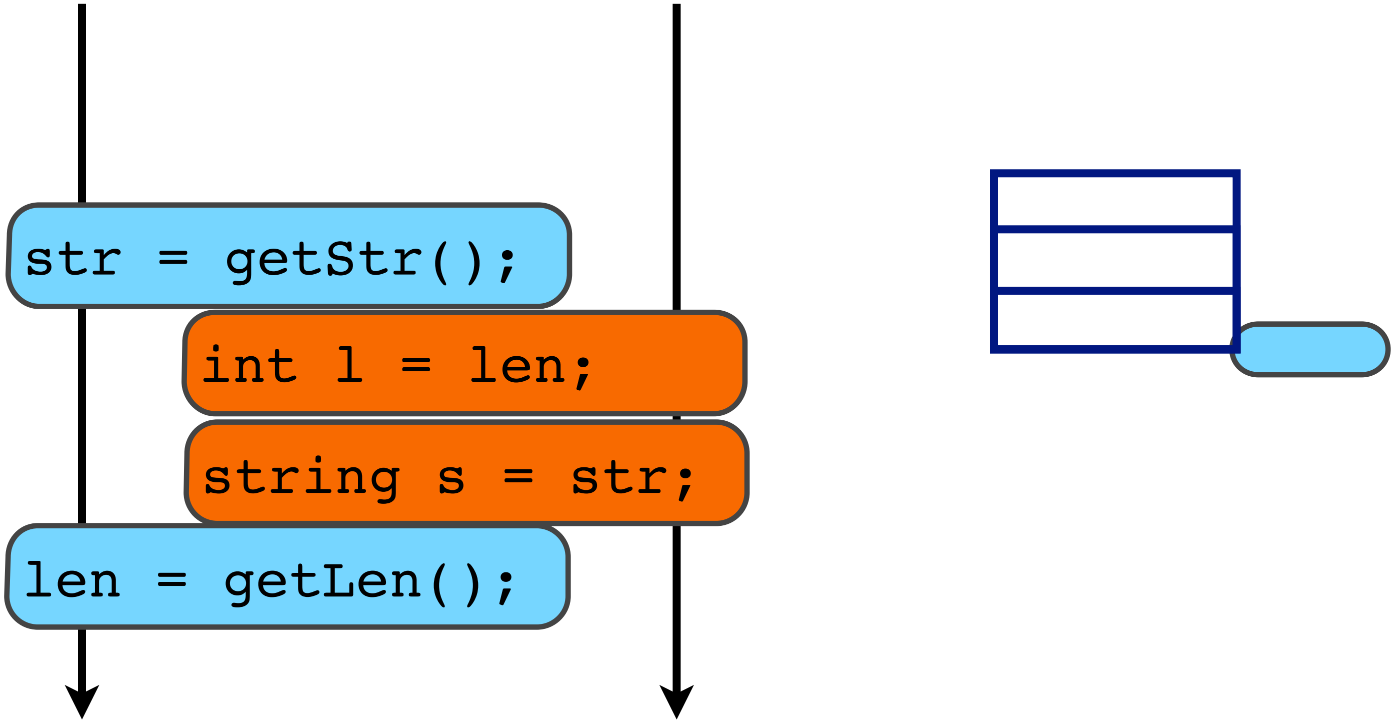
Context-Aware Communication Graphs



```
str = getStr();
            int l = len;
            string s = str;
len = getLen();
```

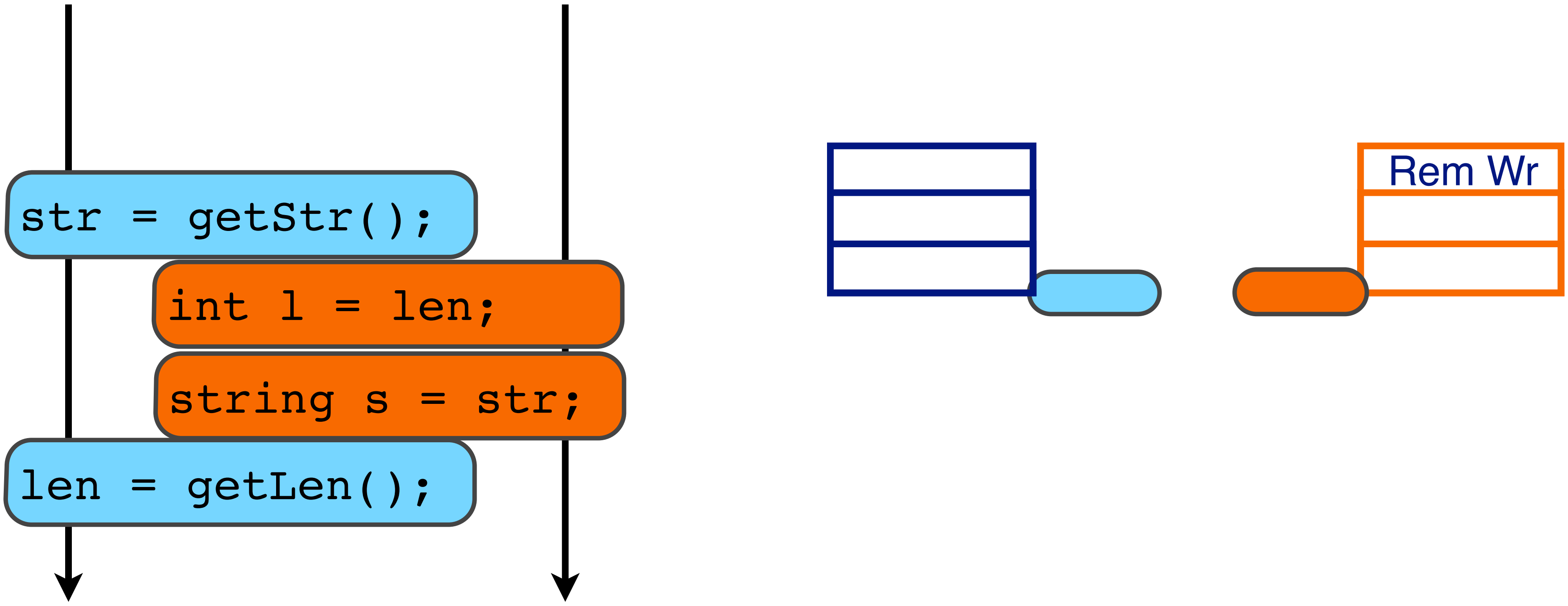


Context-Aware Communication Graphs



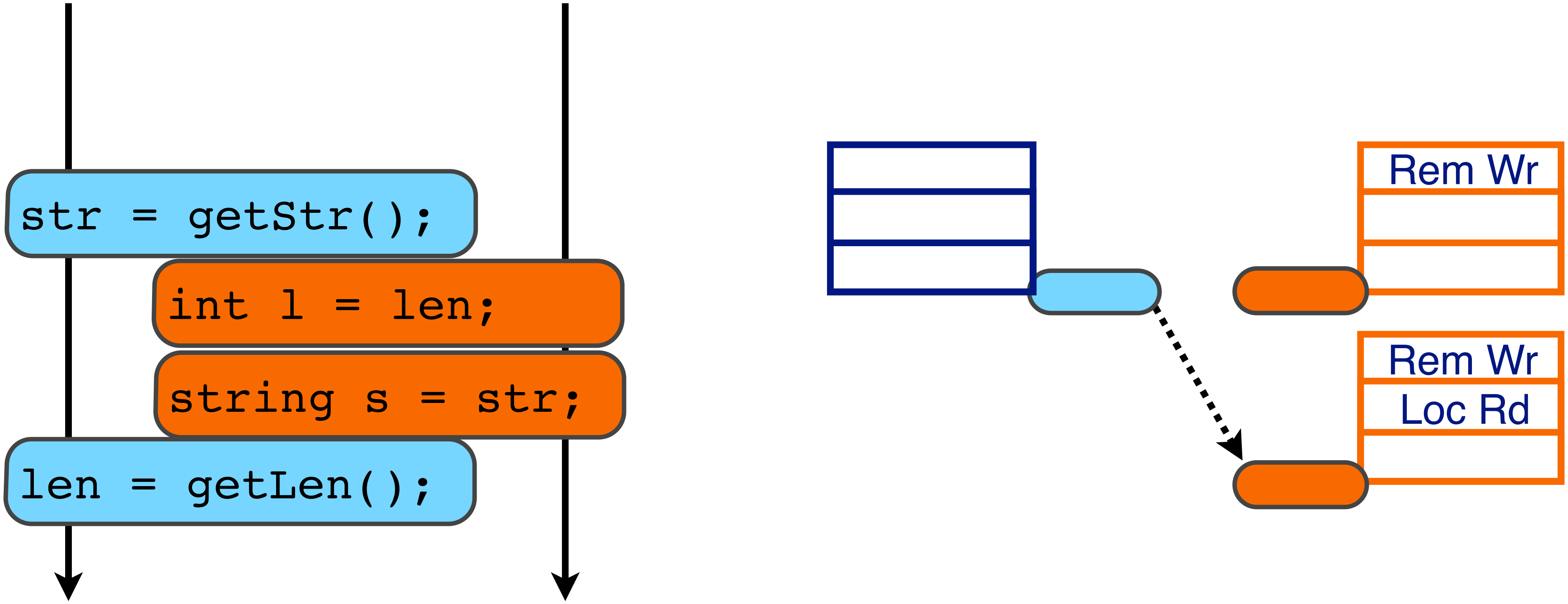
```
str = getStr();
int l = len;
string s = str;
len = getLen();
```

Context-Aware Communication Graphs



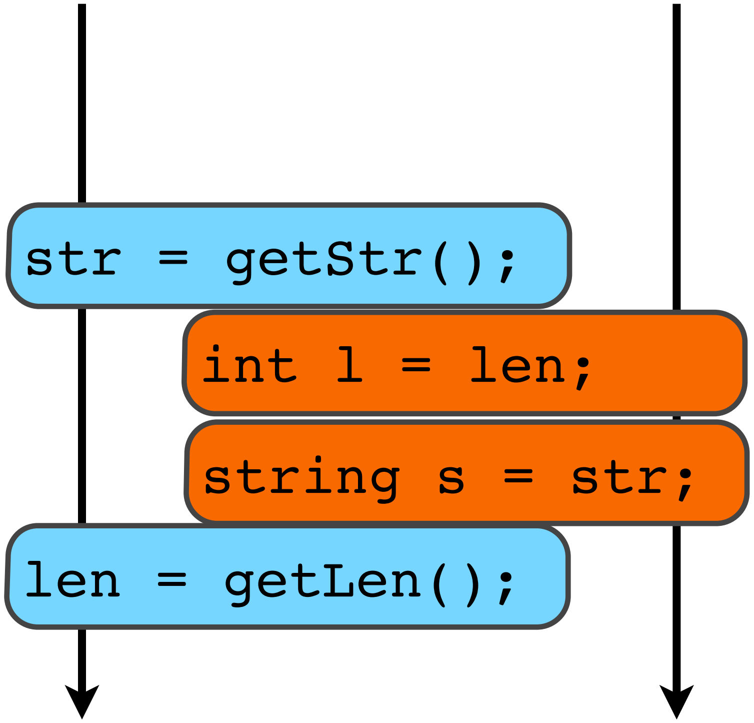
```
str = getStr();
int l = len;
string s = str;
len = getLen();
```

Context-Aware Communication Graphs

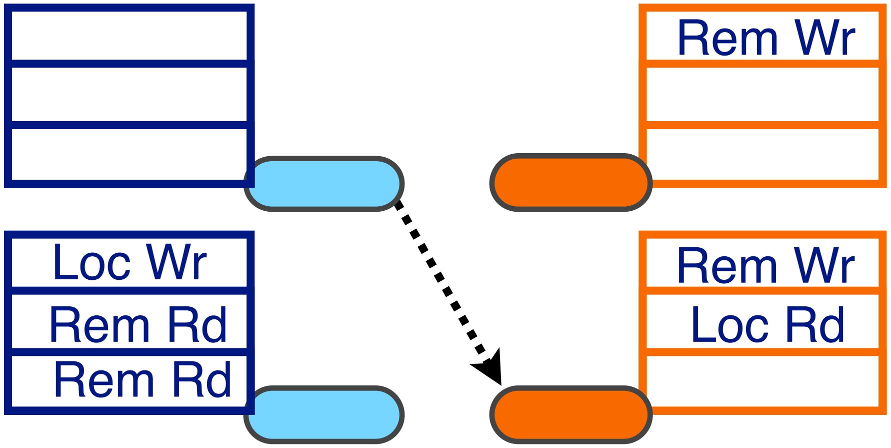


```
str = getStr();
int l = len;
string s = str;
len = getLen();
```

Context-Aware Communication Graphs



```
str = getStr();
        int l = len;
        string s = str;
len = getLen();
```

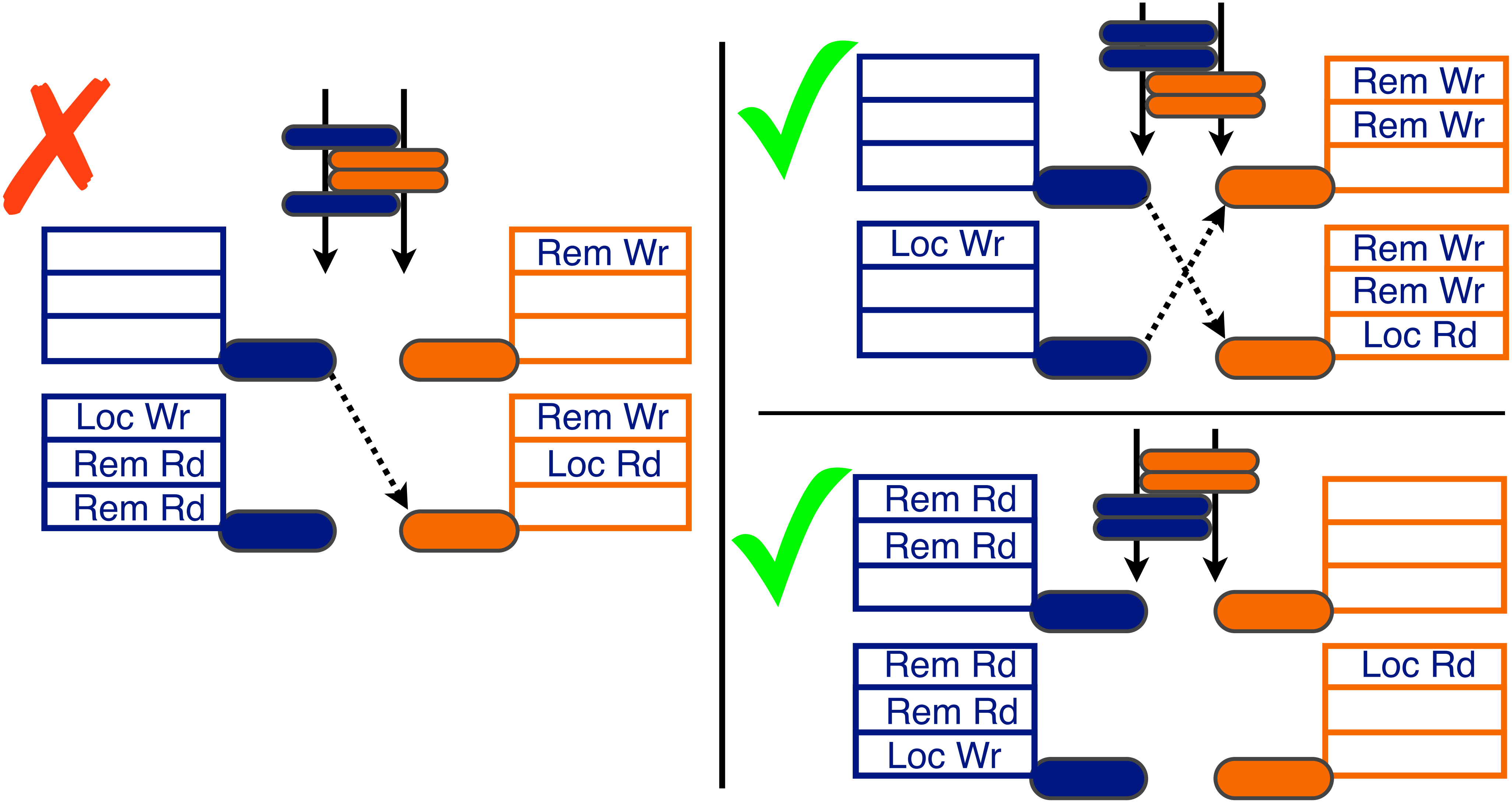


Loc Wr
Rem Rd
Rem Rd

Rem Wr

Rem Wr
Loc Rd

Context-Aware Communication Graphs



✗

Loc Wr
Rem Rd
Rem Rd

Rem Wr

Rem Wr
Loc Rd

✓

Loc Wr

Rem Wr
Rem Wr

Rem Wr
Rem Wr
Loc Rd

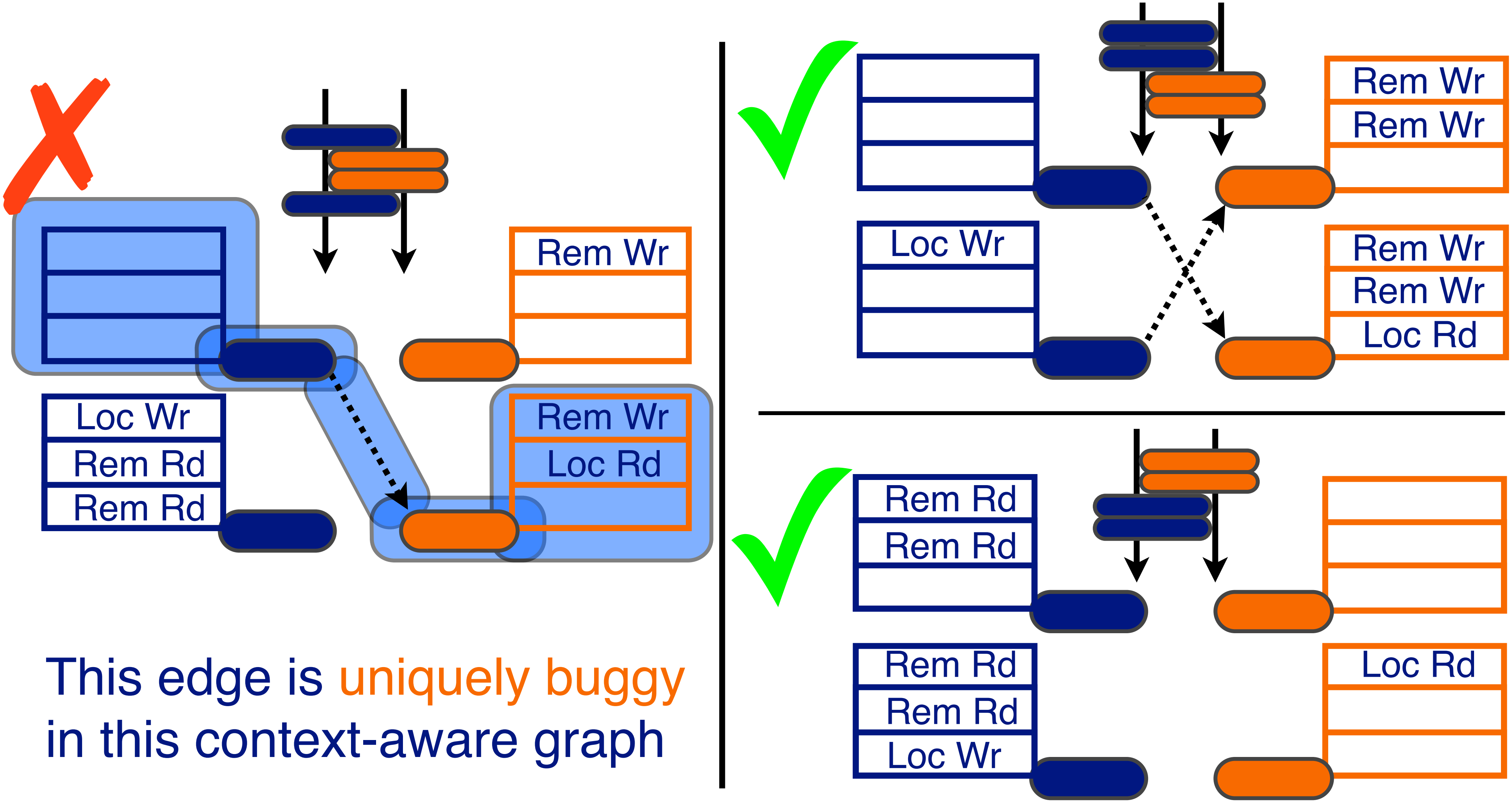
✓

Rem Rd
Rem Rd

Rem Rd
Rem Rd
Loc Wr

Loc Rd

Context-Aware Communication Graphs

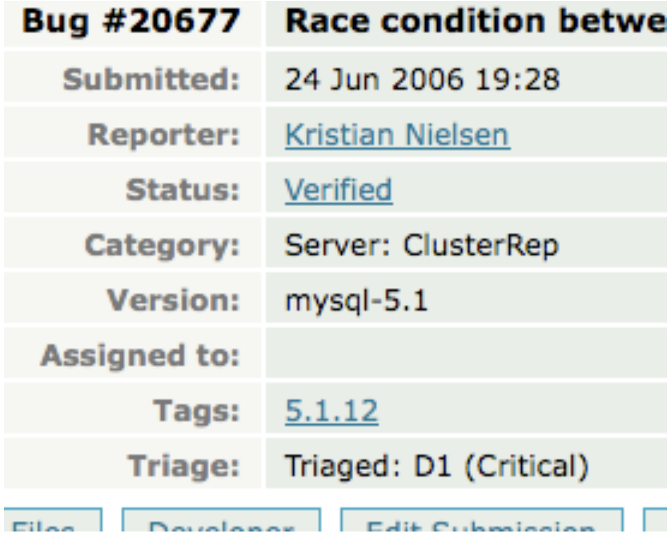


This edge is uniquely buggy in this context-aware graph

Debugging With Bugaboo

Labeled Graph Debugging

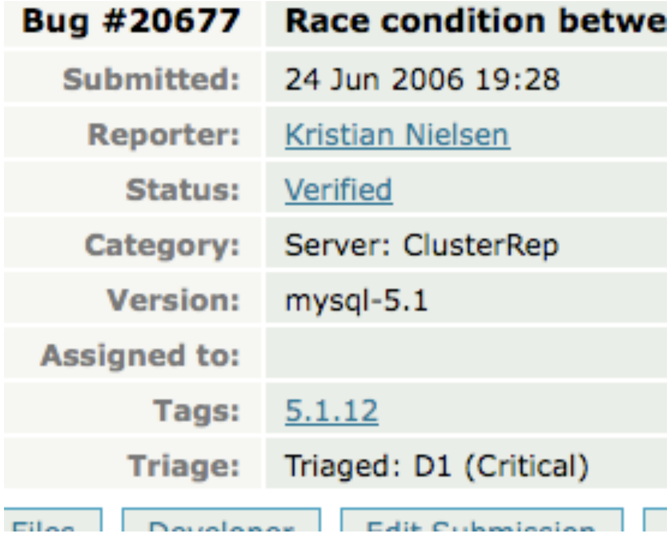
Starting with a bug report
or buggy behavior...



Bug #20677	Race condition betwe
Submitted:	24 Jun 2006 19:28
Reporter:	Kristian Nielsen
Status:	Verified
Category:	Server: ClusterRep
Version:	mysql-5.1
Assigned to:	
Tags:	5.1.12
Triage:	Triaged: D1 (Critical)

Labeled Graph Debugging

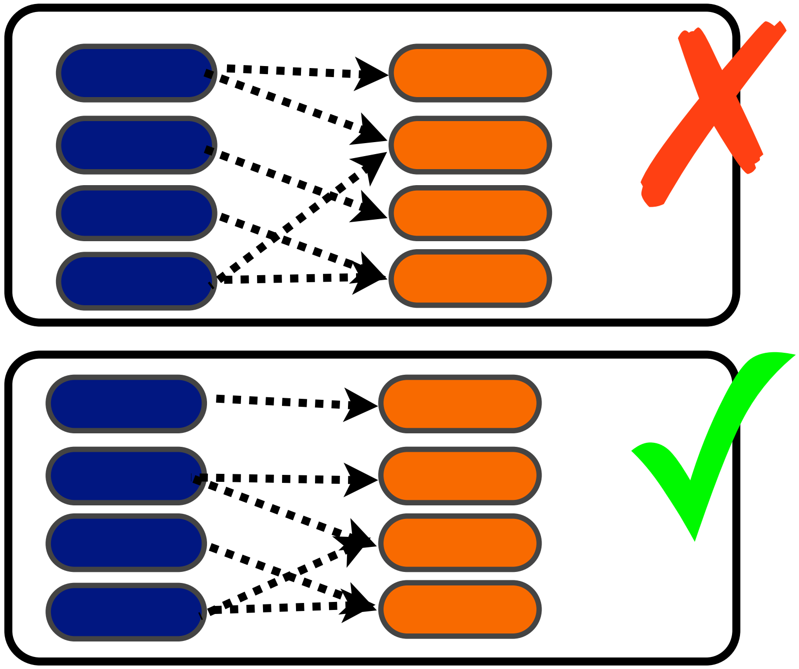
Starting with a bug report
or buggy behavior...



Bug #20677	Race condition betwe
Submitted:	24 Jun 2006 19:28
Reporter:	Kristian Nielsen
Status:	Verified
Category:	Server: ClusterRep
Version:	mysql-5.1
Assigned to:	
Tags:	5.1.12
Triage:	Triaged: D1 (Critical)

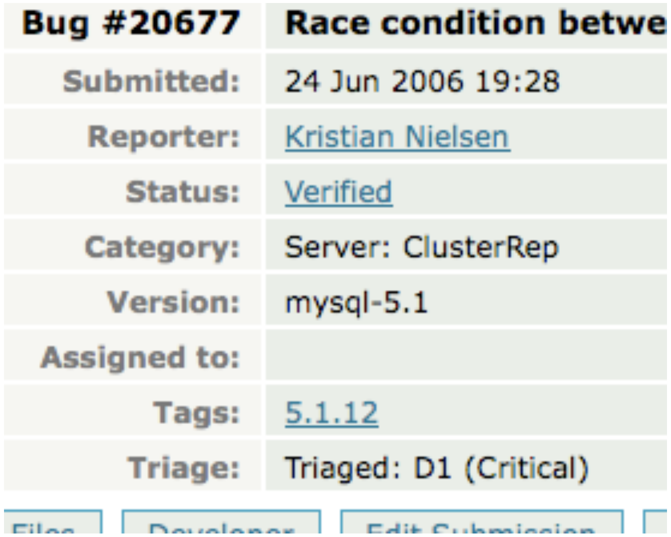
Files Developer Edit Submission

...collect graphs from
many runs, labeling as
buggy or correct



Labeled Graph Debugging

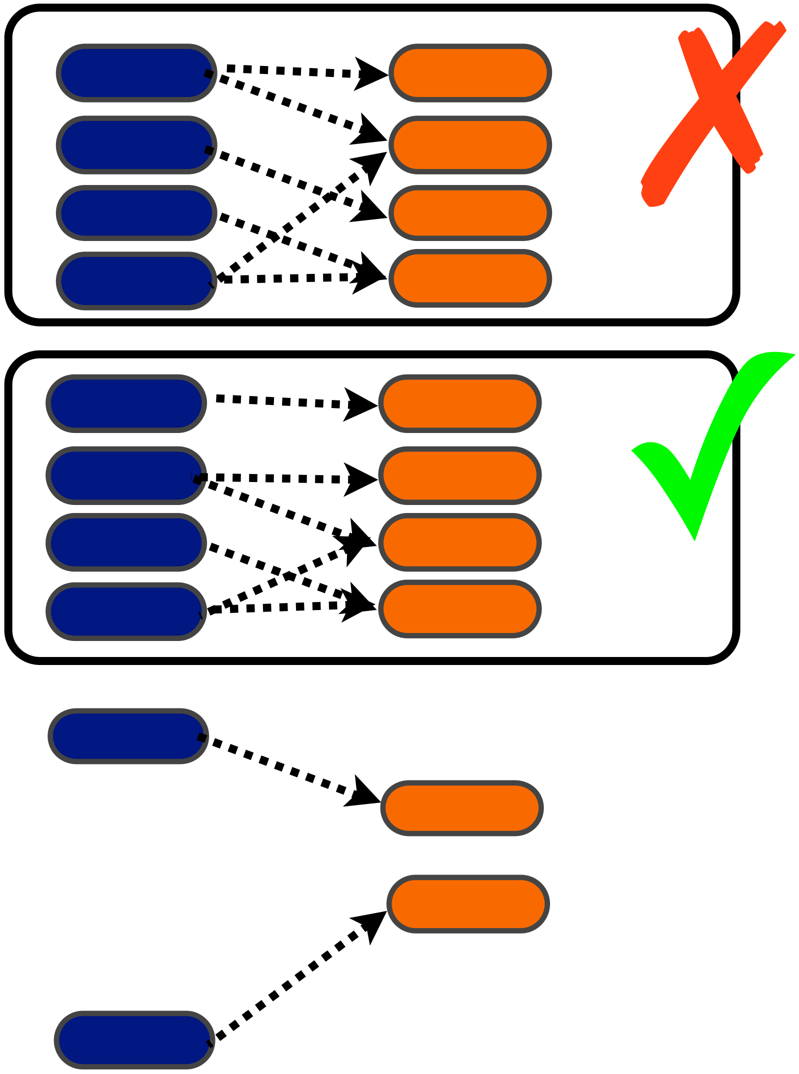
Starting with a bug report or buggy behavior...



Bug #20677	**Race condition betwe**
Submitted:	24 Jun 2006 19:28
Reporter:	Kristian Nielsen
Status:	Verified
Category:	Server: ClusterRep
Version:	mysql-5.1
Assigned to:	
Tags:	5.1.12
Triage:	Triaged: D1 (Critical)

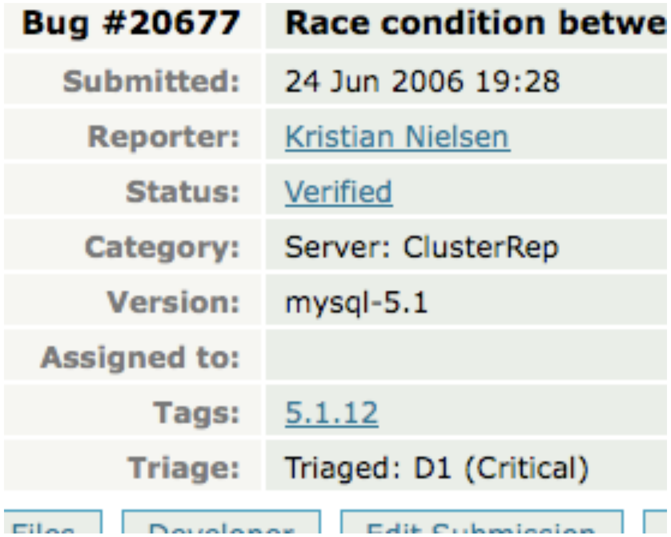
...collect graphs from many runs, labeling as buggy or correct

Find edges in any buggy graph, and in no correct graph



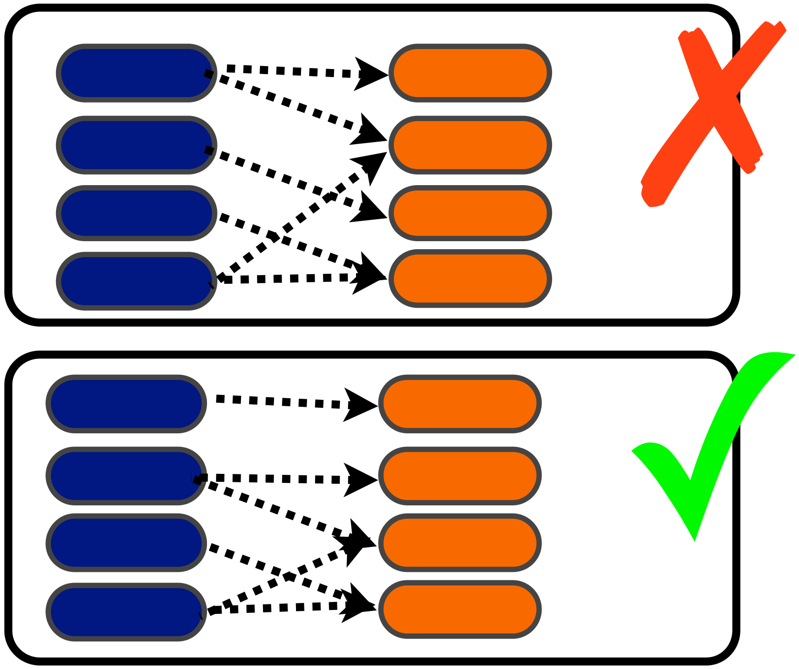
Labeled Graph Debugging

Starting with a bug report
or buggy behavior...



Bug #20677	Race condition betwe
Submitted:	24 Jun 2006 19:28
Reporter:	Kristian Nielsen
Status:	Verified
Category:	Server: ClusterRep
Version:	mysql-5.1
Assigned to:	
Tags:	5.1.12
Triage:	Triaged: D1 (Critical)

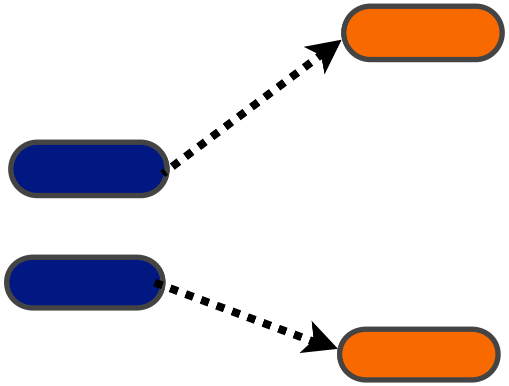
...collect graphs from
many runs, labeling as
buggy or correct



Find edges in any
buggy graph, and in
no correct graph

Rank the resulting edges, giving high rank to:

- Rare communication events
- Communication in a rare context



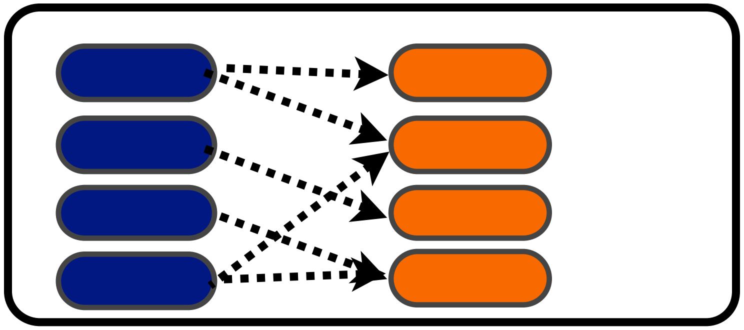
Anomaly-Based Bug Detection

The Bugs-As-Anomalies Hypothesis:
Programs usually work correctly, hence bugs are anomalies

Anomaly-Based Bug Detection

The Bugs-As-Anomalies Hypothesis:
Programs usually work correctly, hence bugs are anomalies

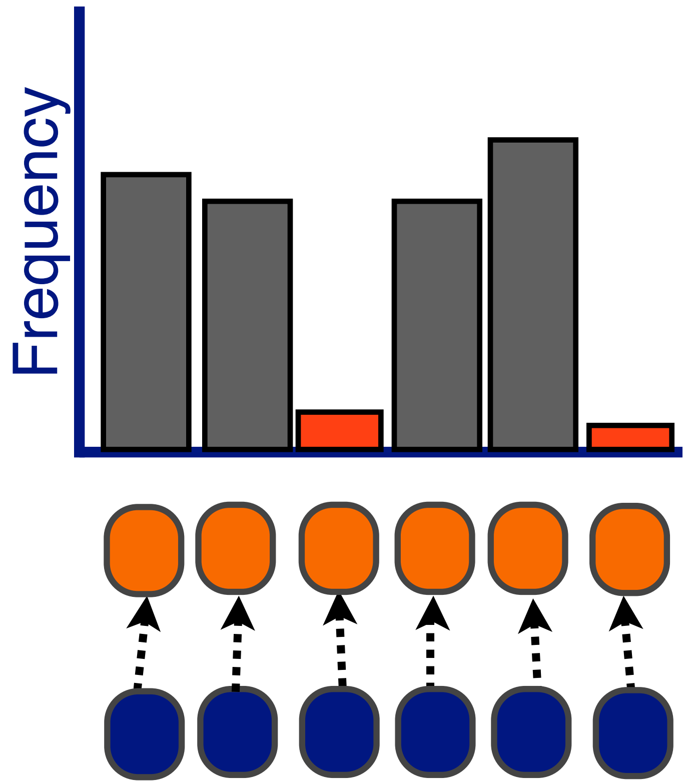
By looking for anomalies,
we are likely to find bugs



Anomaly-Based Bug Detection

The Bugs-As-Anomalies Hypothesis:
Programs usually work correctly, hence bugs are anomalies

By looking for anomalies,
we are likely to find bugs

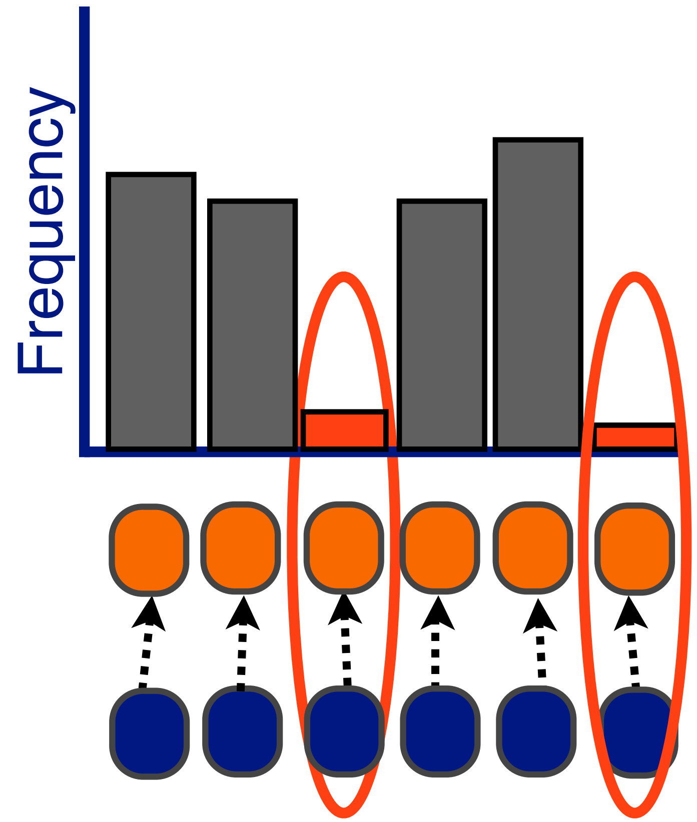


Anomaly-Based Bug Detection

The Bugs-As-Anomalies Hypothesis:
Programs usually work correctly, hence bugs are anomalies

By looking for anomalies,
we are likely to find bugs

Likely bugs are low-frequency
communication events



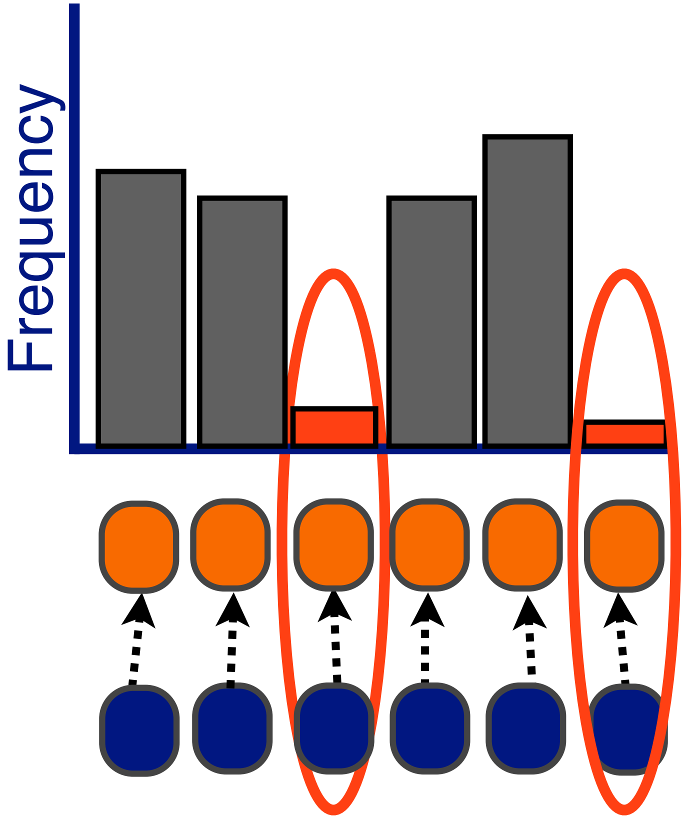
Anomaly-Based Bug Detection

The Bugs-As-Anomalies Hypothesis:
Programs usually work correctly, hence bugs are anomalies

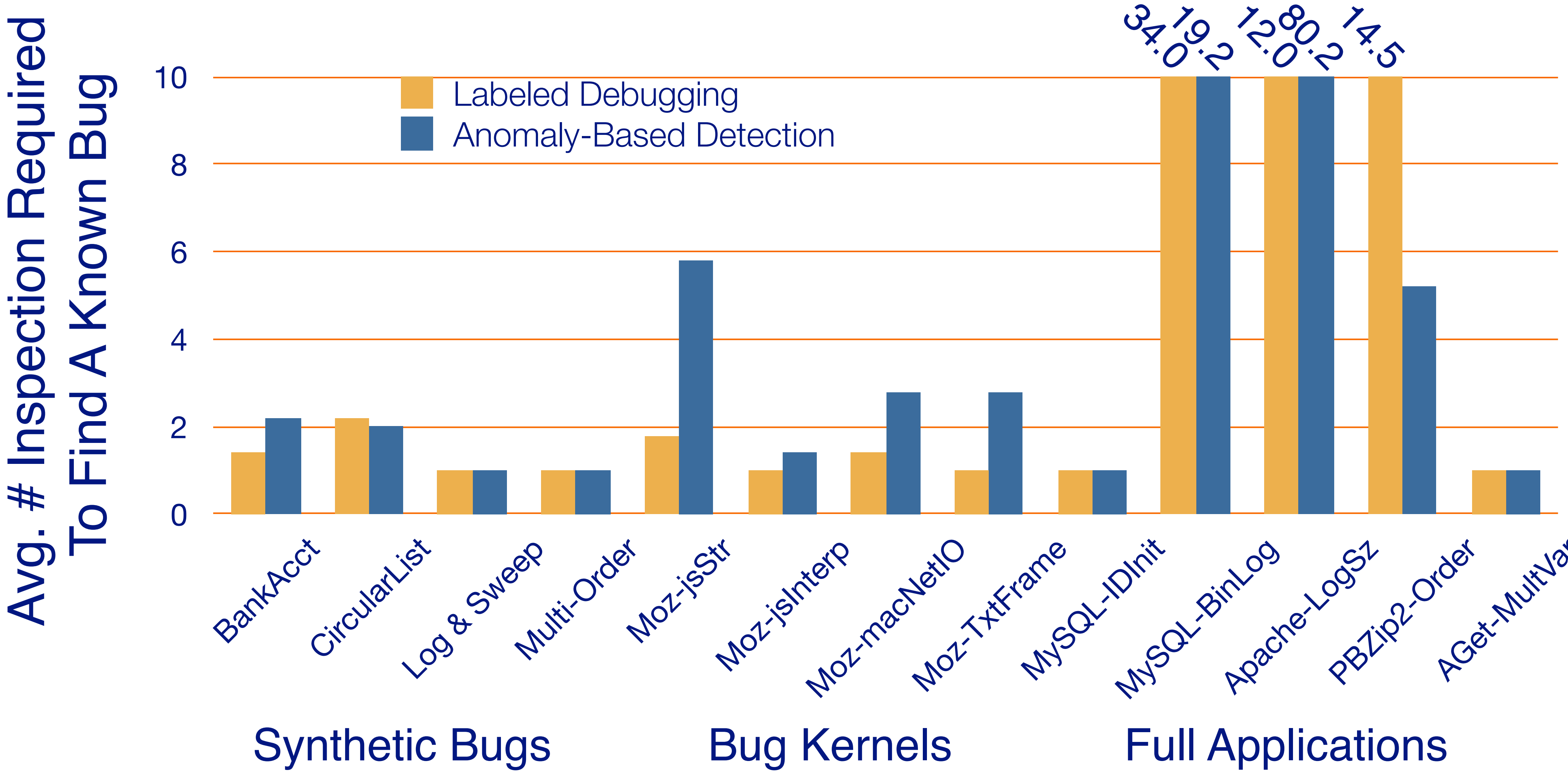
By looking for anomalies,
we are likely to find bugs

Likely bugs are low-frequency communication events

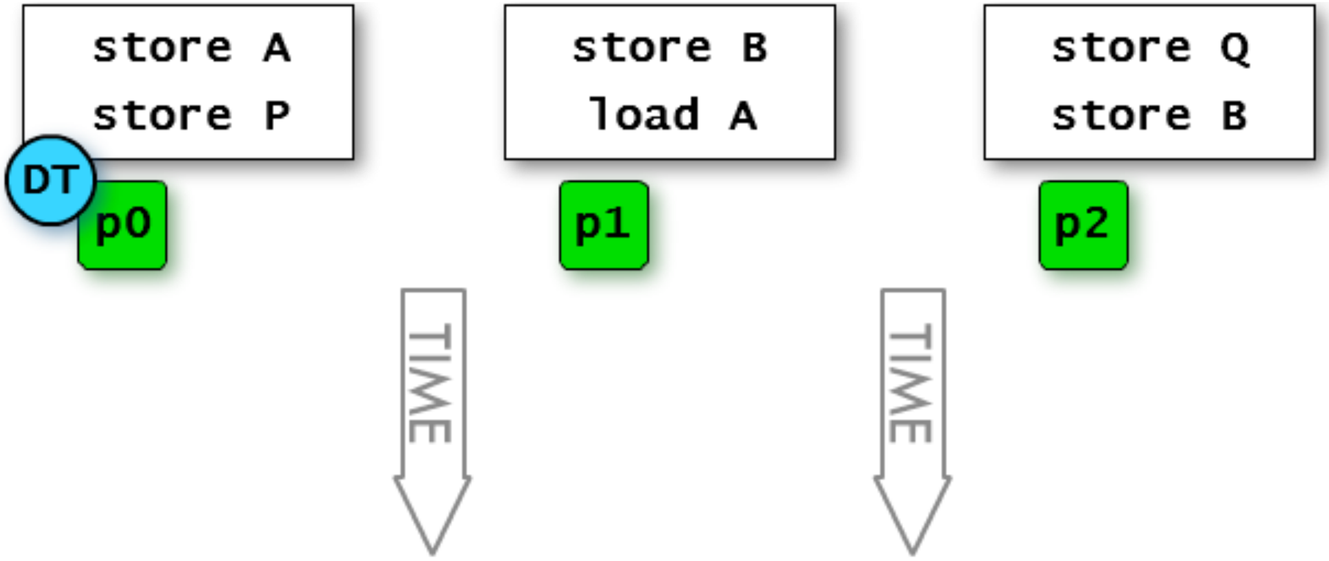
Fully Automatic Detection - No labeling required



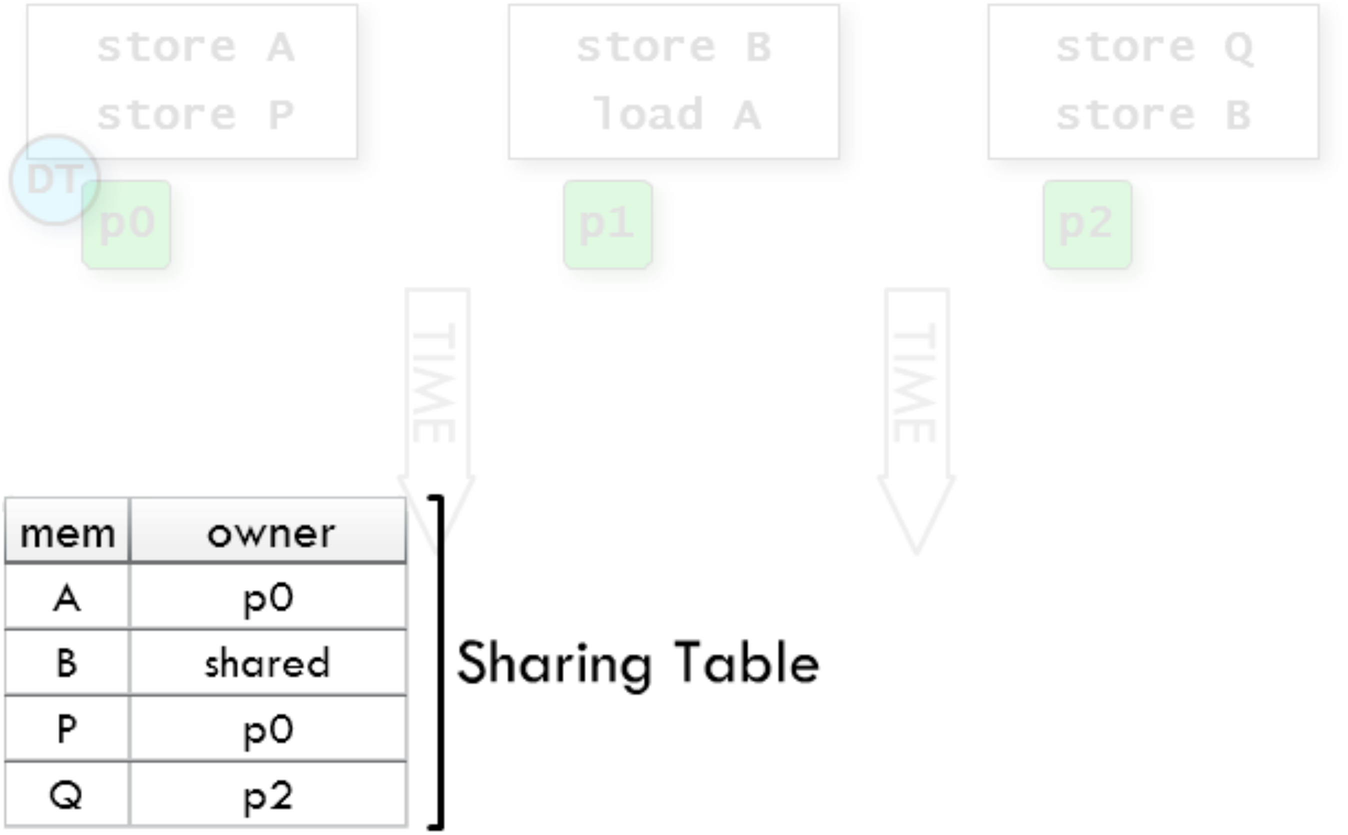
Bug Detection Capability



DMP-Ownership Example



DMP-Ownership Example



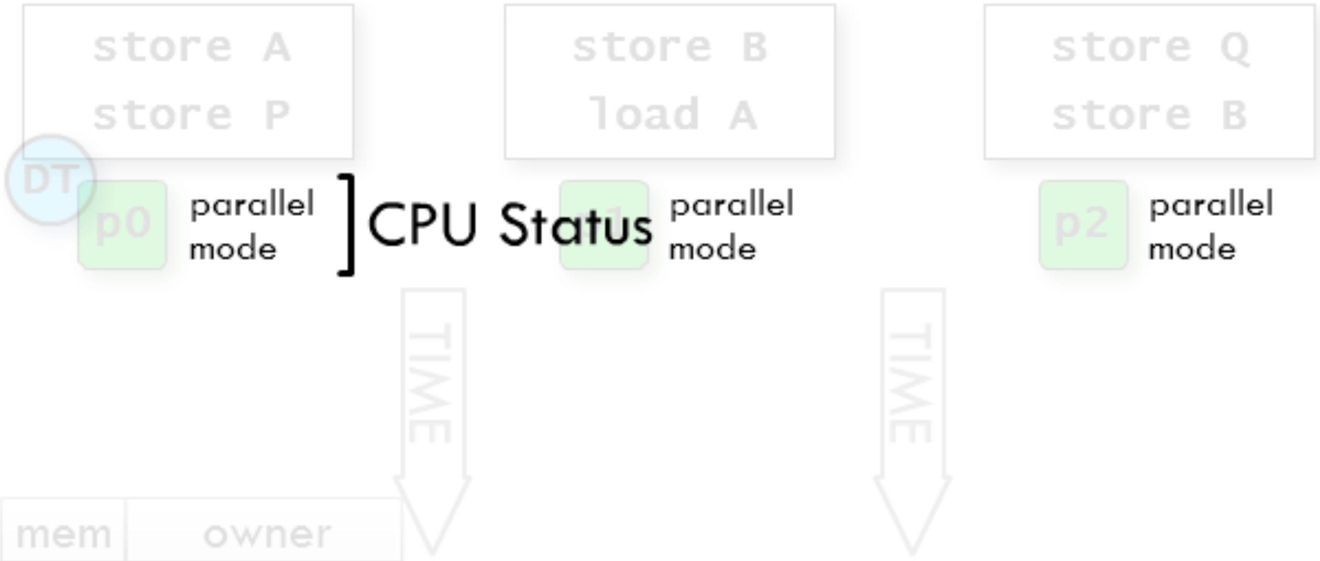
```
store A
store P
```

```
store B
load A
```

```
store Q
store B
```

mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



store A
store P

store B
load A

store Q
store B

DT

p0 parallel mode

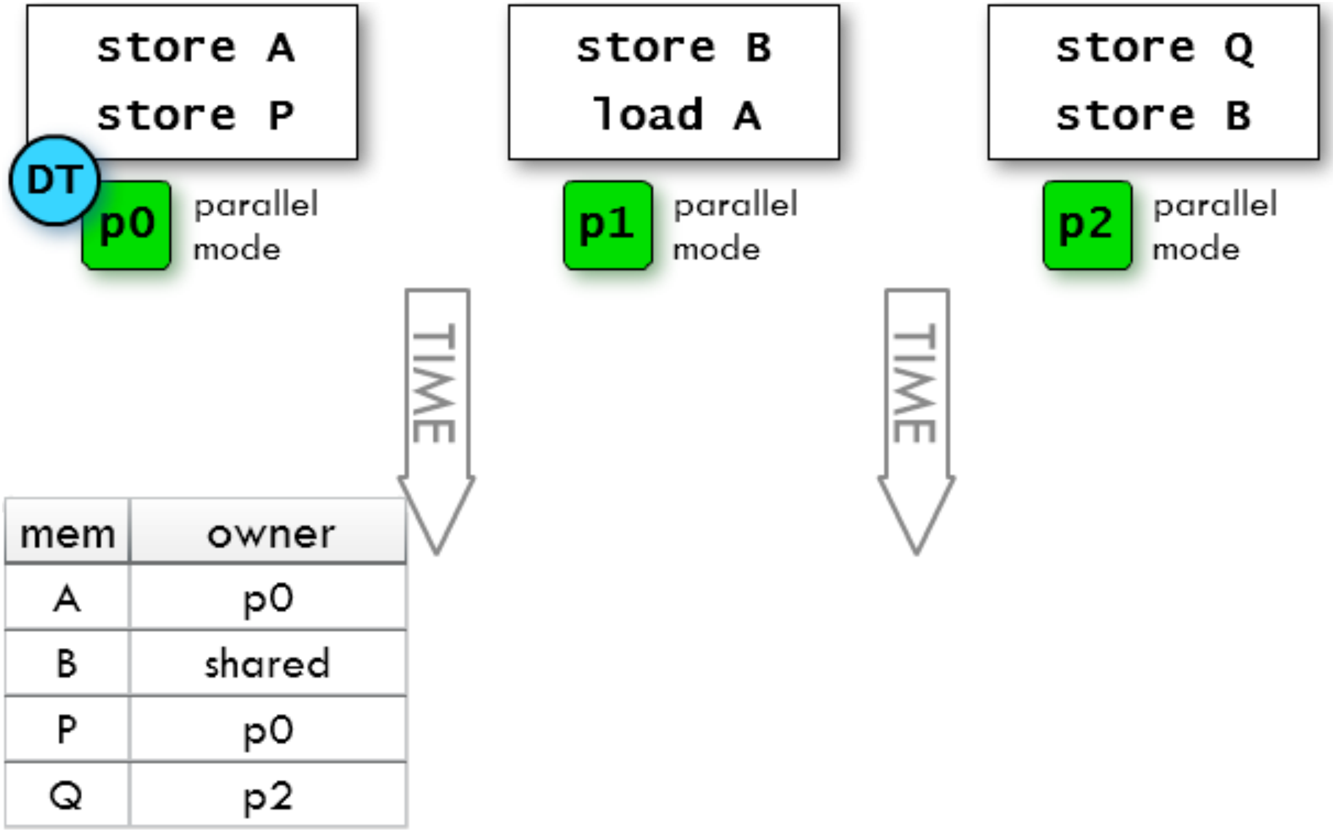
CPU Status

parallel mode

p2 parallel mode

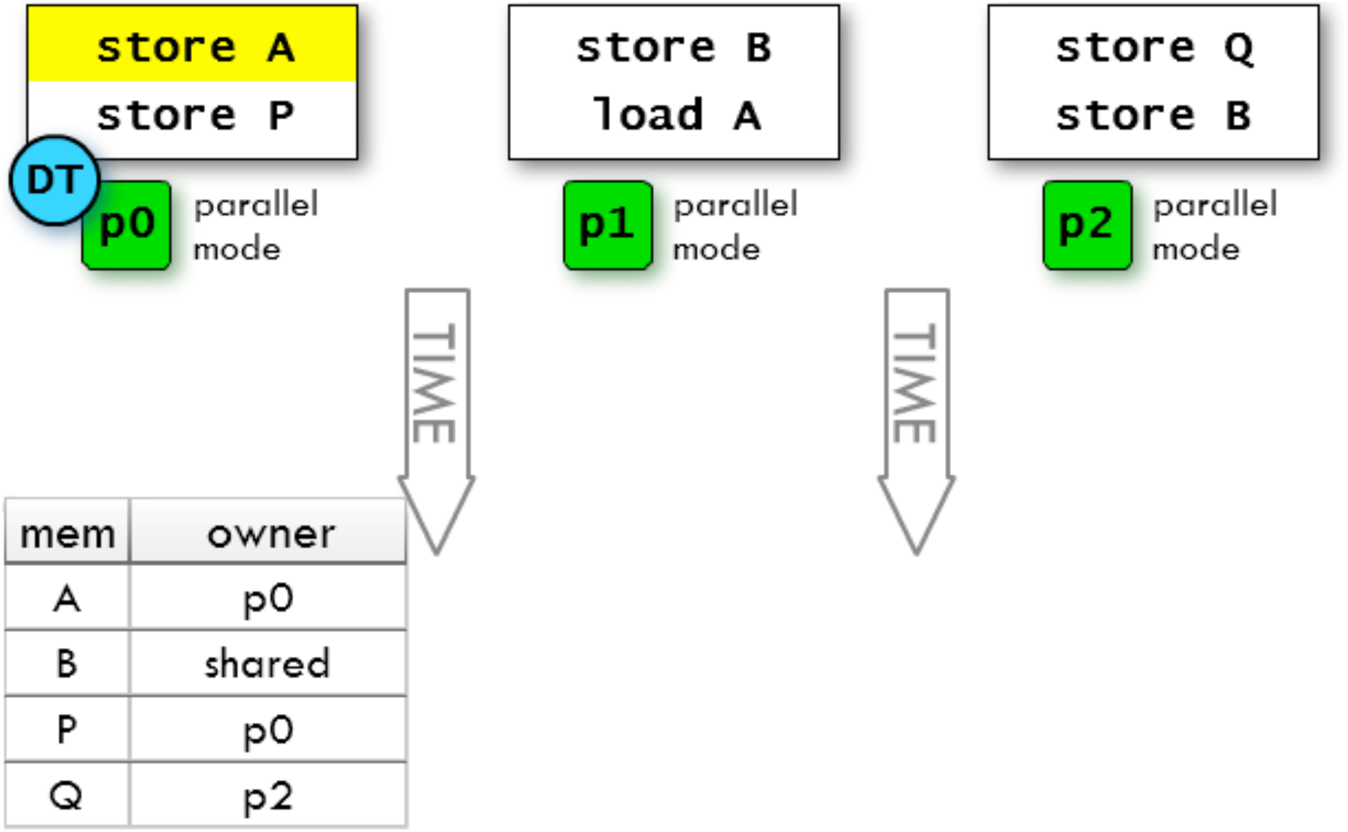
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



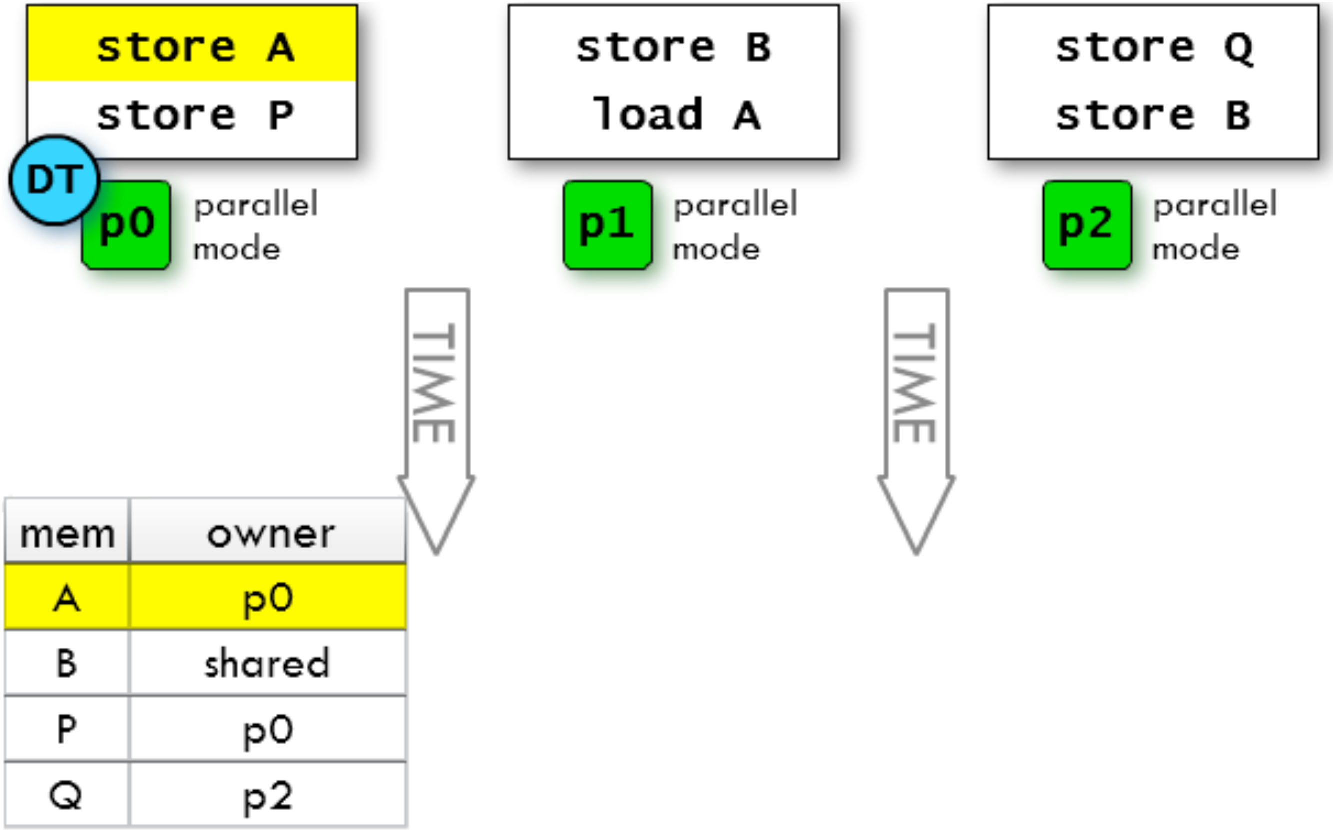
```
store A
store P
```

```
store B
load A
```

```
store Q
store B
```

mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



```
store A
store P
```

DT

p0 parallel mode

```
store B
load A
```

p1 parallel mode

```
store Q
store B
```

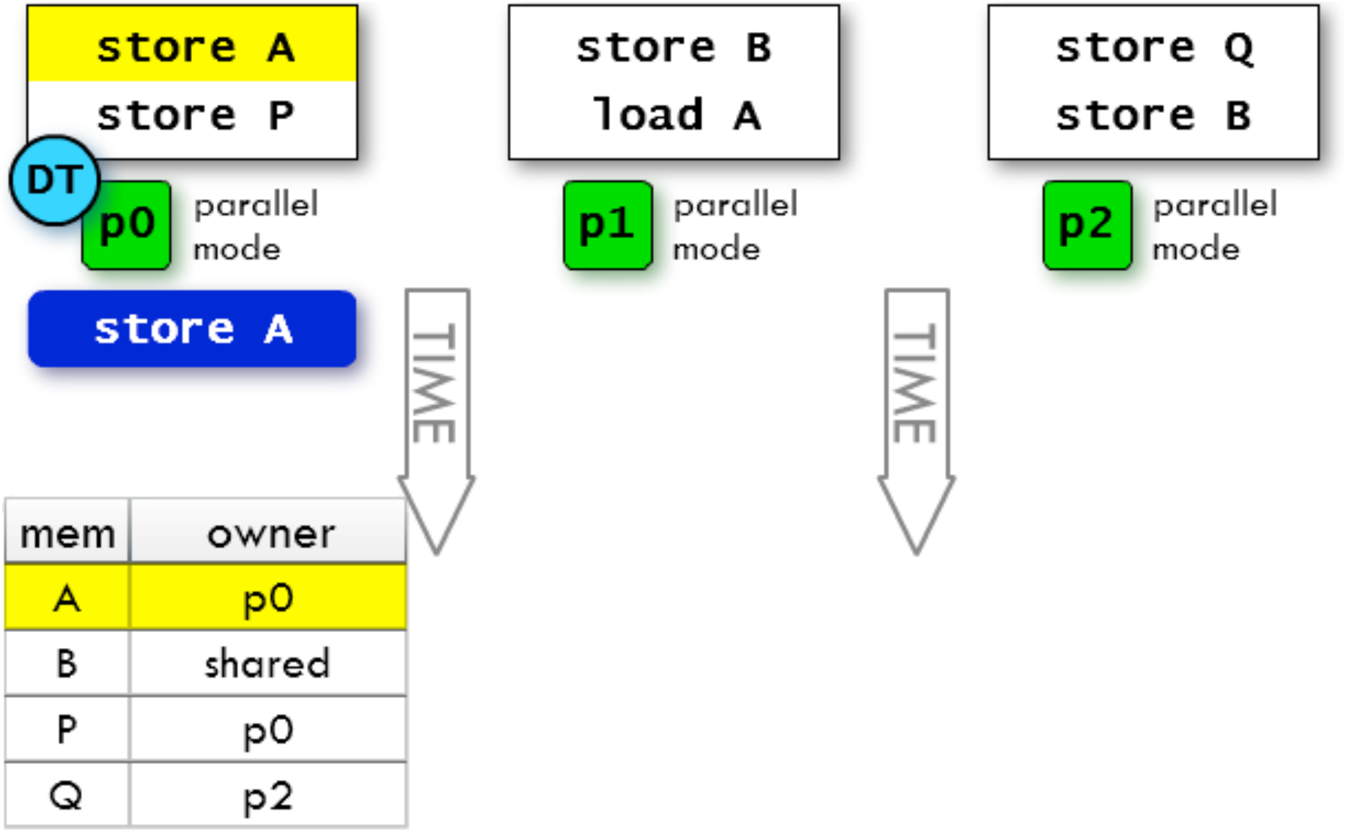
p2 parallel mode

TIME

TIME

mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



```
store A
store P
```

p0 parallel mode

```
store A
```

```
store B
load A
```

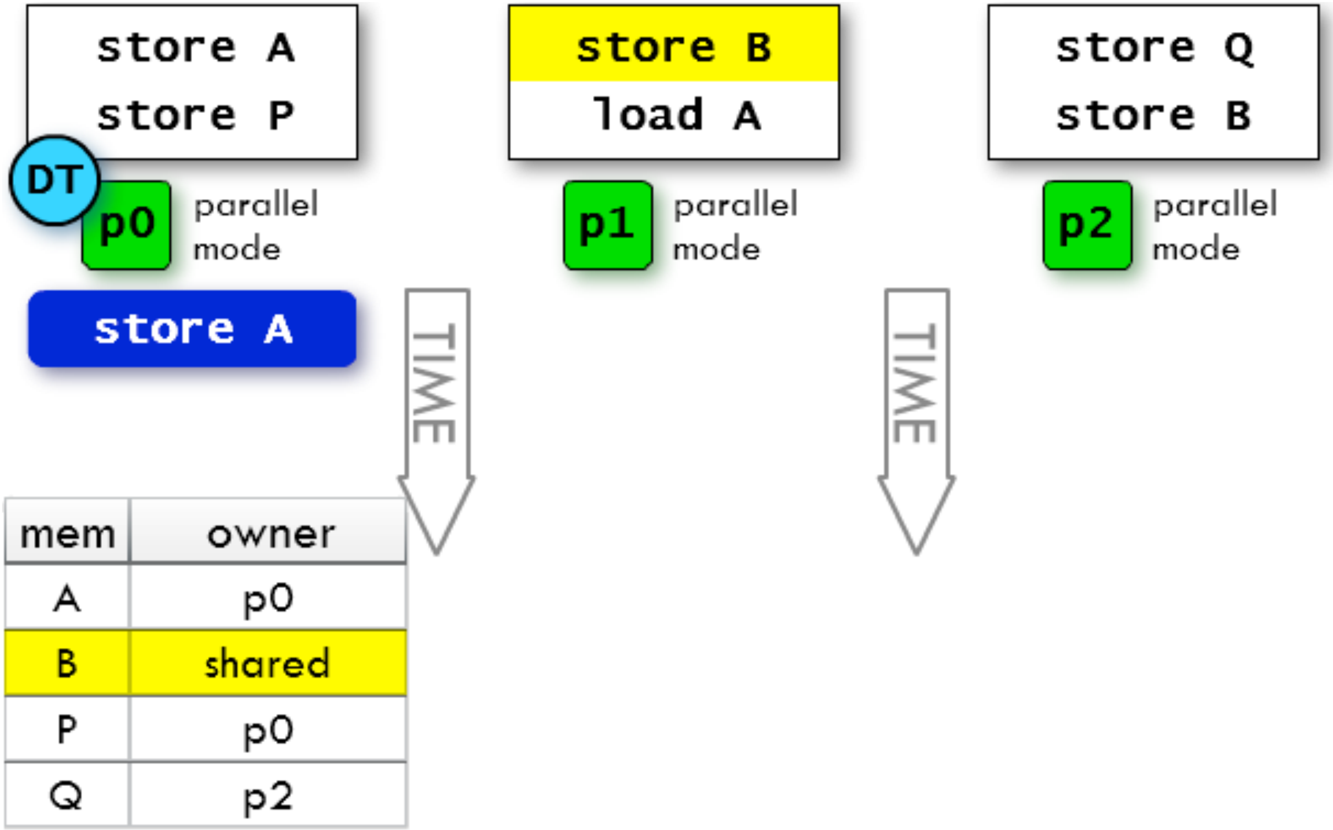
p1 parallel mode

```
store Q
store B
```

p2 parallel mode

mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



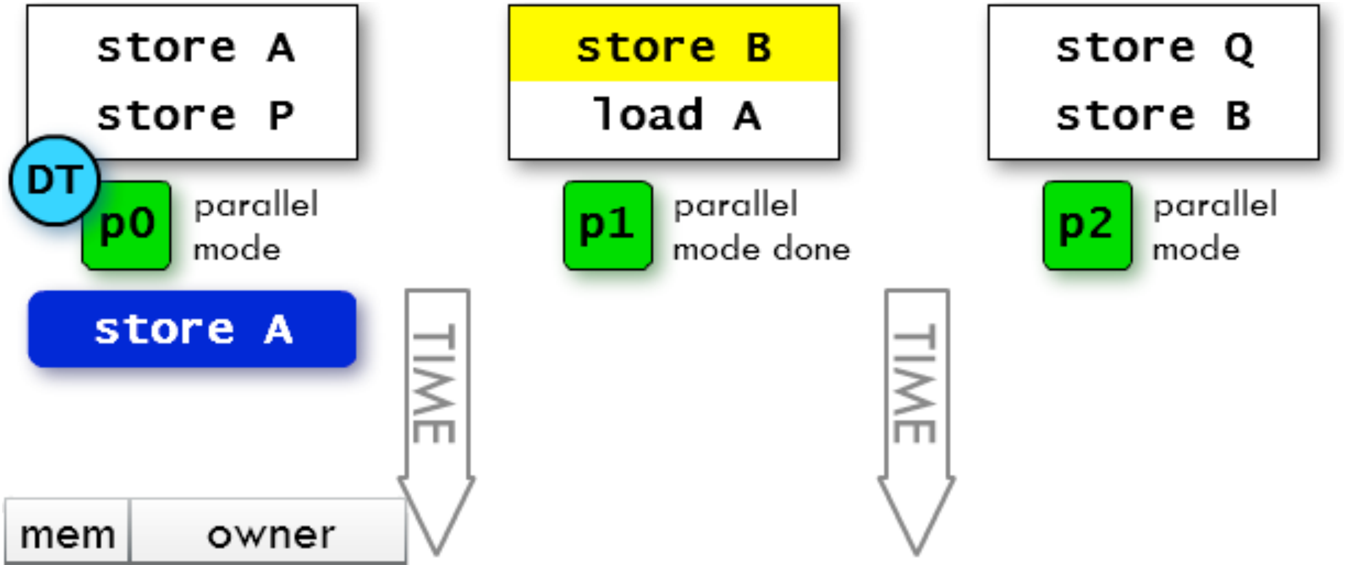
```
store A
store P
```

```
store B
load A
```

```
store Q
store B
```

mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



store A
store P

DT

p0 parallel mode

store A

TIME

store B
load A

p1 parallel mode done

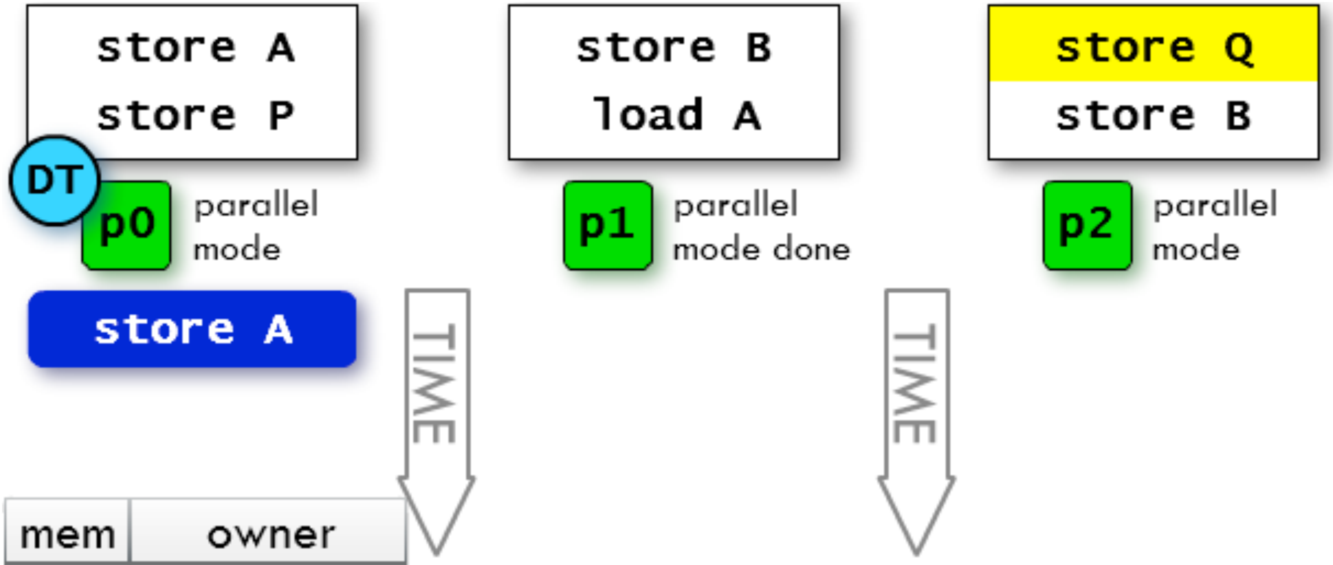
TIME

store Q
store B

p2 parallel mode

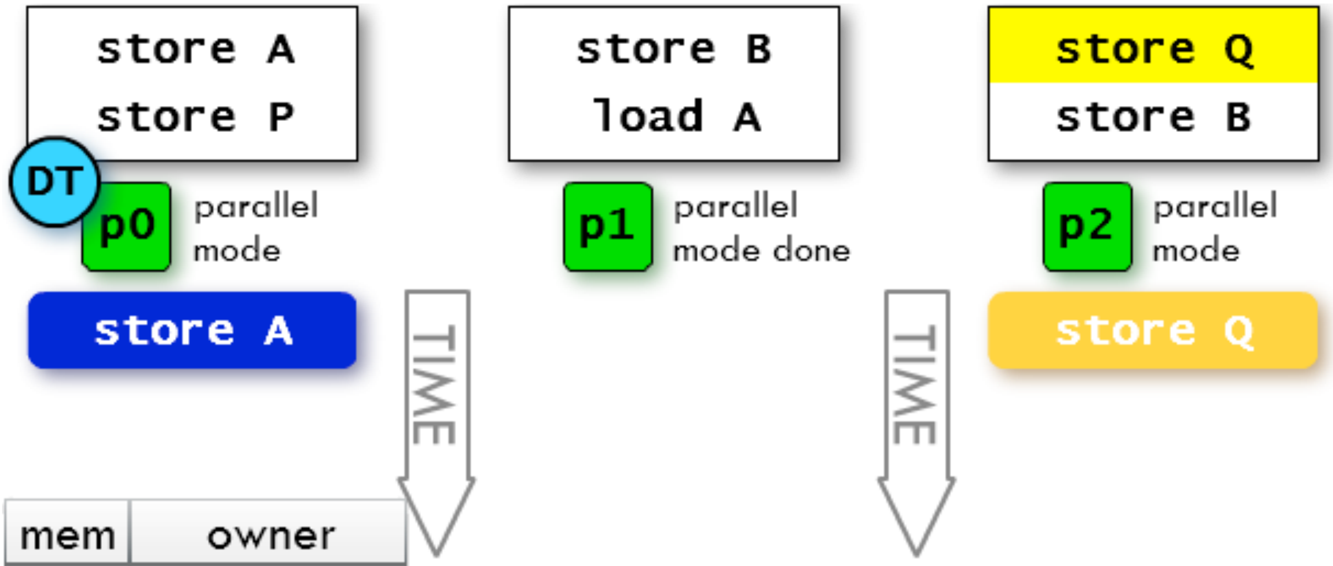
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



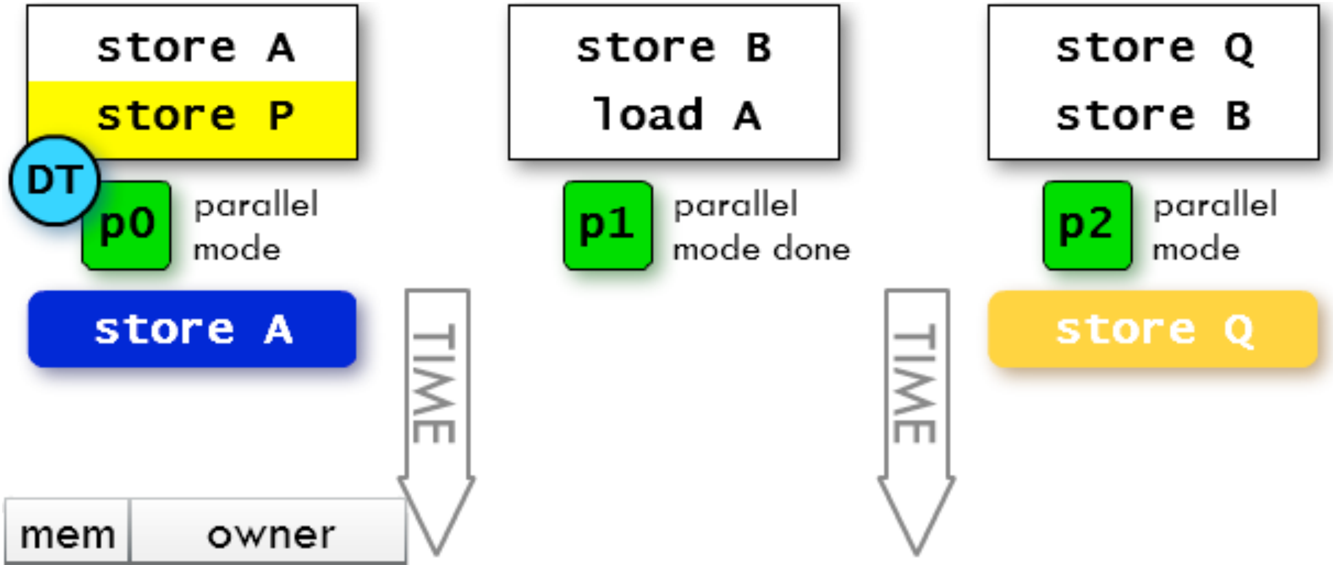
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



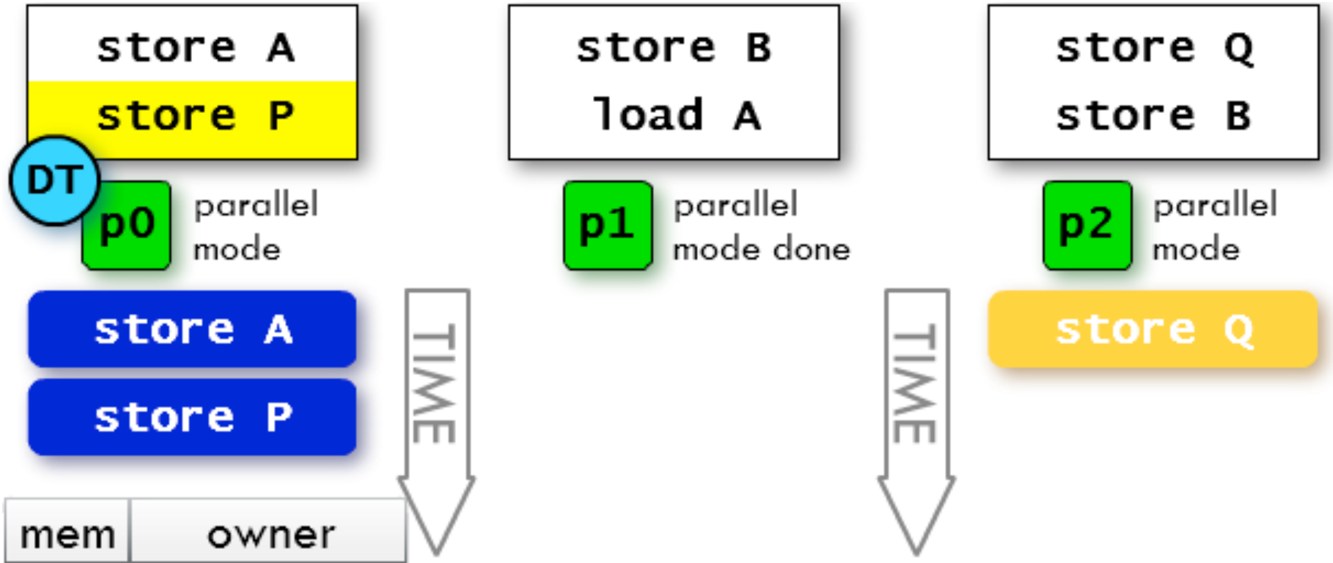
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



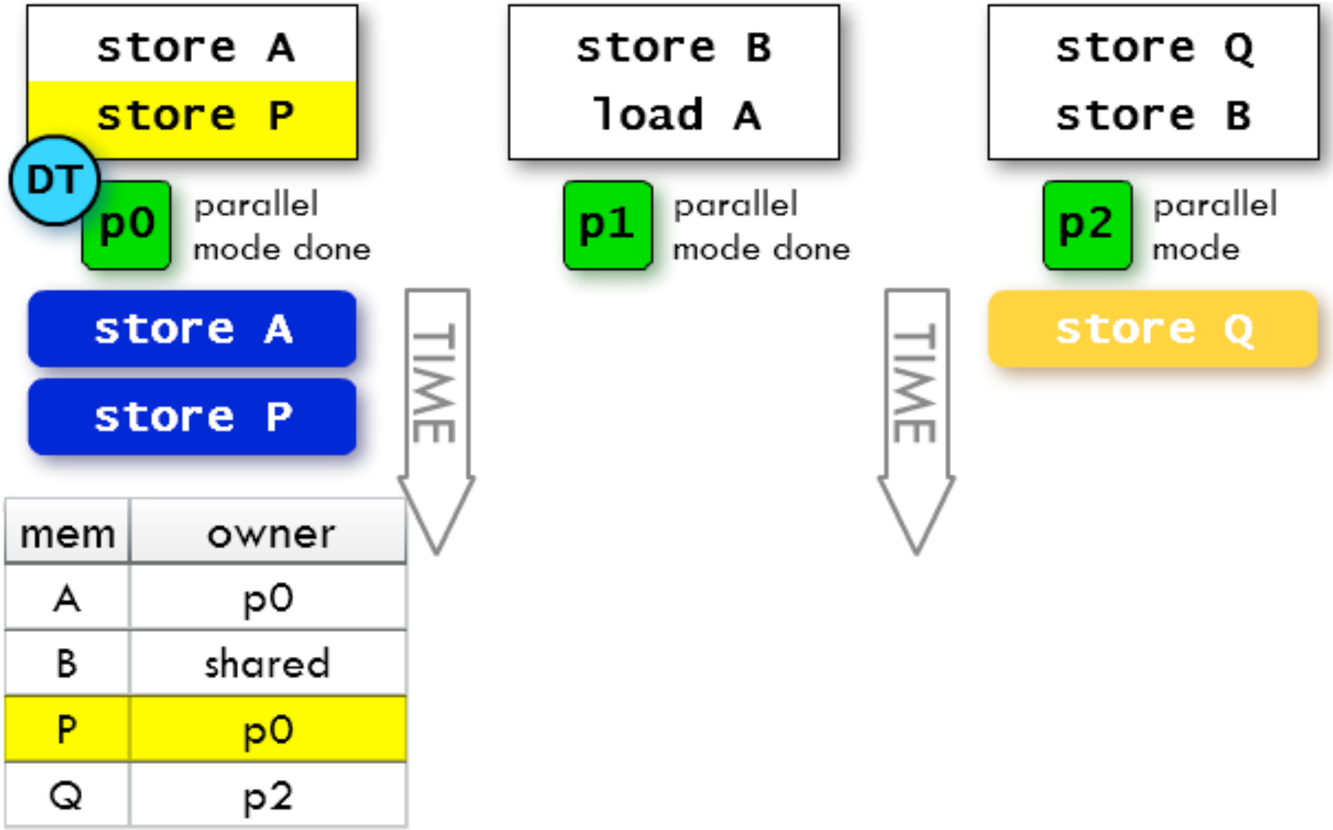
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



store A
store P

DT p0 parallel mode done

store A
store P

store B
load A

p1 parallel mode done

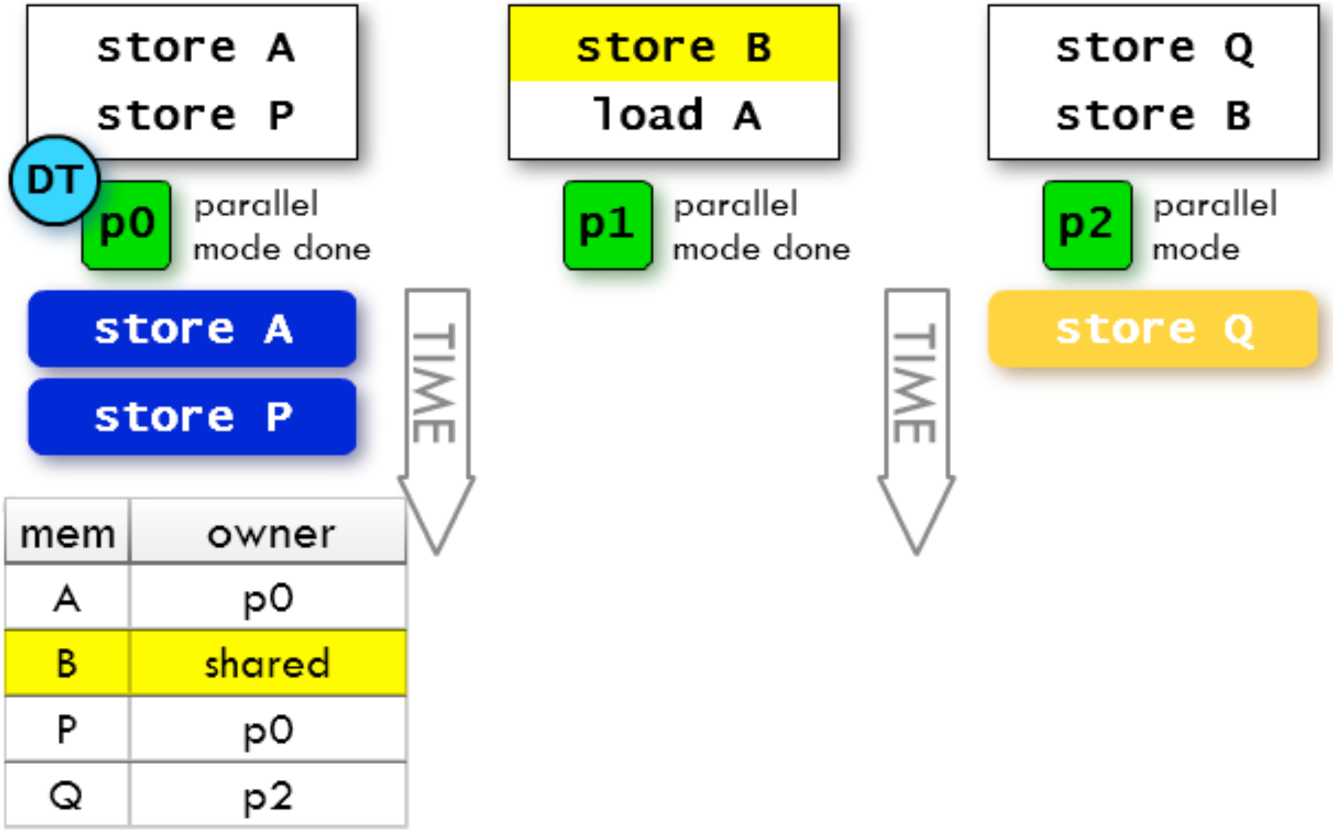
store Q
store B

p2 parallel mode

store Q

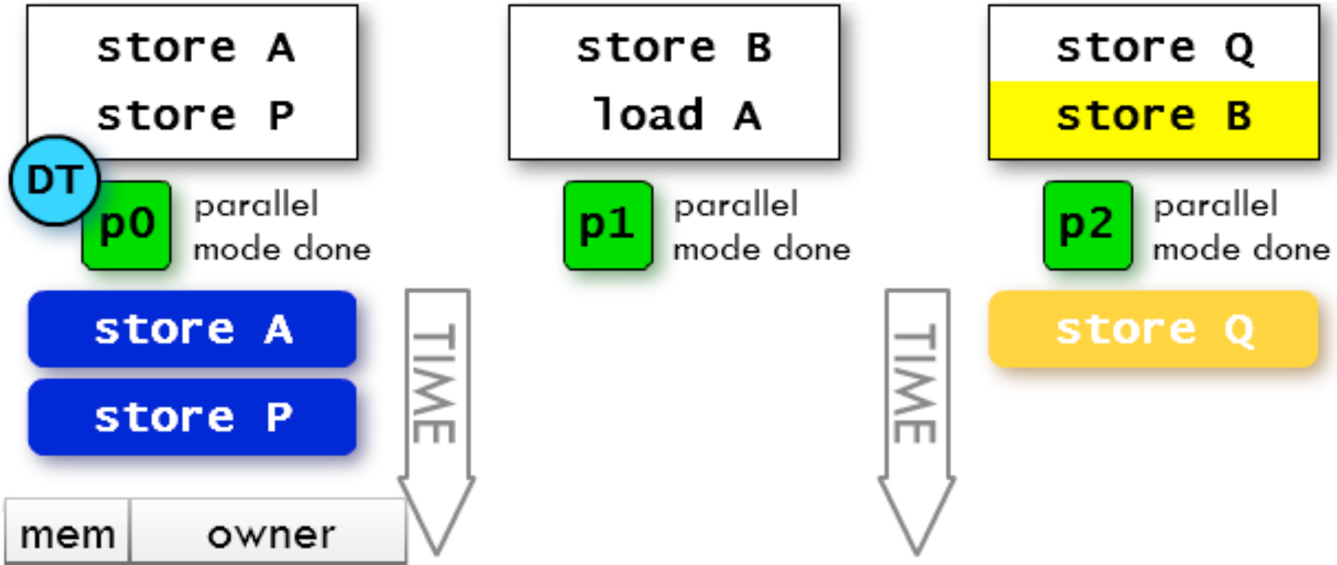
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



store A
store P

DT

p0 parallel mode done

store A

store P

store B
load A

p1 parallel mode done

store Q
store B

p2 parallel mode done

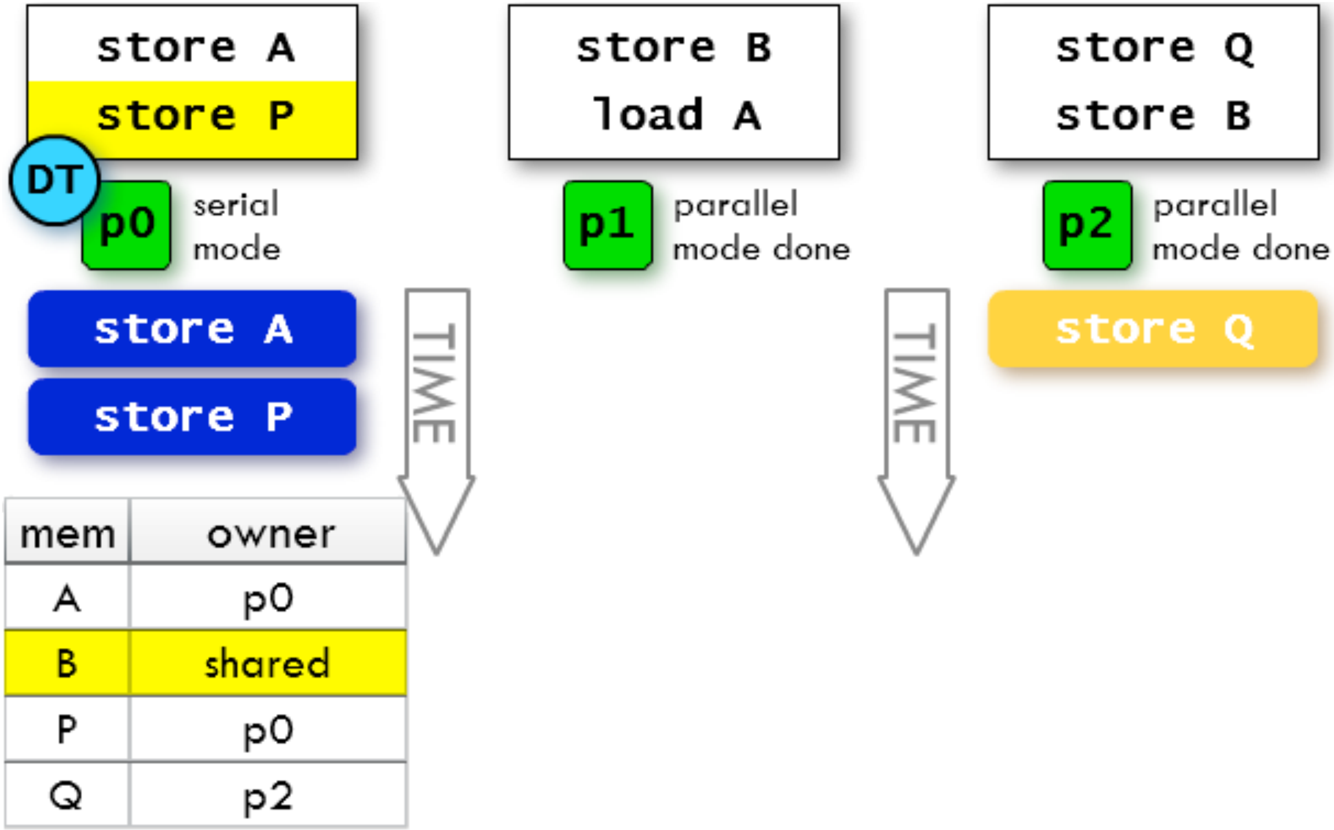
store Q

TIME

TIME

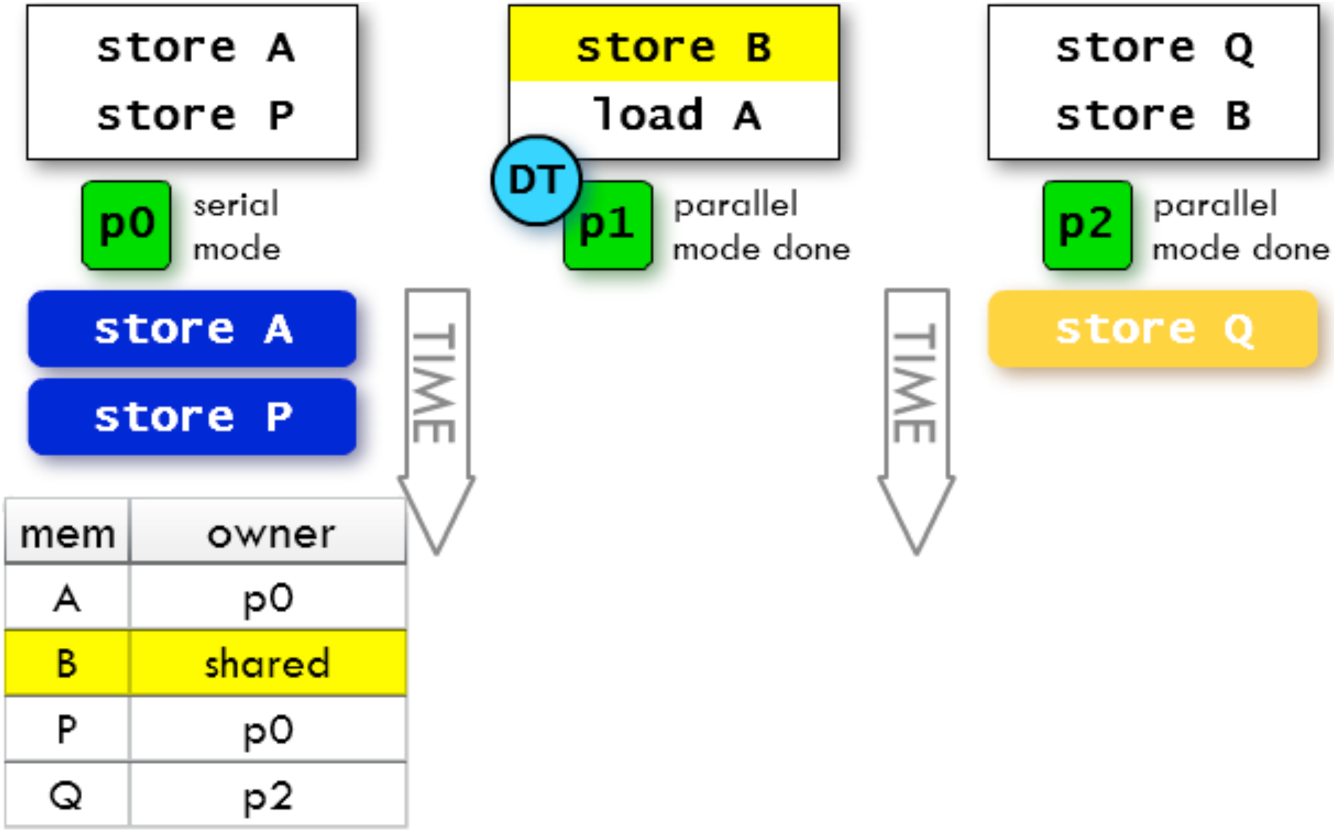
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



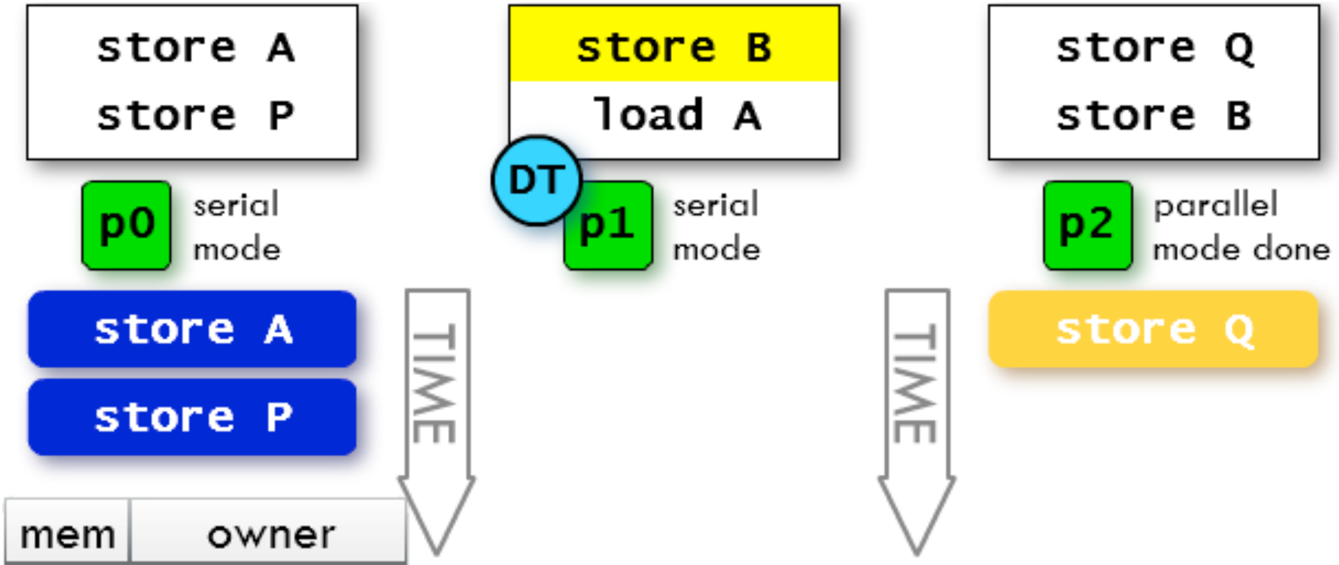
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



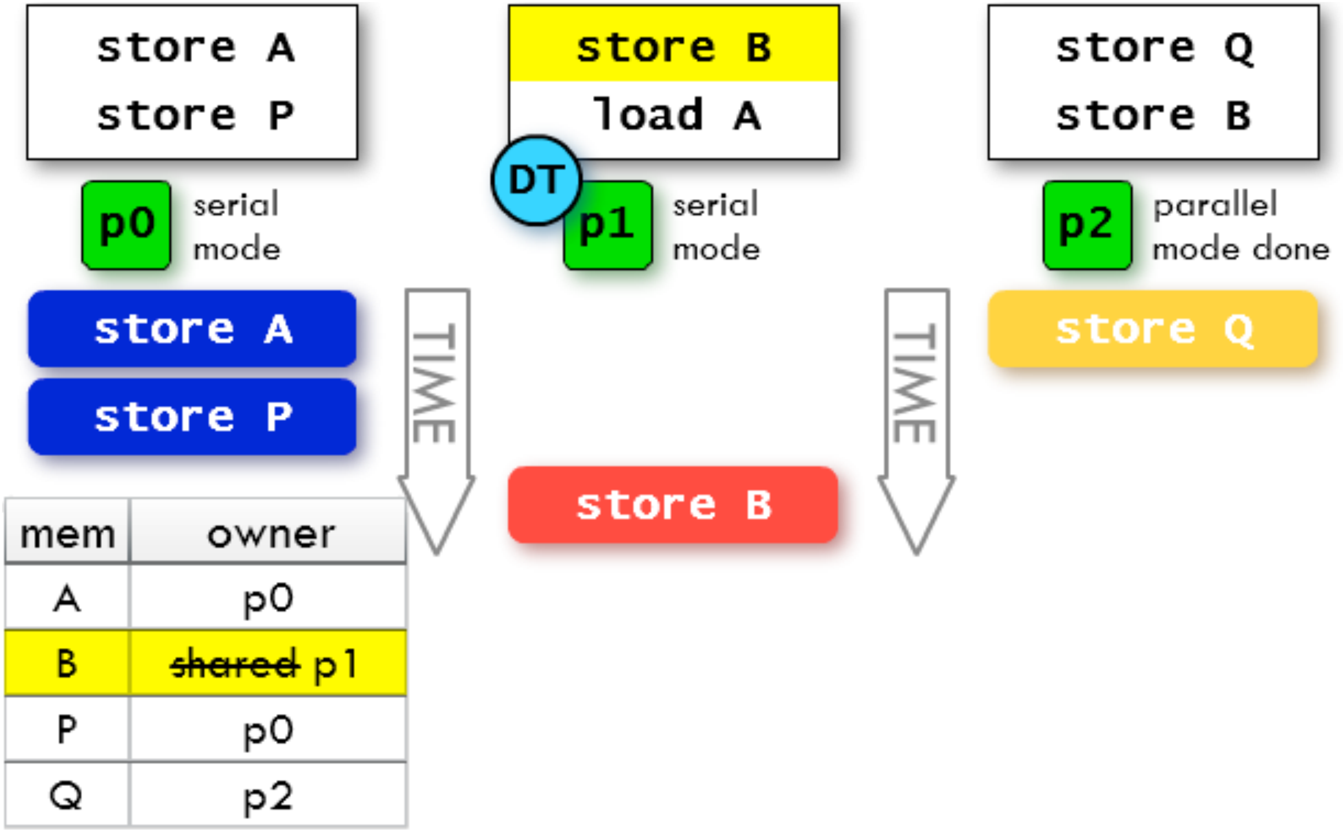
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



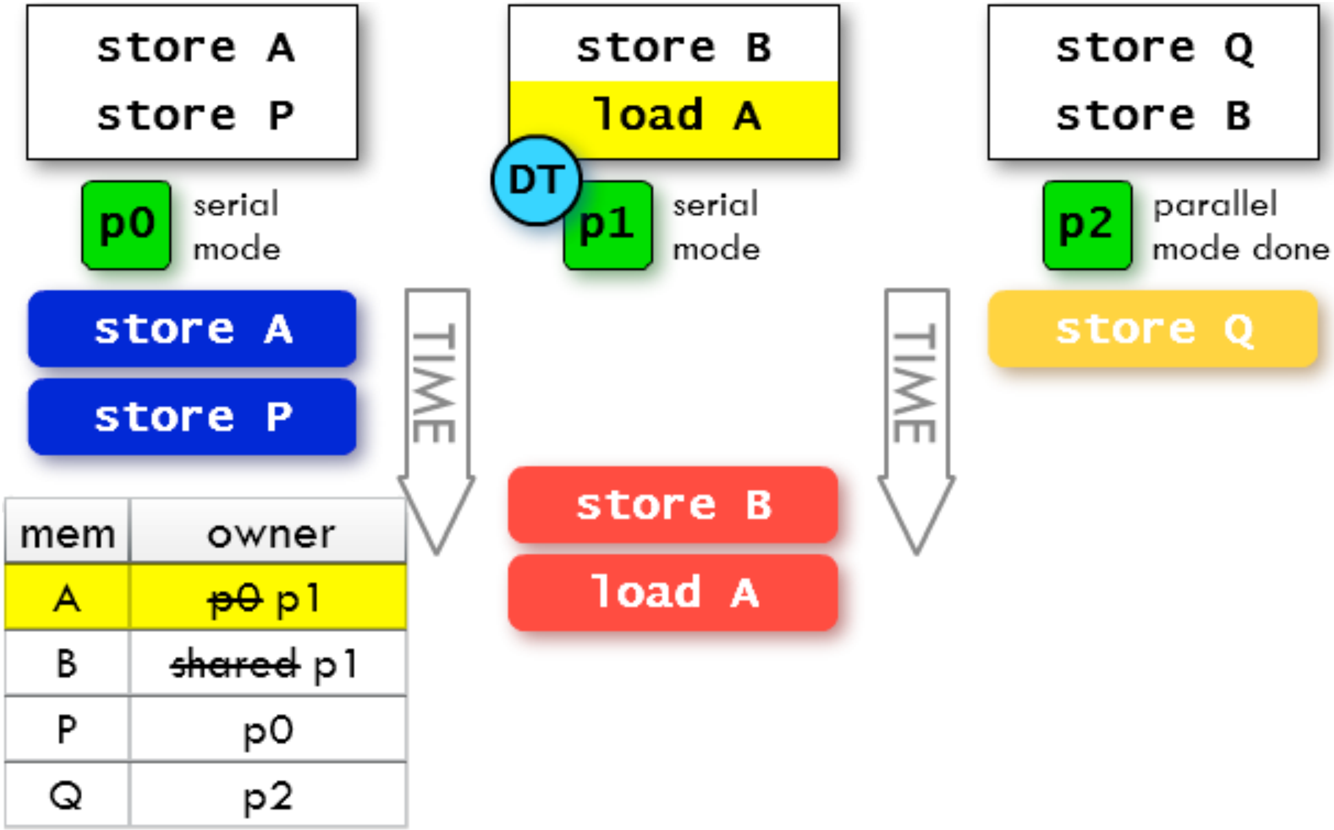
mem	owner
A	p0
B	shared
P	p0
Q	p2

DMP-Ownership Example



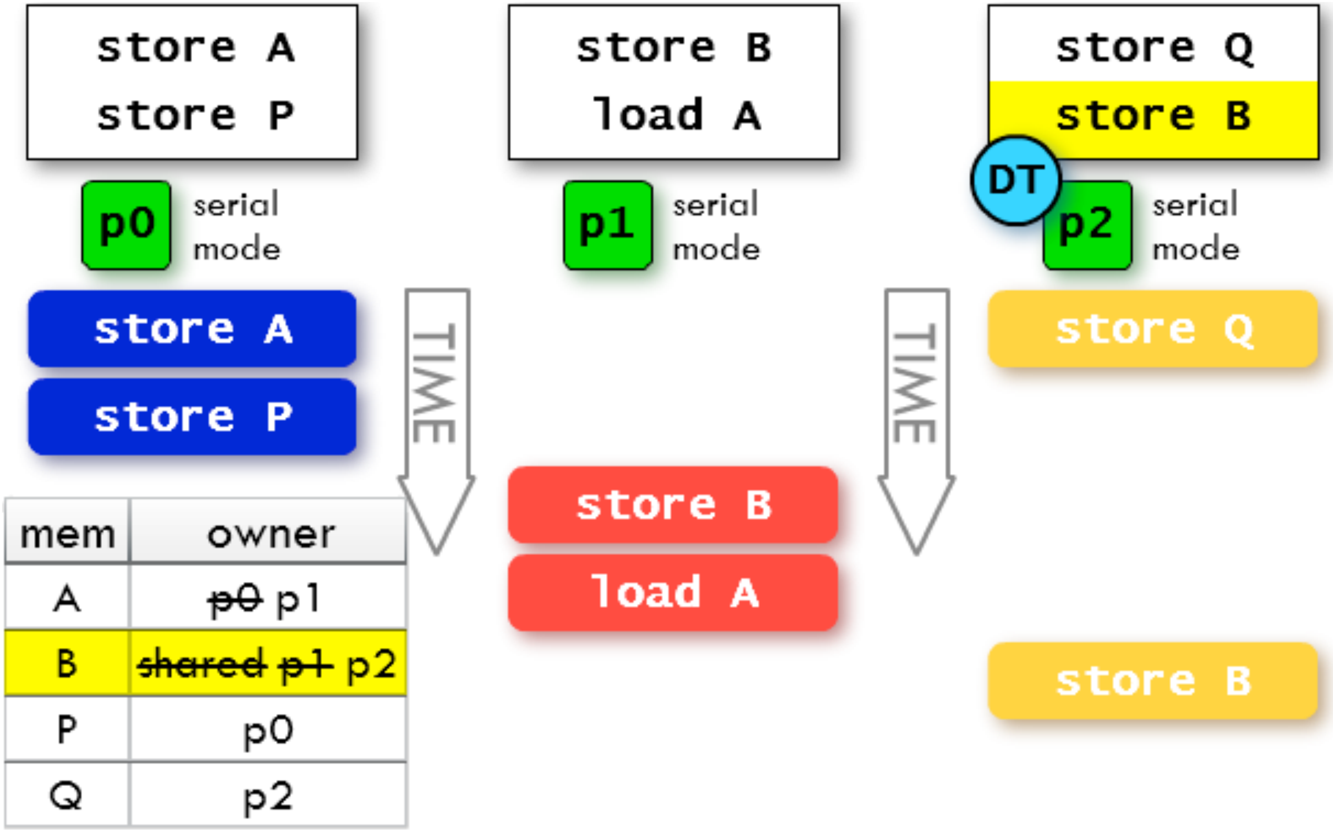
mem	owner
A	p0
B	~~shared~~ p1
P	p0
Q	p2

DMP-Ownership Example



mem	owner
A	~~p0~~ p1
B	~~shared~~ p1
P	p0
Q	p2

DMP-Ownership Example



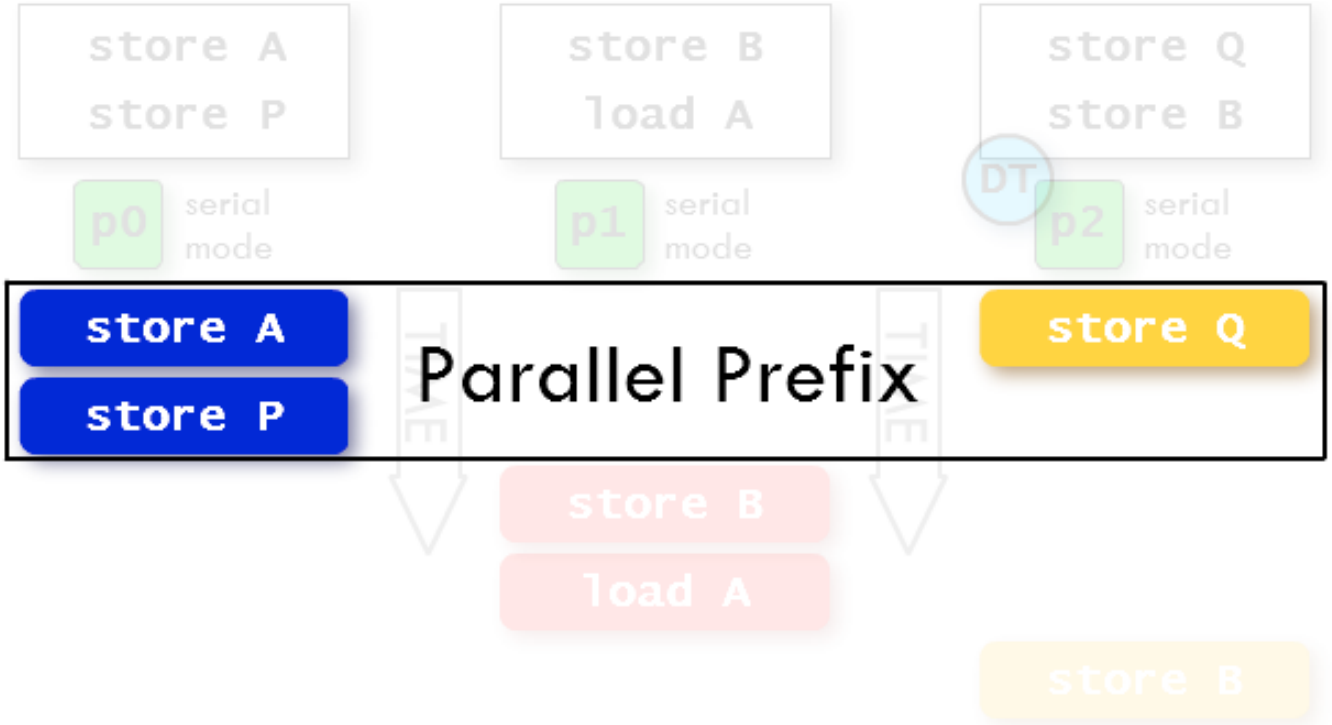
```
store A
store P
```

```
store B
load A
```

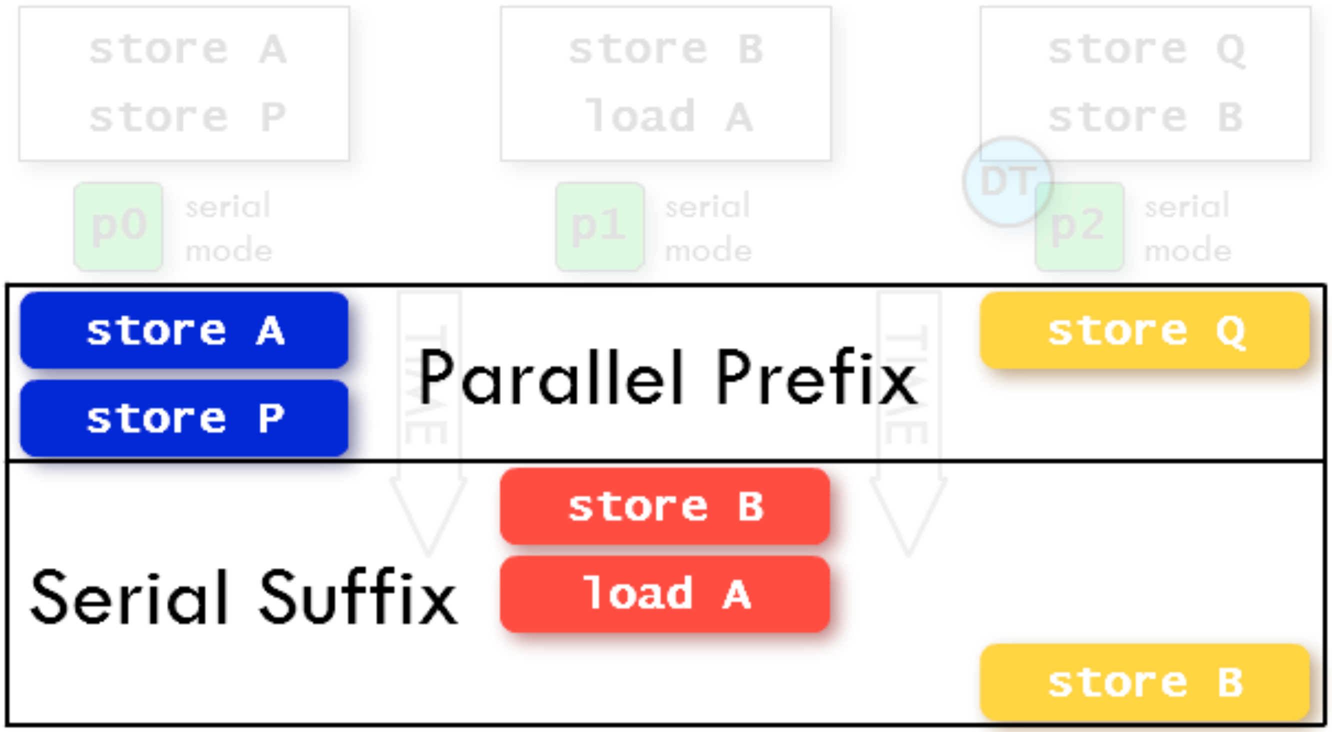
```
store Q
store B
```

mem	owner
A	~~p0~~ p1
B	~~shared~~ ~~p1~~ p2
P	p0
Q	p2

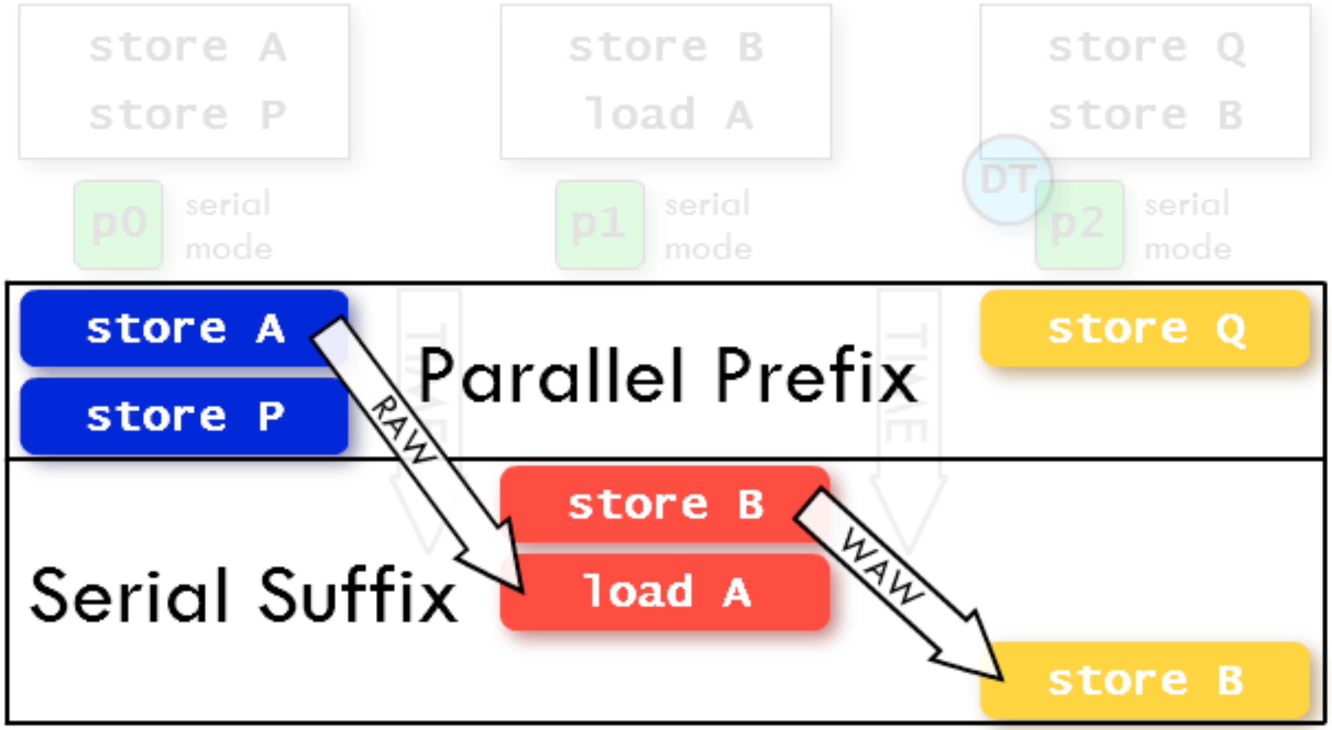
DMP-Ownership Example



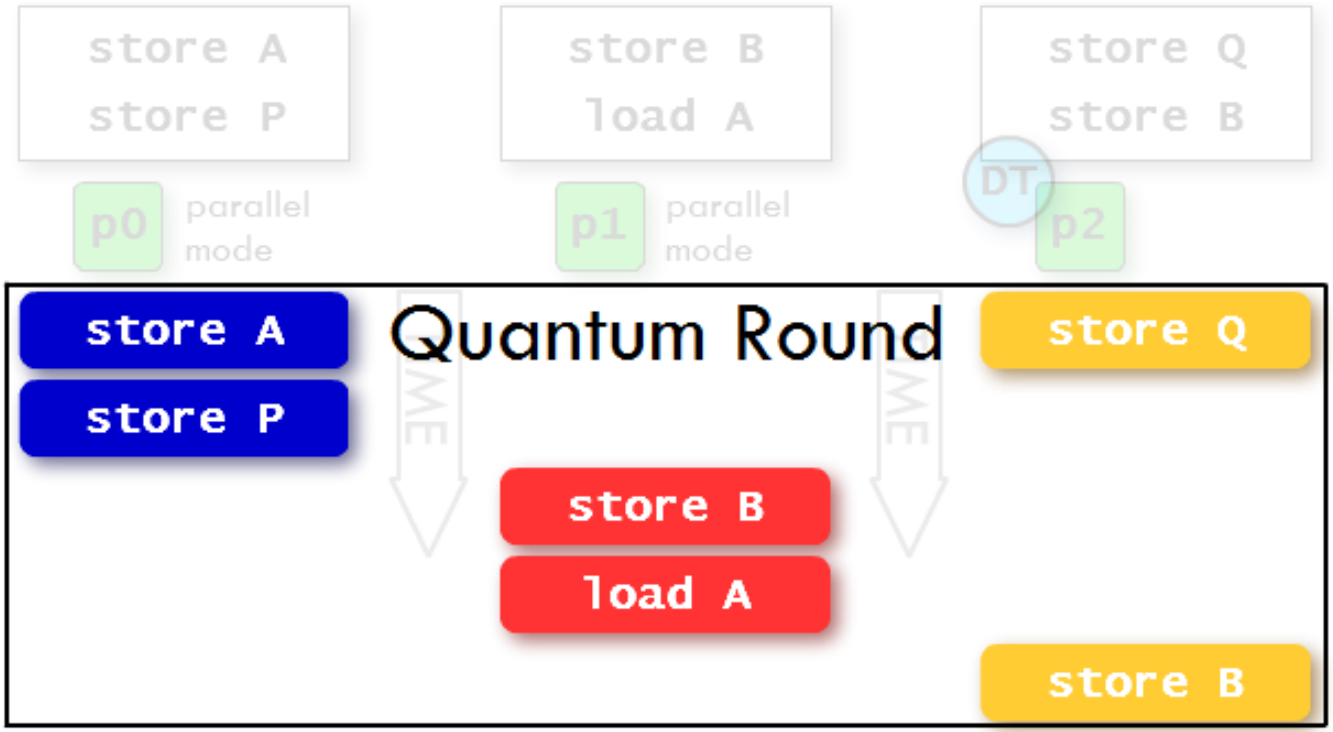
DMP-Ownership Example



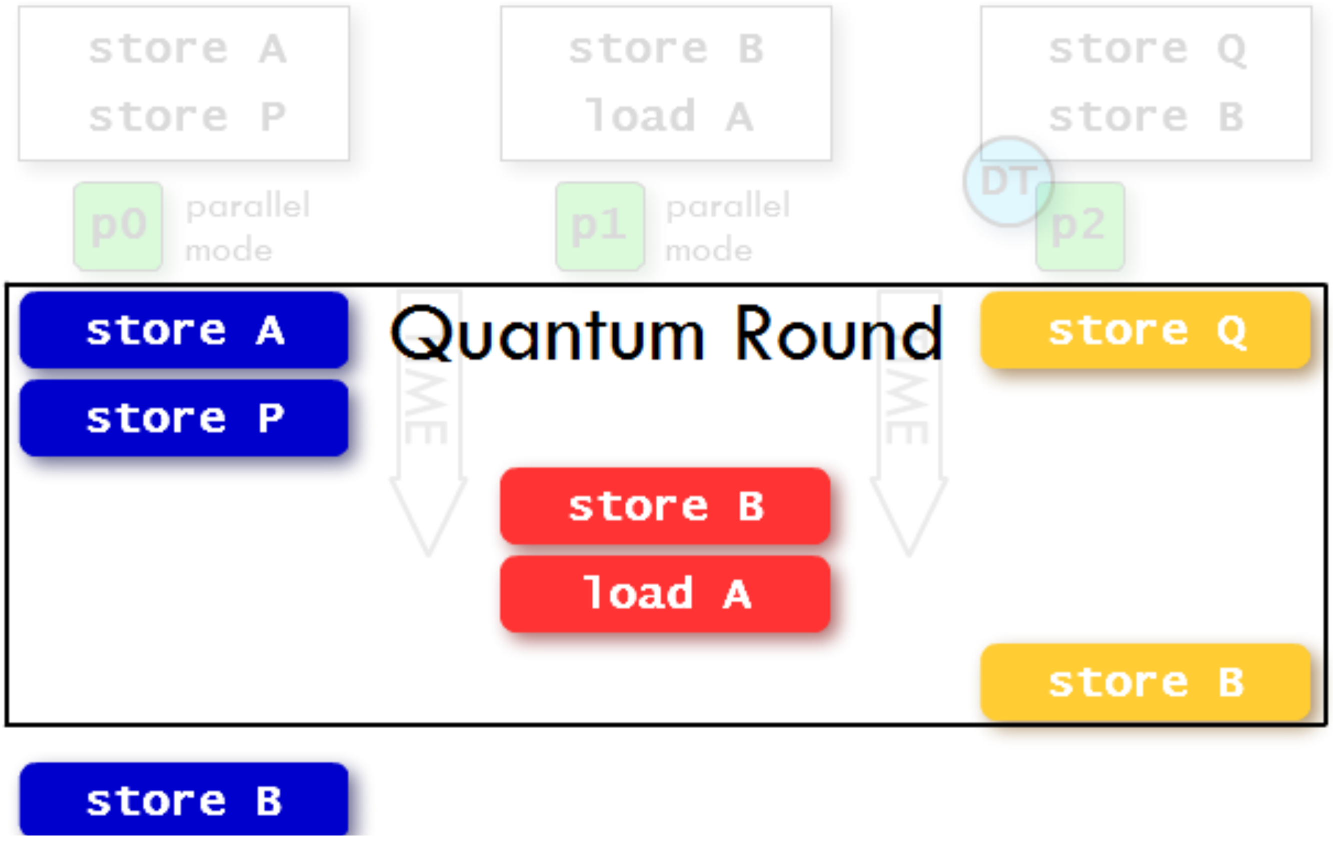
DMP-Ownership Example



DMP-Ownership Example



DMP-Ownership Example



store A
store P

p0 parallel mode

store B
load A

p1 parallel mode

store Q
store B

DT

p2

Quantum Round

store A
store P

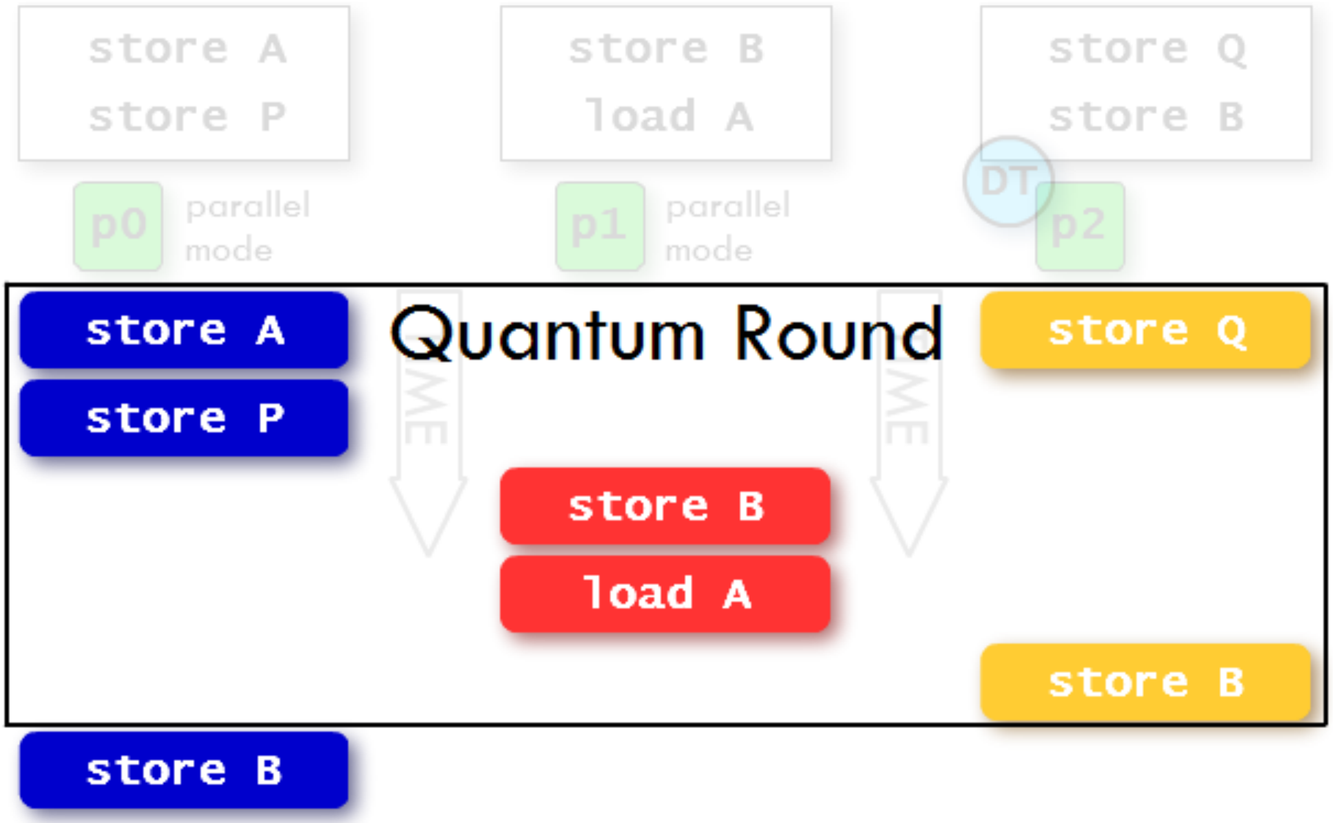
store B
load A

store Q

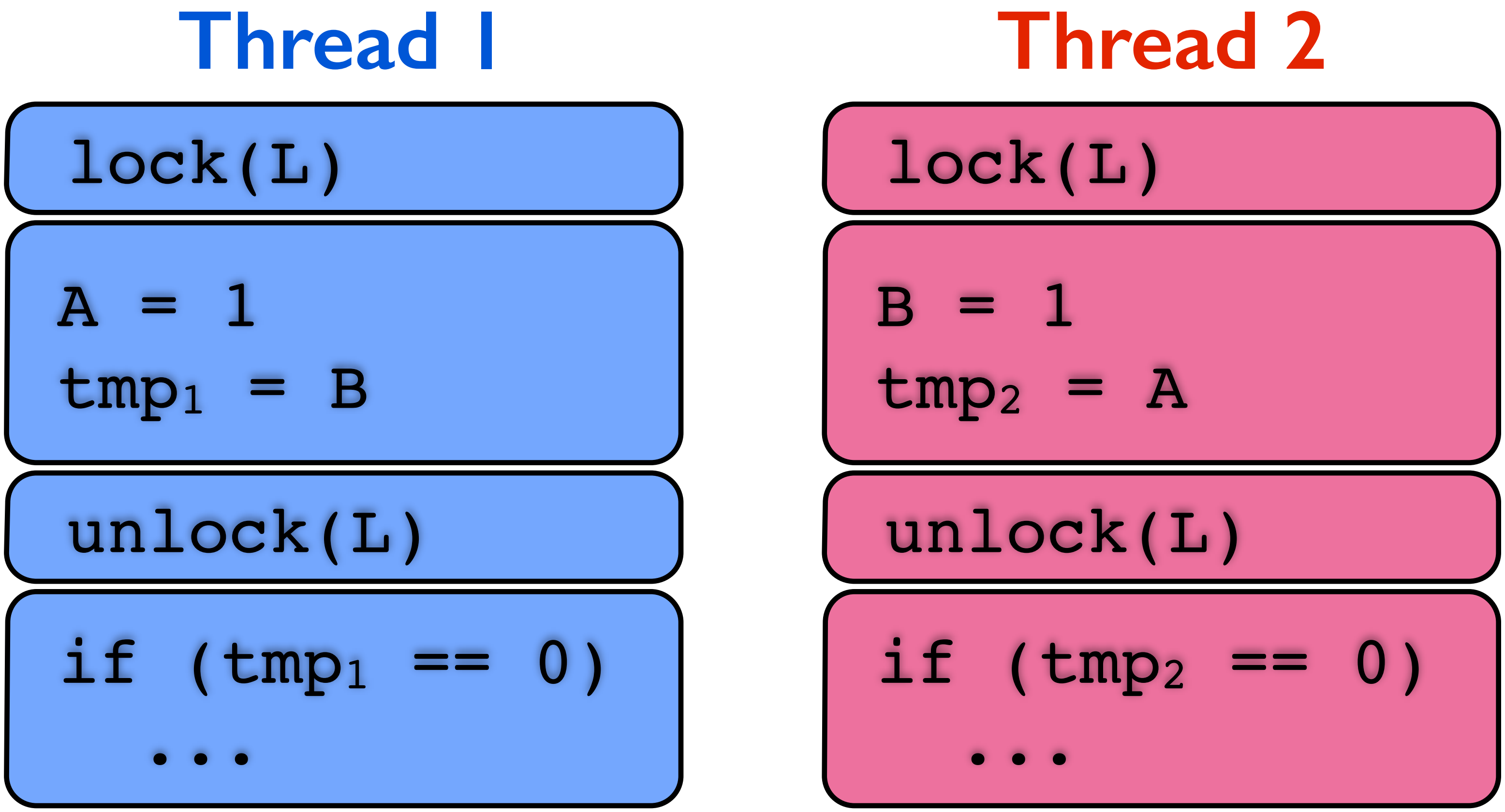
store B

store B

DMP-Ownership Example



DMP-Buffering



Thread 1

```
lock(L)
```

```
A = 1
tmp1 = B
```

```
unlock(L)
```

```
if (tmp1 == 0)
  ...
```

Thread 2

```
lock(L)
```

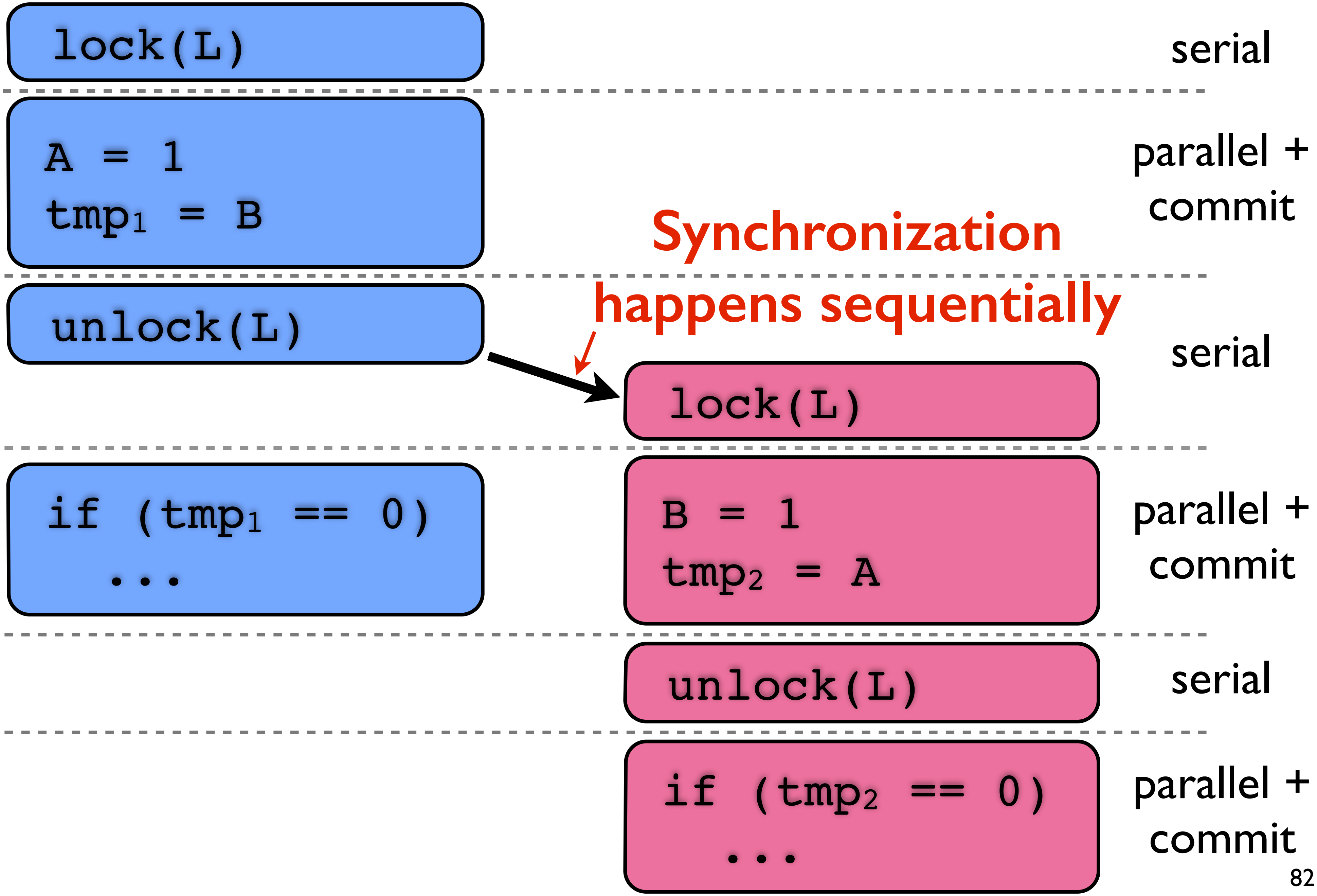
```
B = 1
tmp2 = A
```

```
unlock(L)
```

```
if (tmp2 == 0)
  ...
```

Dekker's Algorithm
(no data race)

DMP-Buffering



lock(L)

serial

A = 1

tmp$_1$ = B

parallel + commit

unlock(L)

Synchronization happens sequentially

lock(L)

serial

if (tmp$_1$ == 0)
  ...

B = 1

tmp$_2$ = A

parallel + commit

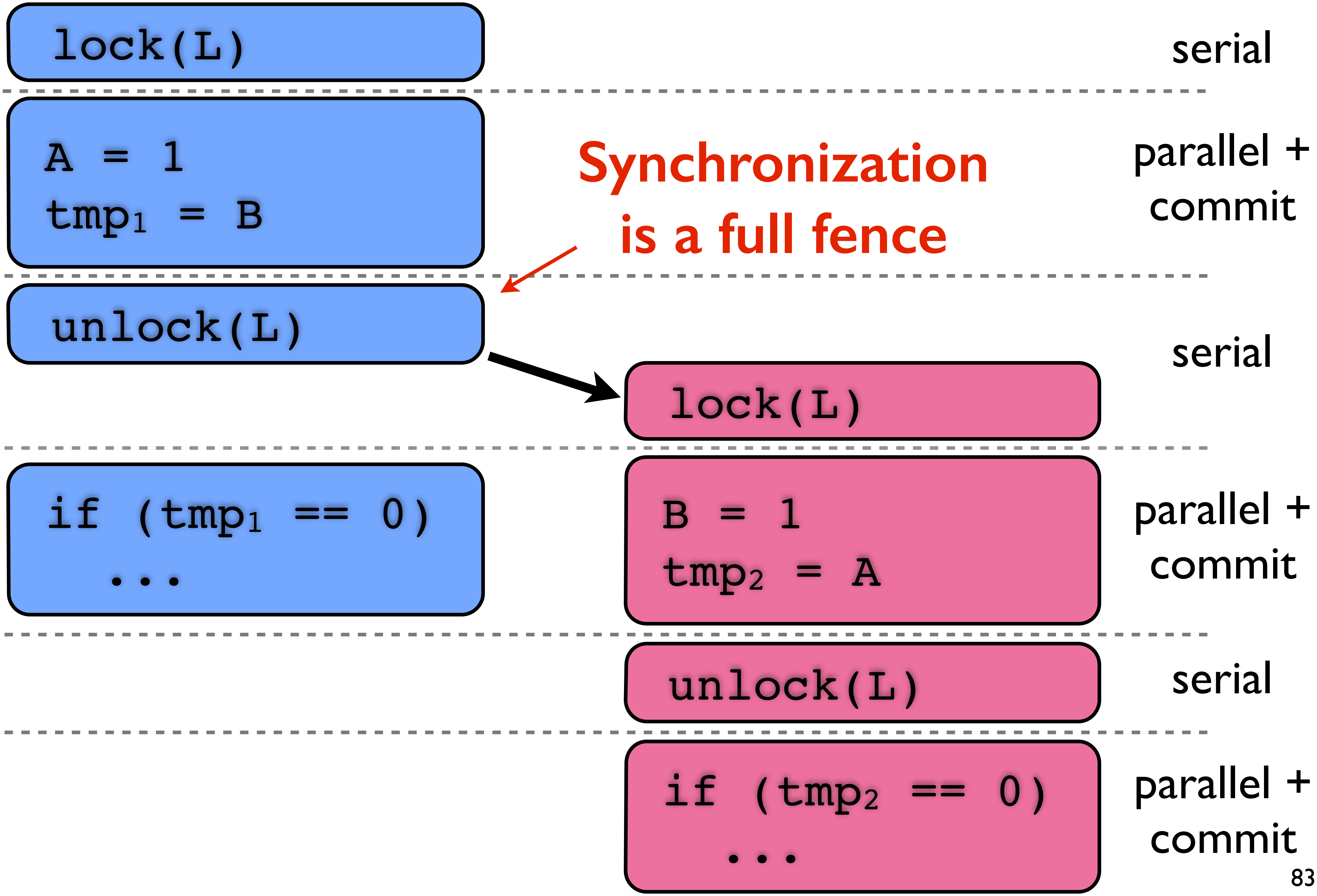
unlock(L)

serial

if (tmp$_2$ == 0)
  ...

parallel + commit

DMP-Buffering



lock(L)

serial

A = 1

tmp_1 = B

parallel + commit

Synchronization is a full fence

unlock(L)

serial

lock(L)

if (tmp_1 == 0)
    ...

B = 1

tmp_2 = A

parallel + commit

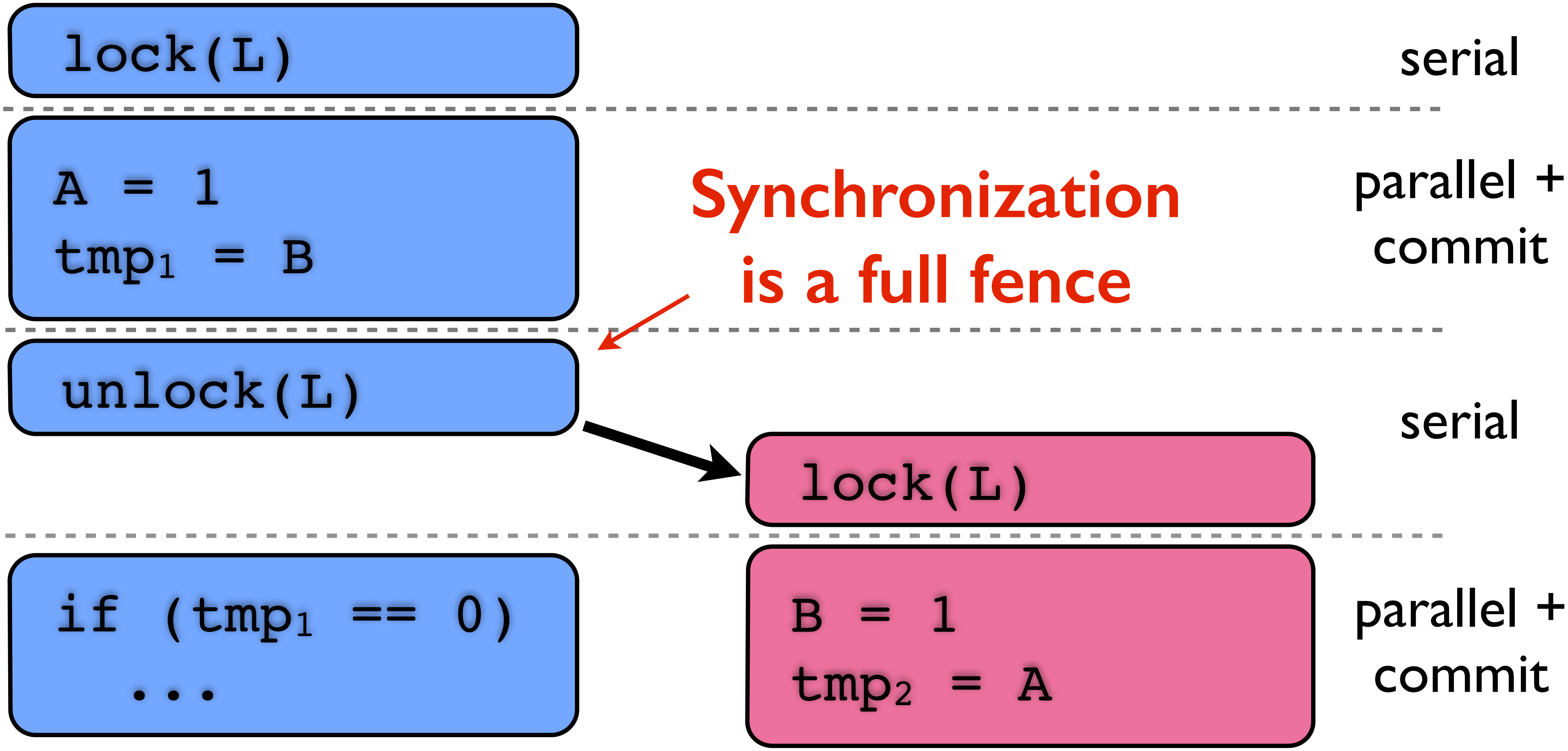
unlock(L)

serial

if (tmp_2 == 0)
    ...

parallel + commit

DMP-Buffering



lock(L)

serial

A = 1

tmp$_1$ = B

parallel + commit

Synchronization is a full fence

unlock(L)

serial

lock(L)

if (tmp$_1$ == 0)

...

B = 1

tmp$_2$ = A

parallel + commit

Data race free programs are sequentially consistent (required by C++ and Java memory models)