Indistinguishability: Friend and Foe of Concurrent Data Structures

Hagit Attiya
CS, Technion

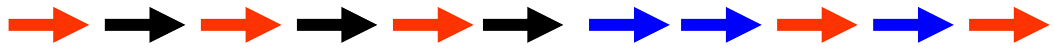
What if things are pretending to be what they really are...?

- **Uncertainty** is a main obstacle for designing correct applications in concurrent systems

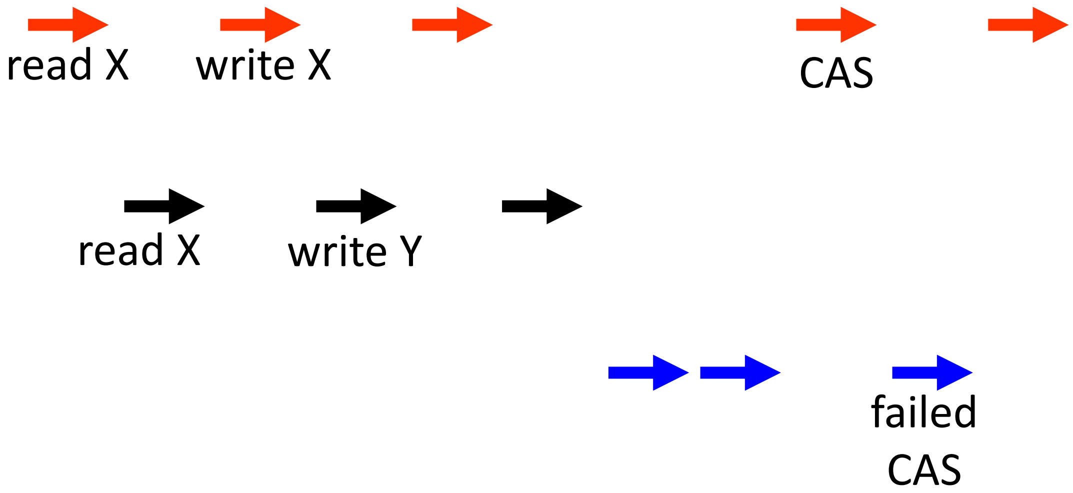
- Formally captured by **indistinguishabilty**,
  so arguing about it gives us important insights

- Three examples
  The good (helpful) ☺
  The bad (limiting) ☹
  & The ???... 😐

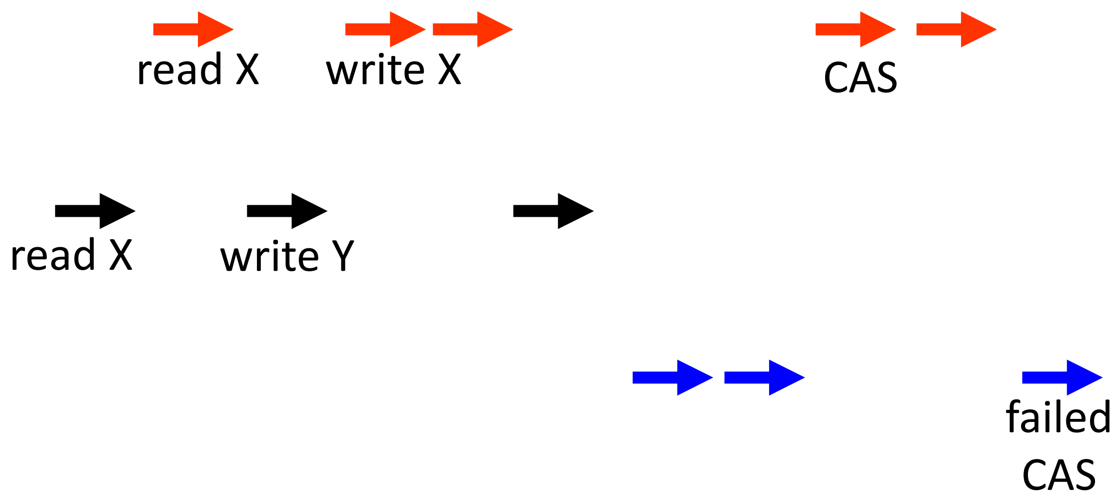
Traces of a Concurrent System



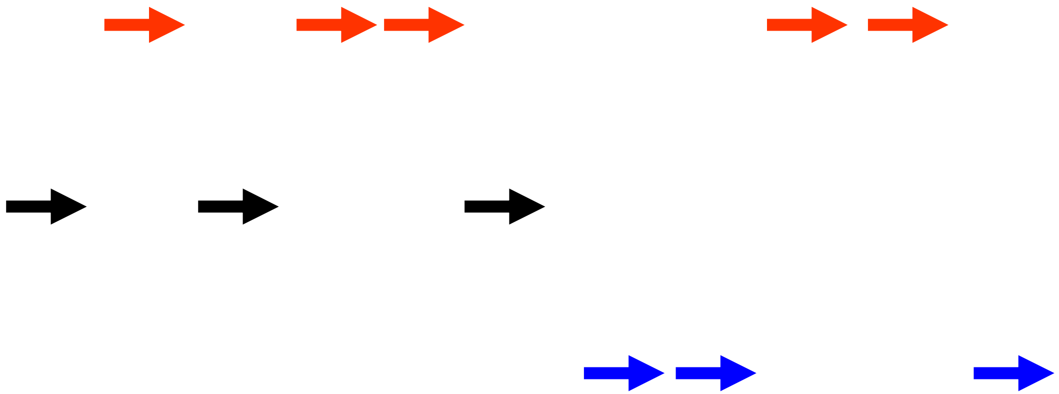
Projecting on Thread-Local Views



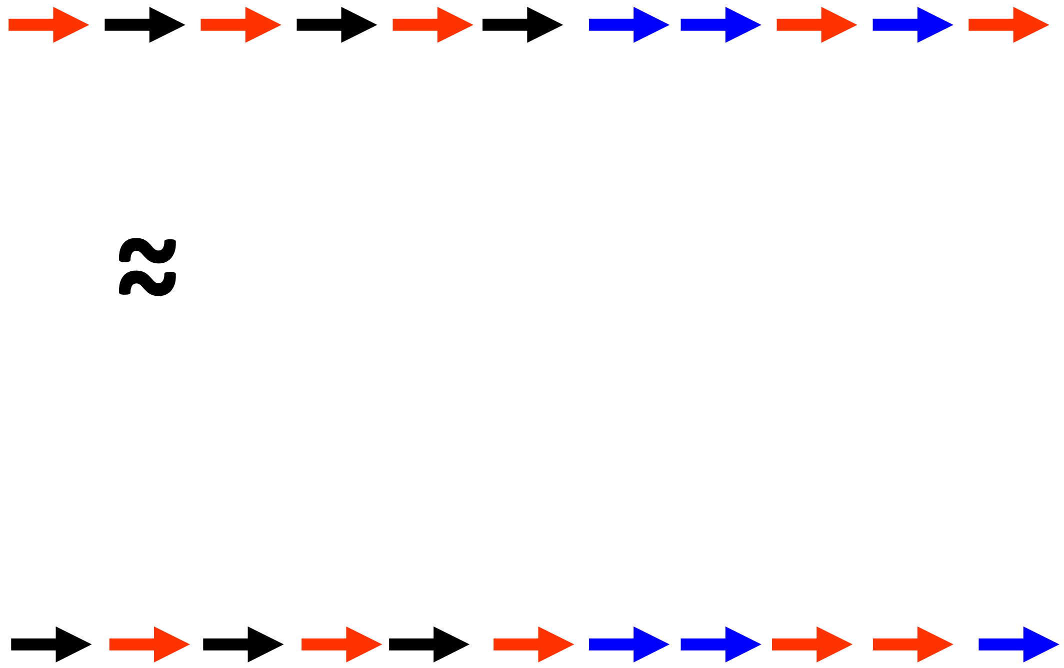
Indistiguishability: Same Local Views



Indistiguishable Traces:
Same Local Views



Indistiguishable Traces: Same Local Views



1. Reductions for Serializability ☺

Static analysis of concurrent data structures, by **sequential reductions**:

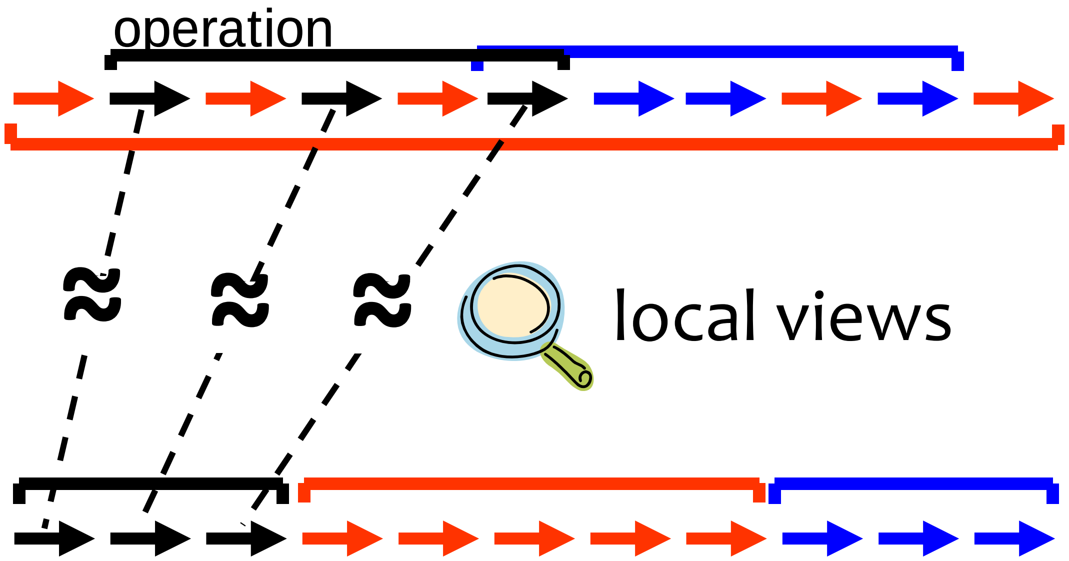
Consider only **sequential traces**, and deduce properties in **all traces**

Attiya, Ramalingam, Rinetzky: Sequential verification of serializability. POPL 2010

Serializability

[Papadimitriou '79]

interleaved **trace**



complete non-interleaved **trace**

Serializability ⇨ Sequential Reduction

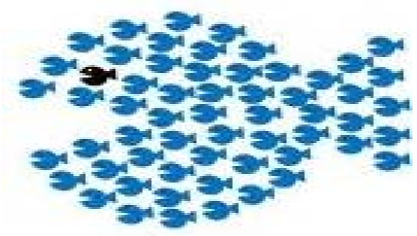
Concurrent serializable code M, **local property** ϕ

– Holds in a trace iff holds in all **indistinguishable** traces

[Papadimitriou '79] easily imply

ϕ holds in all traces of M iff ϕ holds in all complete non-interleaved traces of M

How to check M is serializable,
w/o considering all traces?



Disciplined Programming with Locks

Locking protocol ensures conflict serializability

- two-phase locking (2PL), tree locking (TL), (dynamic) DAG locking

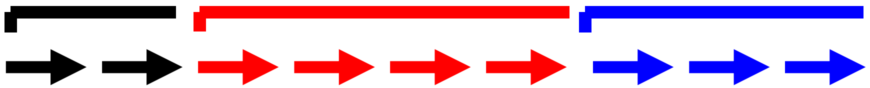
Verify that M **respects** a **local** locking protocol

- Depending only on thread's local variables & global variables locked by it
- Not centralized concurrency control monitor!

Considering only non-interleaved traces

Our Contribution: First Step

~~complete~~ non-interleaved traces of M



Two phase locking
Tree locking
Dynamic tree locking

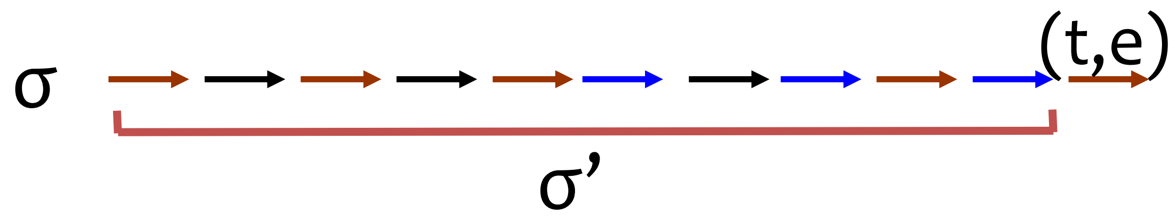
A local conflict serializable locking policy is respected in all traces iff it is respected in all non-interleaved traces

A local property holds in all traces iff it holds in all non-interleaved traces

Reduction to Non-Interleaved Traces: Idea

Let σ be the **shortest** trace that violates the locking protocol LP

⇨ σ' follows LP, guarantees conflict-serializability

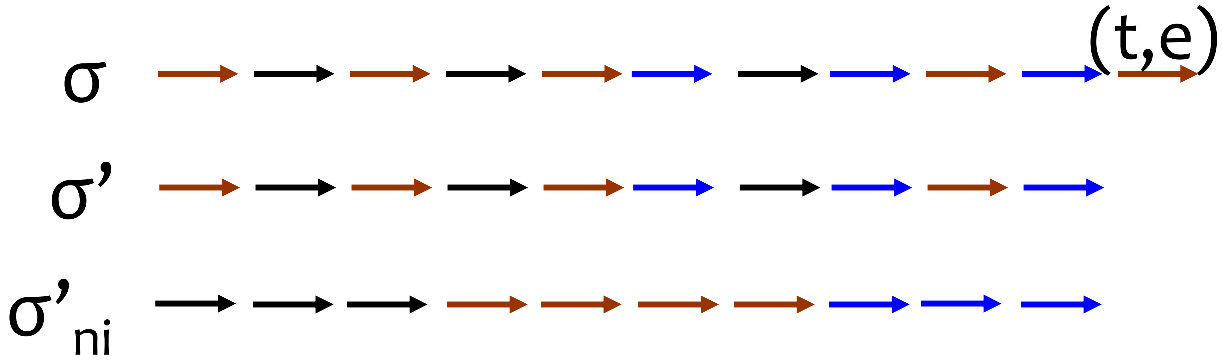


Reduction to Non-Interleaved Traces: Idea

Let σ be the **shortest** trace that violates the locking protocol LP

⇨ σ' follows LP, guarantees conflict-serializability

⇨ ∃ non-interleaved trace **indistinguishable** from σ'

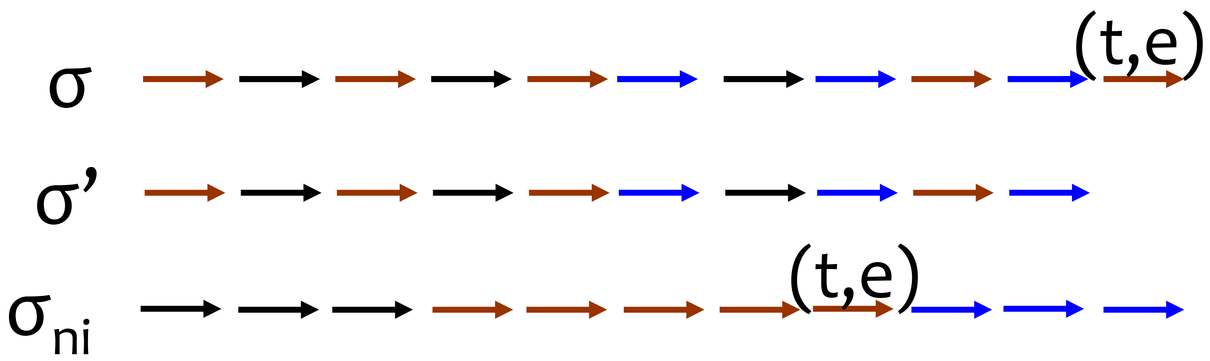


Reduction to Non-Interleaved Traces: Idea

Let σ be the **shortest** trace that violates the locking protocol LP

⇨ σ' follows LP, guarantees conflict-serializability

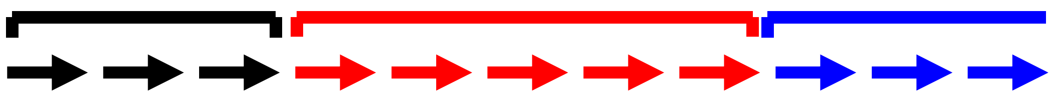
⇨ ∃ non-interleaved trace **indistinguishable** from σ'



⇨ ∃ non interleaved trace (indistinguishable from σ) where LP is violated

Further reduction

Almost-complete non-interleaved traces



A local conflict serializable locking policy is respected in all traces iff it is respected in all almost-complete non-interleaved traces

Need to argue about termination

2. When are barriers necessary? ☹

Expensive memory ordering should be enforced in order to ensure correctness of certain concurrent data structures

Attiya, Guerraoui, Hendler, Kuznetsov, Michael, Vechev: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. POPL 2011

The Result & Its Scope

- Concurrent data types:
  - **Strongly non-commutative** operations
    - Operations A and B s.t. A influences B, and B influences A
    - E.g., two deq operations, counters, hash tables, trees,…
  - Serializable solo-terminating implementations

- Mutual exclusion

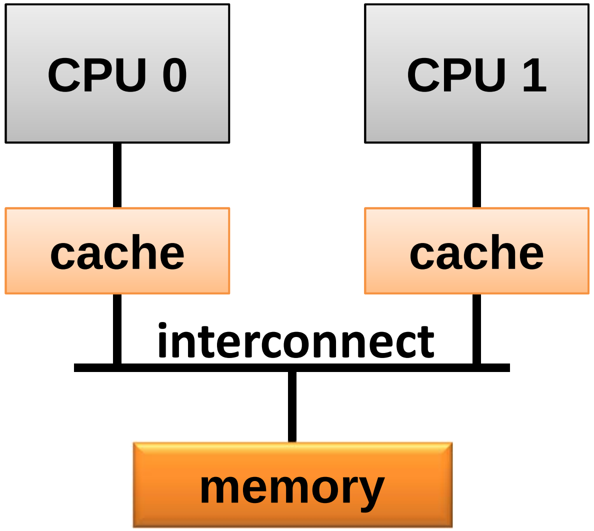
Any concurrent program for these problems must use read-after-write unless it has atomic-write-after-read

What this Means?

Multicores issue memory accesses **out of order**,
to compensate for slow writes

In particular (and very common)

Issue a read before an earlier write,
if they access different locations



Avoiding Out-of-Order Execution

Insert read-after-write (**RAW**) fence

```
Write(X,1)

FENCE

Read(Y)
```

Use **atomic-write-after-read** (**AWAR**)

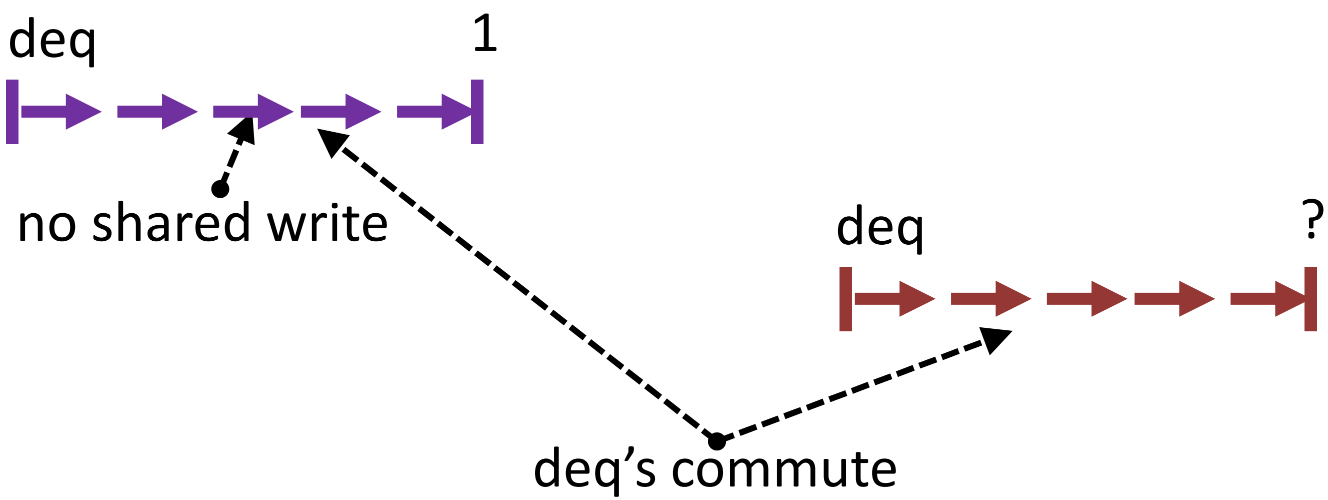
E.g., CAS, test&set, fetch&add,…

```
atomic{
   read(Y)
   …
   write(X,1)
}
```

RAW fences / AWAR are ~60 slower than (remote) memory accesses

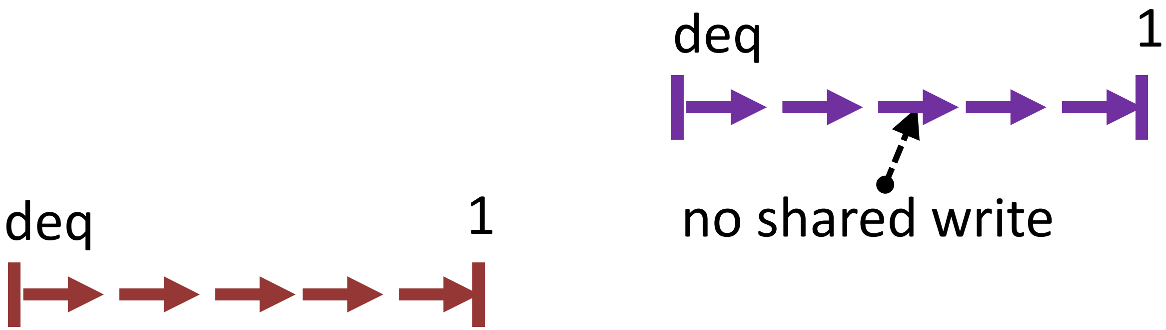
Proof: Must Write

If a deq does not write,
it does not influence other operations



Proof: Must Write

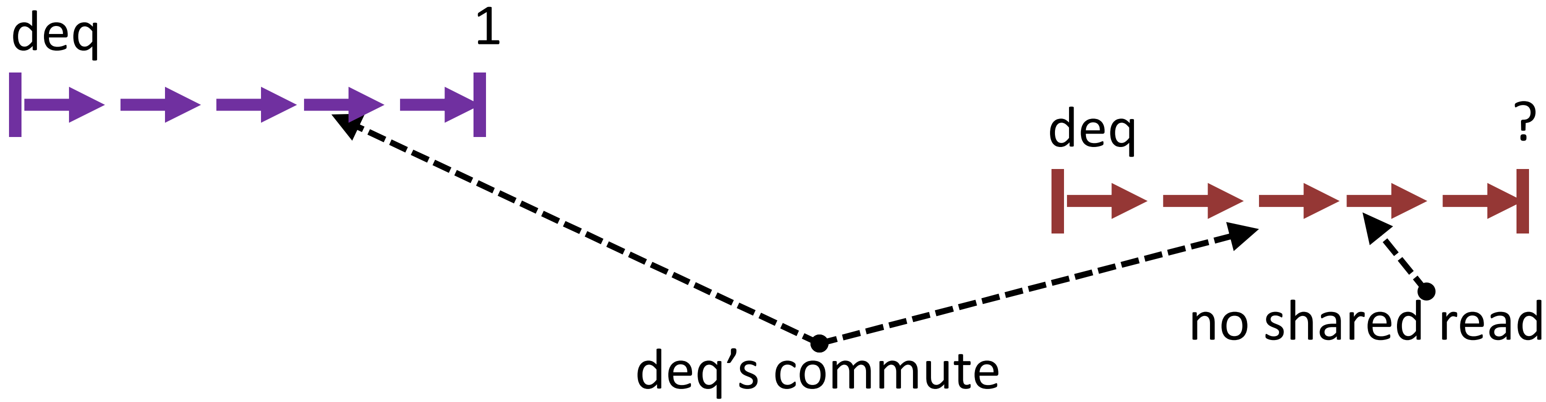
If a deq does not write,
it does not influence other operations



Indistinguishable from a trace where deq's are exchanged (and 1 is returned twice)

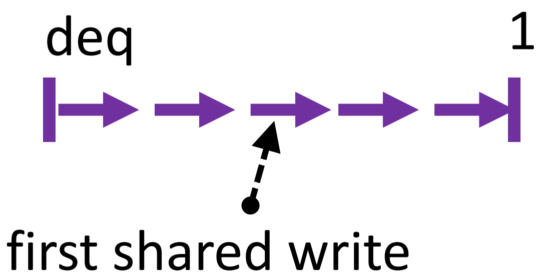
Proof: Must Also Read

If a deq does not read,
it is not influenced by other operations

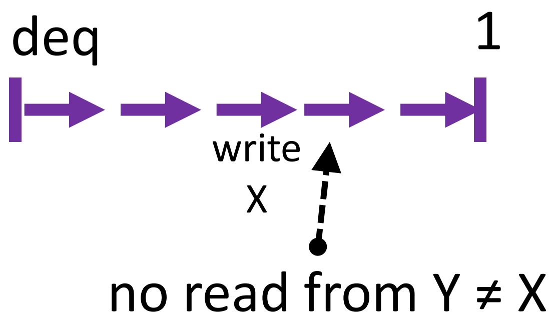


Indistinguishable from a trace where deq's are exchanged

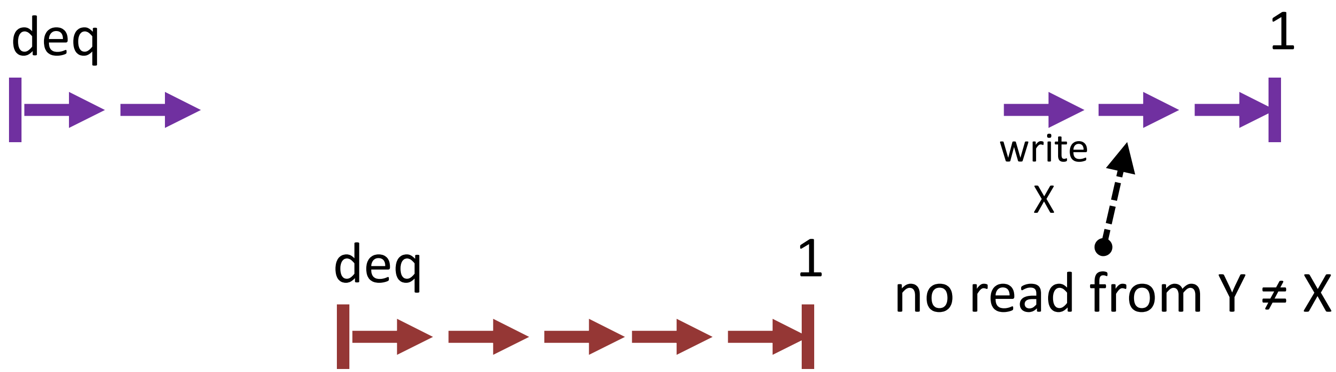
Close-Up on the 1st Dequeue



Close-Up on the 1st Dequeue



Covering Leads to Indistinguishability



No legal serialization (1 is dequed twice)

3. Substituting TM for atomic blocks

Opaque transactional memory is equivalent to **atomic blocks** in concurrent programs

Attiya, Hans, Gotsman, Rinetzky: Abstractions for Transactional memory. To appear in PODC 2013

Programming with Atomic Blocks

```
g :=0;
r := atomic{
  x := A.write(2);
  y := B.write(4)};
If (r = commit) then
  g := 1
else e := x
```

```
s := abort
while (s ≠ commit) do
  s := atomic{
    u := A.read();
    v := B.read()};
z := g;
if (z = 1) then
  three := 6 / (v - u)
```

Programming with Atomic Blocks

```
g :=0;
r := atomic{
  x := A.write(2);
  y := B.write(4)};
If (r = commit) then
  g := 1
else e := x
```

```
s := abort
while (s ≠ commit) do
  s := atomic{
    u := A.read();
    v := B.read()};
z := g;
if (z = 1) then
  three := 6 / (v - u)
```

Design for an abstract Transactional Memory, assuming **code blocks** that execute **atomically**

TM_A

Programming with Atomic Blocks

```
g :=0;
r := atomic{
  x := A.write(2);
  y := B.write(4)};
If (r = commit) then
  g := 1
else e := x
```

```
s := abort
while (s ≠ commit) do
  s := atomic{
    u := A.read();
    v := B.read()};
z := g;
if (z = 1) then
  three := 6 / (v - u)
```

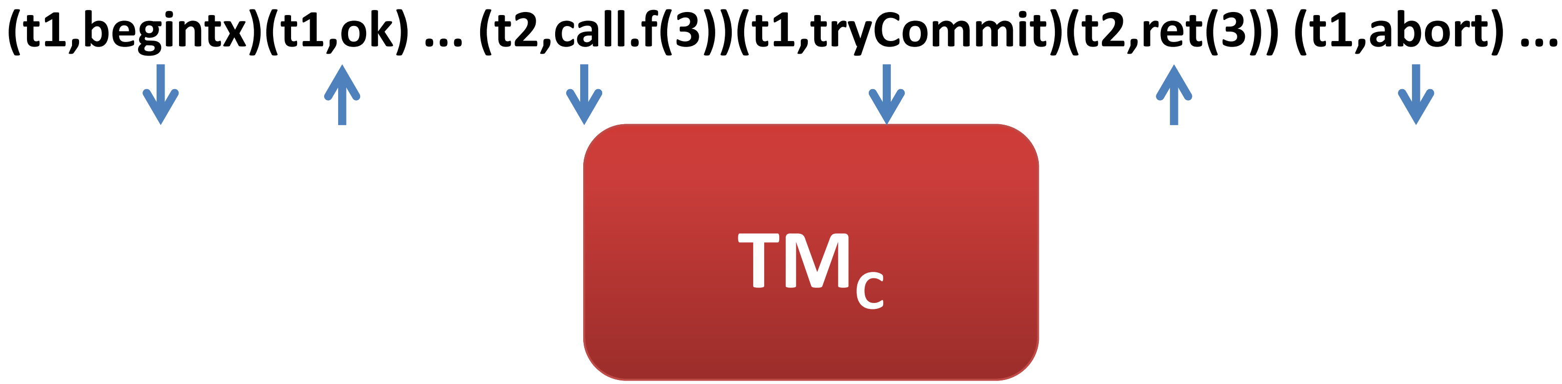
Execute with a concrete TM implementation, replacing atomic blocks with **transactions**

TM_C

Concrete TMs

TM_C is a library for read, write, commit, …

History: invocations and responses between the program and the TM_c



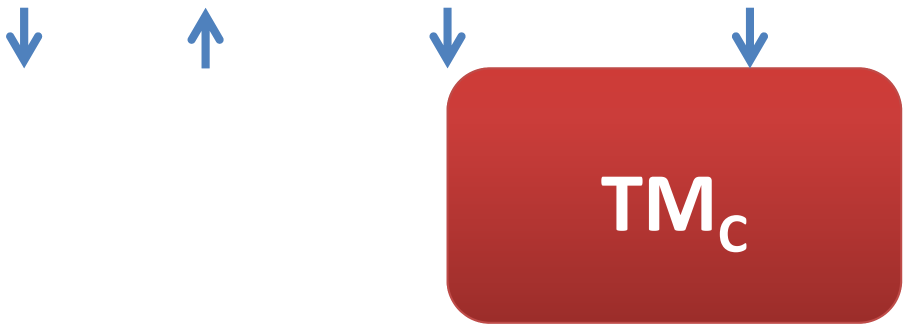
TM Consistency Conditions

Restrict the possible histories, e.g.

- **Opacity** [Guerraoui & Kapalka, '08]
- **Virtual World Consistency** [Imbs et al. '09]
- **TMS** [Doherty et al. '09]

But which of them is **THE RIGHT ONE**?

(t1,begintx)(t1,ok) ... (t2,call.f(3))(t1,tryCommit)(t2,ret(3)) (t1,abort) ...



TM Consistency Conditions

Restrict the possible histories, e.g.

- **Opacity** [Guerraoui & Kapalka, '08]
- **Virtual World Consistency** [Imbs et al. '09]
- **TMS** [Doherty et al. '09]

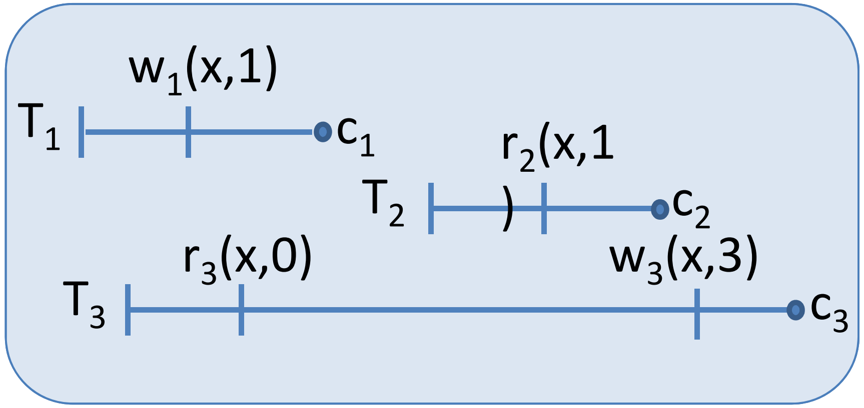
But which of them is **THE RIGHT ONE**?

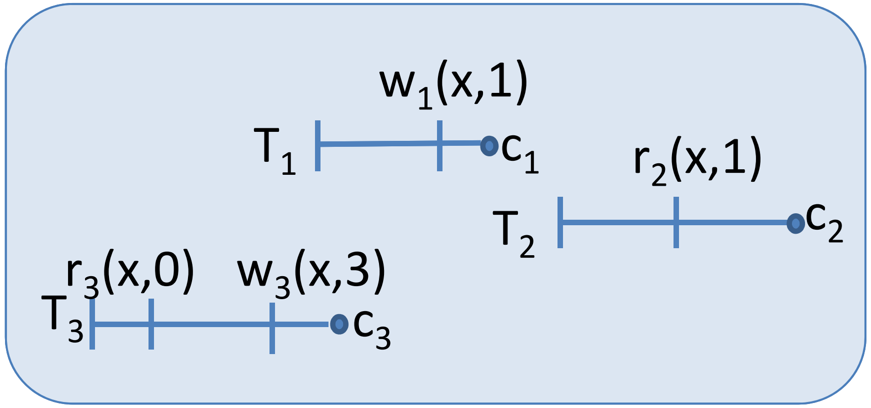
- Ensures TM_C replaces TM_A correctly (**soundness**)
- Enforces minimal restrictions (**completeness**)

Observational refinement: Programs (in some set) have the same views under TM_c and TM_A

Opacity Relation $H \sqsubseteq_{OP} S$

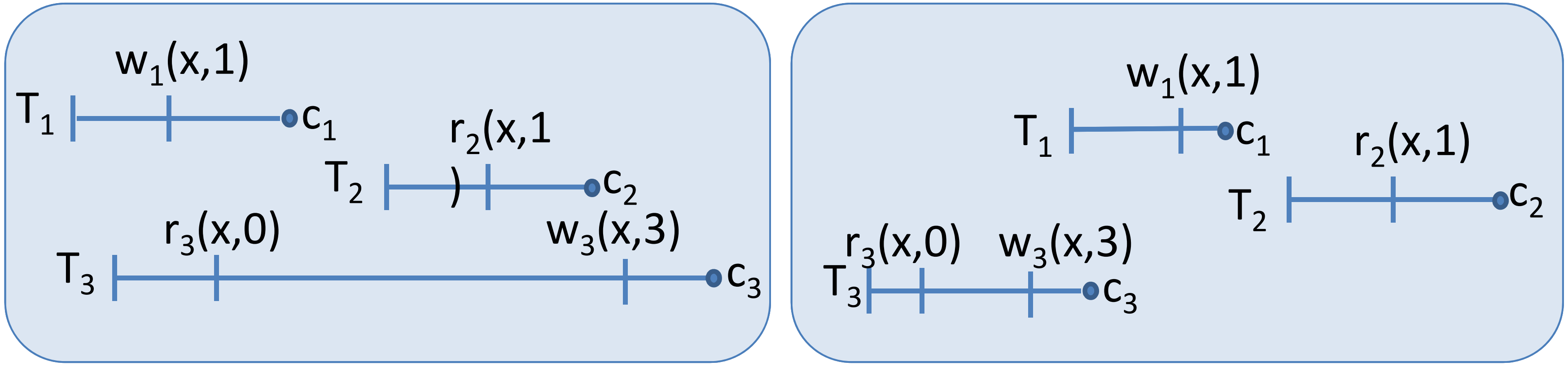
History S preserves **per-thread order** and
order of non-overlapping transactions in history H





Soundness: $H \sqsubseteq_{OP} S \Rightarrow$ Observational Refinement

History S preserves **per-thread order** and **order of non-overlapping** transactions in history H



- no nesting
- no privatization
- finite histories

Soundness: Proof Outline

[Fix a program and an initial state…]

Consider a trace σ of TM_C with history H,
and assume $H \sqsubseteq_{OP} S$ for some history S of TM_A

Construct a trace τ' ≈ τ of TM_A

⇨Every view observed when running the program with TM_C is also observed with TM_A

How to Construct τ' From τ?

From the trace τ of TM_C & the history S of TM_A construct a trace $\tau' \approx \tau$ of TM_A

Can gather together events of each atomic block (between start & end of a transaction) since

- No access to global variables inside atomic blocks, only to transactional variables
- Changes to transactional variables impact other threads only at the end of a block

Completeness: $\sqsubseteq_{\mathrm{OP}}$ is Necessary

- Construct a program P_H for every history H
- Real-time order in every trace of P_H
  must agree with the real-time order of the transactions in H

Summary

Indistinguishability **partitions** computations into classes

Reduce the difficulty of designing / verifying concurrent programs by picking / constructing a **representative** computation from each class

  to verify, or
  to show it violates desired properties

You can do it too…