

DeterV: Verifying Deterministic Parallel Execution

Scofield Zhengqing Liu, Marios Kogias

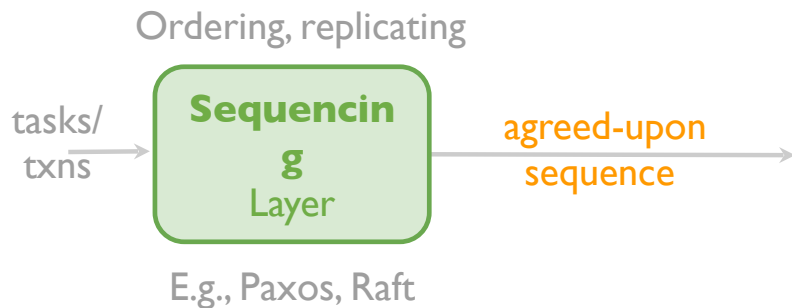


Imperial College London

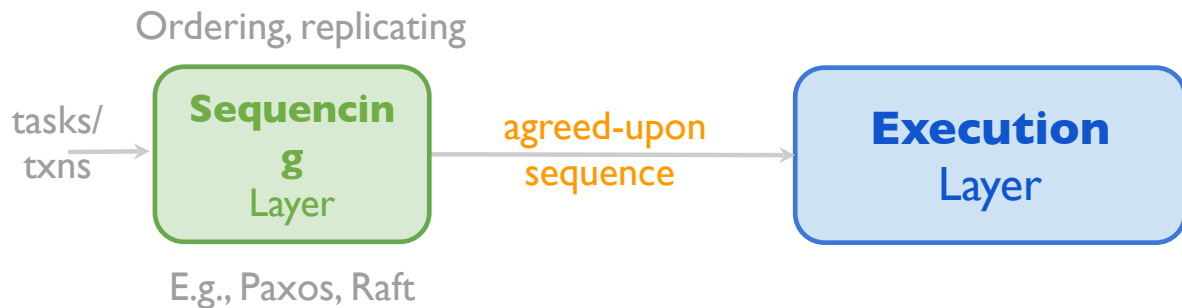
PLOS 2026

- In distributed systems, replication is widely used for fault-tolerance

- In distributed systems, replication is widely used for fault-tolerance
- Replication consists of two layers: sequencing + execution

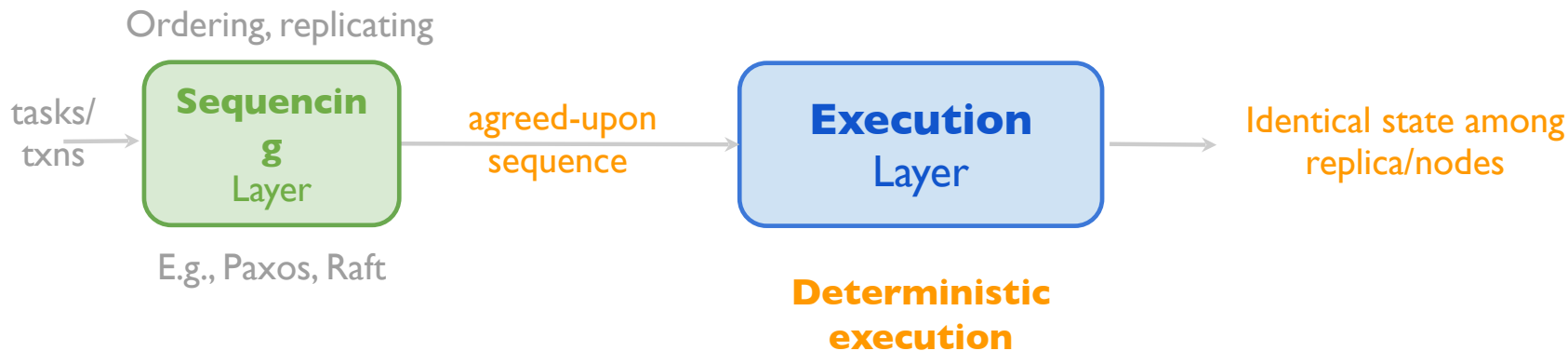


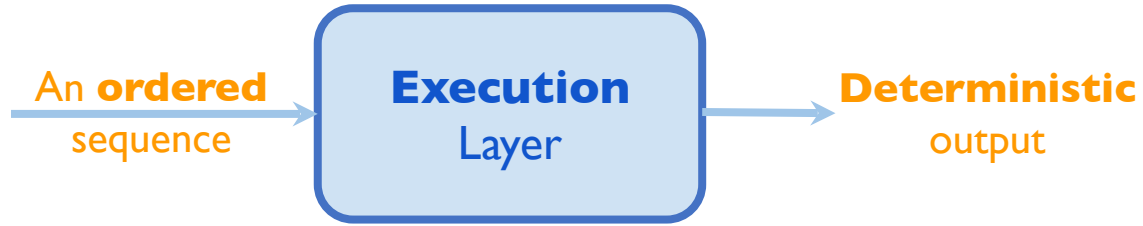
- In distributed systems, replication is widely used for fault-tolerance
- Replication consists of two layers: sequencing + execution

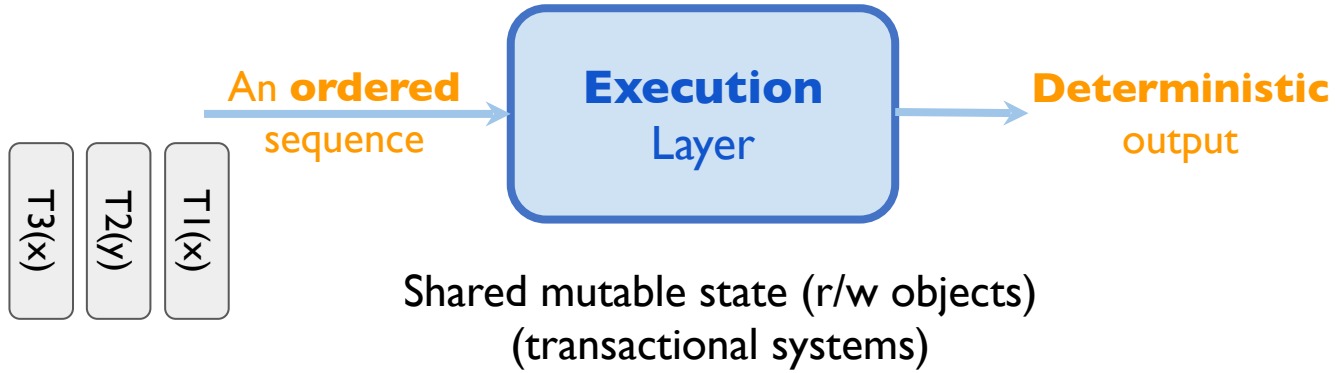


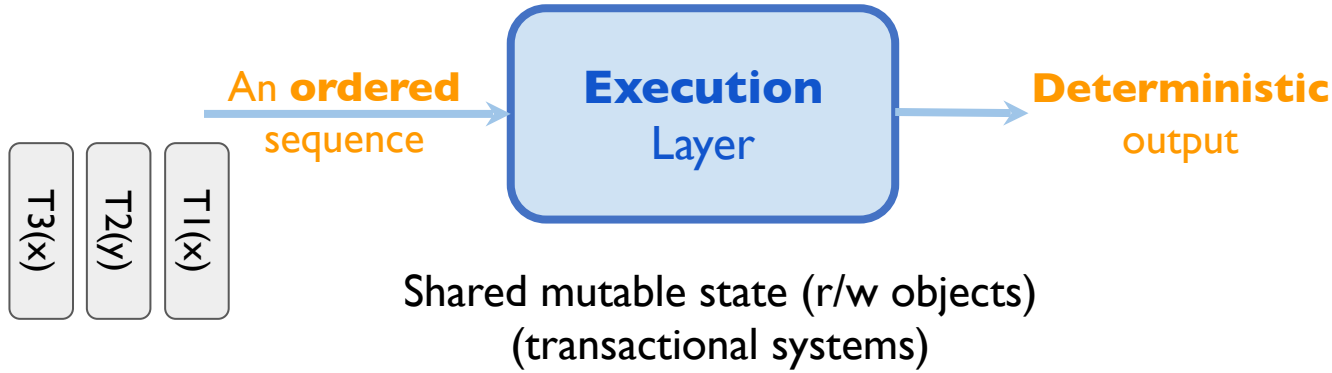
Deterministic execution is critical for replicated systems

- In distributed systems, replication is widely used for fault-tolerance
- Replication consists of two layers: sequencing + execution

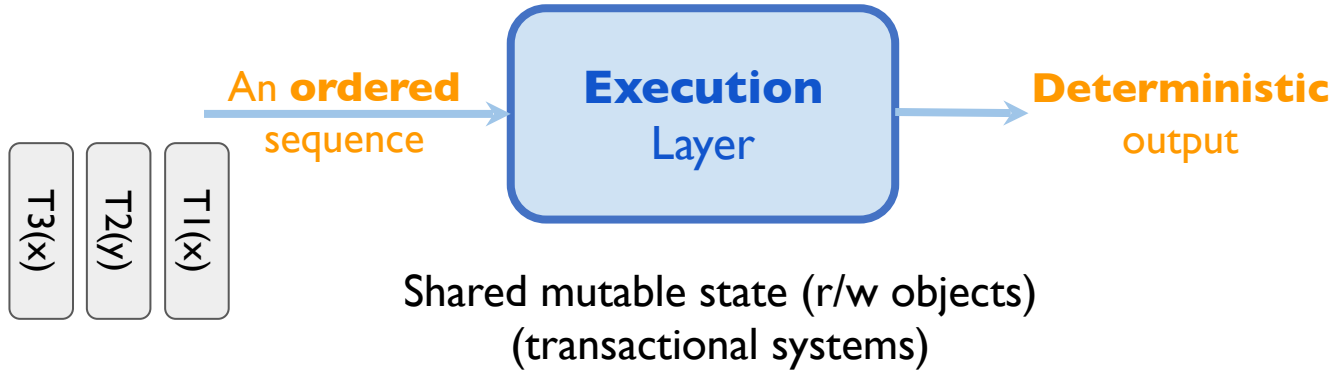




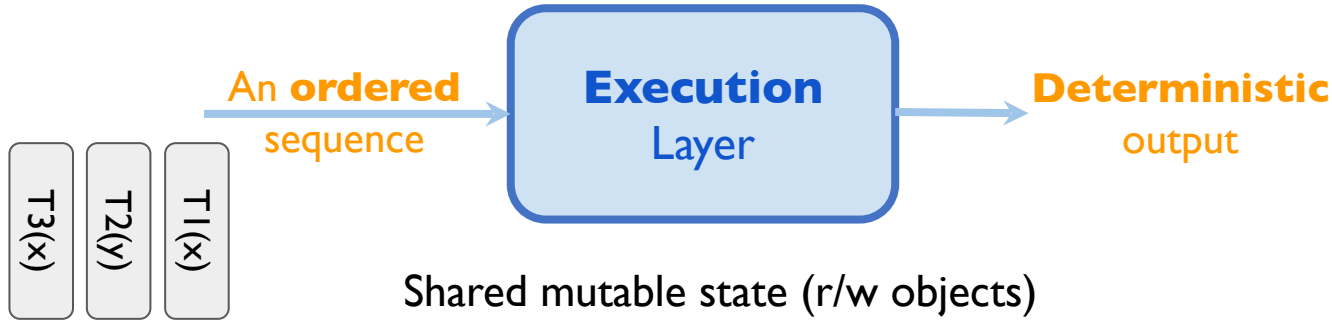




Sources of Non-Deter:
floating point,
rng, timer
thread-interleavings



Sources of Non-Deter:
floating point,
rng, timer
▲ thread-interleavings

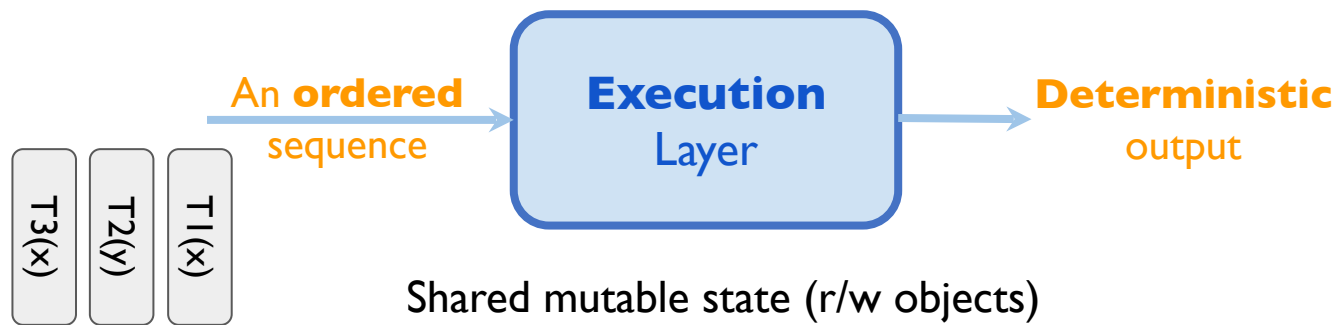


Shared mutable state (r/w objects)
(transactional systems)

Single-threaded execution

Sources of Non-Deter:
floating point,
rng, timer
▲ thread-interleavings

Deterministic parallel execution (DPE) eliminates the single-threaded execution bottleneck!



Shared mutable state (r/w objects)
(transactional systems)

Single-threaded execution

**Deterministic parallel execution
(DPE)**

Sources of Non-Deter:
floating point,
rng, timer
▲ thread-interleavings

DPE has been explored in DB, blockchains, and datacenter microservices (Bohm, Caracal, Block-STM, DORADD, etc.)

DPE boosts performance, but leaves correctness assumed

DPE has been explored in DB, blockchains, and datacenter microservices (Bohm, Caracal, Block-STM, DORADD, etc.)

DPE boosts performance, but leaves correctness assumed

DPE has been explored in DB, blockchains, and datacenter microservices (Bohm, Caracal, Block-STM, DORADD, etc.)

- **Existing DPE:** determinism by design, correctness argued informally

DPE boosts performance, but leaves correctness assumed

DPE has been explored in DB, blockchains, and datacenter microservices (Bohm, Caracal, Block-STM, DORADD, etc.)

- **Existing DPE:** determinism by design, correctness argued informally
- **Bug impact:** replica divergence, lost reproducibility, exploitable order violations*

*Yunhao et al. Ordered Consensus. OSDI'26, OSDI'20.

Determinism proof exists, not at runtime/sys levels

Determinism proof exists, not at runtime/sys levels

Model/
Language

Kahn networks,
Vélus (PLDI'17),
Concurrent revisions (CPP'20)
Lingua Franca (TECS'21)

Determinism proof exists, not at runtime/sys levels

Model/
Language

Kahn networks,
Vélus (PLDI'17),
Concurrent revisions (CPP'20)
Lingua Franca (TECS'21)

Program

Muskeer (POPL'26)
Sequential determinism (ICSE'21)

Determinism proof exists, not at runtime/sys levels

Model/
Language

Kahn networks,
Vélus (PLDI'17),
Concurrent revisions (CPP'20)
Lingua Franca (TECS'21)

Program

Muskeer (POPL'26)
Sequential determinism (ICSE'21)

**Abstract
semantic
s**

Kernel: Atmosphere (SOSP25)
Memory management: CortenMM (SOSP25)
Cluster controller: Anvil (OSDI24, SOSP26)
Storage: PoWER (OSDI25)

Our work focuses on **verifying DPE at the runtime system level**. Recent advances in systems verification tools (Verus, Dafny, etc.) make this practical.

Challenges in Verifying DPE

Challenges in Verifying DPE

CI: Thread interleaving blows up the global schedule space

Challenges in Verifying DPE

CI: Thread interleaving blows up the global schedule space

- Can we prove determinism without ever enumerating a schedule?

Challenges in Verifying DPE

C1: Thread interleaving blows up the global schedule space

- Can we prove determinism without ever enumerating a schedule?

C2: Different design choices/implementation mechanisms

- Pessimistic: schedule-then-execute
- Optimistic: execute-then-validate (abort/retry)

Challenges in Verifying DPE

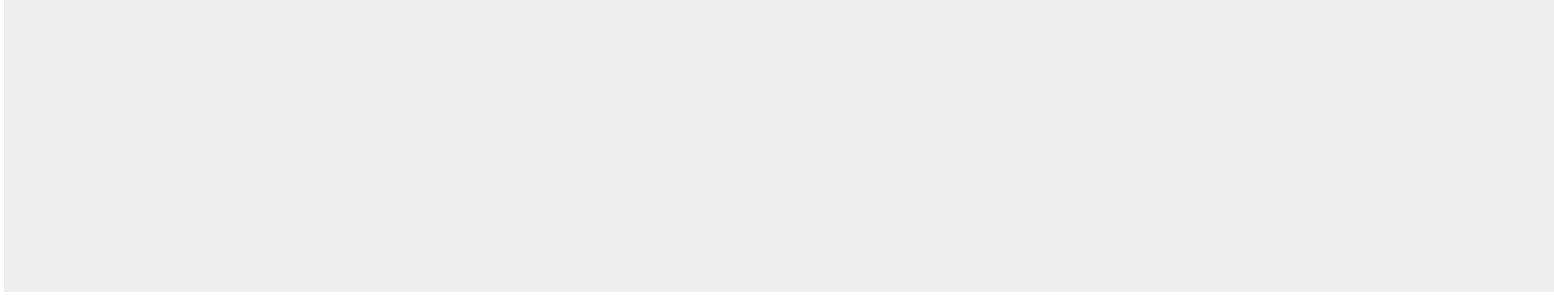
C1: Thread interleaving blows up the global schedule space

- Can we prove determinism without ever enumerating a schedule?

C2: Different design choices/implementation mechanisms

- Pessimistic: schedule-then-execute
- Optimistic: execute-then-validate (abort/retry)
- Can we have unified primitive/abstraction to reuse the verification machinery?

DeterV Insights

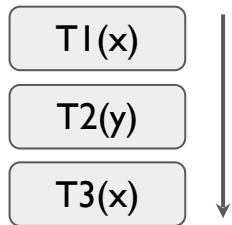


DeterV Insights

Insight I (local reasoning): **Localize** by objects, then **compose**

- Intuition: each object has a dependency chain for exec
- Avoid considering numerous interleavings (CI) and support disjoint states reasoning

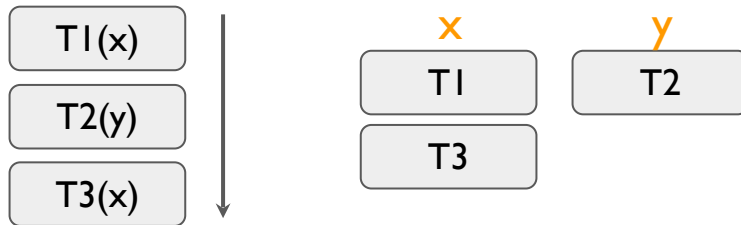
DeterV Insights



Insight I (local reasoning): **Localize** by objects, then **compose**

- Intuition: each object has a dependency chain for exec
- Avoid considering numerous interleavings (CI) and support disjoint states reasoning

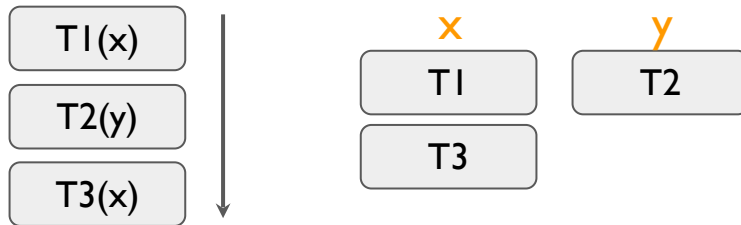
DeterV Insights



Insight I (local reasoning): **Localize** by objects, then **compose**

- Intuition: each object has a dependency chain for exec
- Avoid considering numerous interleavings (CI) and support disjoint states reasoning

DeterV Insights



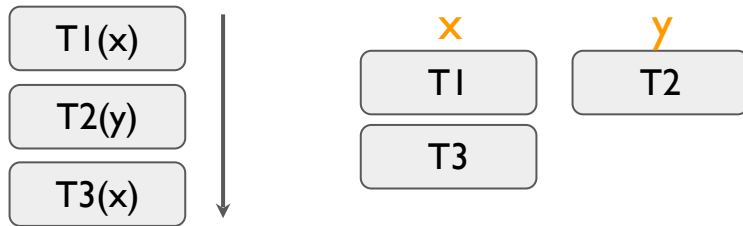
Insight 1 (local reasoning): **Localize** by objects, then **compose**

- Intuition: each object has a dependency chain for exec
- Avoid considering numerous interleavings (CI) and support disjoint states reasoning

Insight 2 (unified interface): Determinism is enforced at one effect boundary

- Both runtime families produces visible effects at a **commit** point.

DeterV Insights



Insight 1 (local reasoning): **Localize** by objects, then **compose**

- Intuition: each object has a dependency chain for exec
- Avoid considering numerous interleavings (CI) and support disjoint states reasoning

Insight 2 (unified interface): Determinism is enforced at one effect boundary

- Both runtime families produces visible effects at a **commit** point.

```
let res = task.run();  
commit_effect(res, state);
```

DeterV Overview

A verified DPE runtime in Verus

Build atop an unverified DPE system

Localize

each object follows its
own version order

(Insight 1)

Enforce

Check that order at
one commit boundary

(Insight 2)

Compose

layered refinement that
composes local reasoning
into the global one

An Unverified DPE Runtime

A Rust impl inspired by DORADD (PPoPP'25); and used in Remora (VLDB'26)

An Unverified DPE Runtime

A Rust impl inspired by DORADD (PPoPP'25); and used in Remora (VLDB'26)

- Single dispatcher thread
 - consumes incoming log
 - resolves dependencies
- Worker threadpool
 - async/await via dependencies

```
1 // Dispatcher: runs sequentially given the log order
2 while let Some(&mut task) = incoming.recv() {
3     assign_versions(task);
4     let (pre_deps, post_deps) = resolve_deps(task);
5
6     spawn_async(move || {
7         // worker: runs concurrently on the thread pool
8         pre_deps.await(); // 1. wait for predecessors
9         task.run();       // 2. execute task body
10        post_deps.signal(); // 3. unblock successors
11    });
12 }
```

An Unverified DPE Runtime

A Rust impl inspired by DORADD (PPoPP'25); and used in Remora (VLDB'26)

- Single dispatcher thread
 - consumes incoming log
 - resolves dependencies
- Worker threadpool
 - async/await via dependencies



```
1 // Dispatcher: runs sequentially given the log order
2 while let Some(&mut task) = incoming.recv() {
3     assign_versions(task);
4     let (pre_deps, post_deps) = resolve_deps(task);
5
6     spawn_async(move || {
7         // worker: runs concurrently on the thread pool
8         pre_deps.await(); // 1. wait for predecessors
9         task.run();       // 2. execute task body
10        post_deps.signal(); // 3. unblock successors
11    });
12 }
```

An Unverified DPE Runtime

A Rust impl inspired by DORADD (PPoPP'25); and used in Remora (VLDB'26)

- Single dispatcher thread
 - consumes incoming log
 - resolves dependencies
- Worker threadpool
 - async/await via dependencies

```
1 // Dispatcher: runs sequentially given the log order
2 while let Some(&mut task) = incoming.recv() {
3     assign_versions(task);
4     let (pre_deps, post_deps) = resolve_deps(task);
5
6     spawn_async(move || {
7         // worker: runs concurrently on the thread pool
8         pre_deps.await(); // 1. wait for predecessors
9         task.run();       // 2. execute task body
10        post_deps.signal(); // 3. unblock successors
11    });
12 }
```

An Unverified DPE Runtime

A Rust impl inspired by DORADD (PPoPP'25); and used in Remora (VLDB'26)

- Single dispatcher thread
 - consumes incoming log
 - resolves dependencies
- Worker threadpool
 - async/await via dependencies

```
1 // Dispatcher: runs sequentially given the log order
2 while let Some(&mut task) = incoming.recv() {
3     assign_versions(task);
4     let (pre_deps, post_deps) = resolve_deps(task);
5
6     spawn_async(move || {
7         // worker: runs concurrently on the thread pool
8         pre_deps.await(); // 1. wait for predecessors
9         task.run();       // 2. execute task body
10        post_deps.signal(); // 3. unblock successors
11    });
12 }
```

Verification Focus

- Target: **final state** determinism
- A reference spec model: sequential_exec
- $\text{DPE}(\log) \equiv \text{sequential_exec}(\log)$

DeterV Design

A verified DPE runtime in Verus

Localize

each object follows its
own version order

(Insight 1)



Enforce

Check that order at
one commit boundary

(Insight 2)

Compose

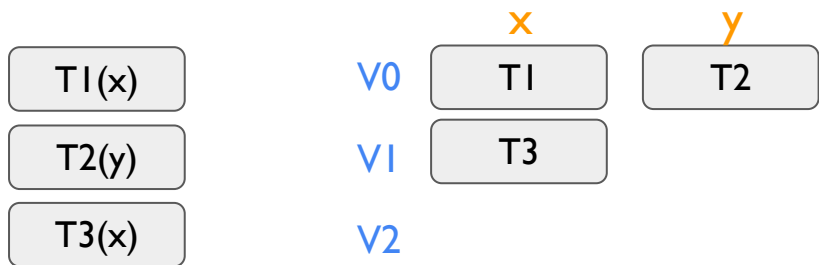
layered refinement that
composes local reasoning
into the global one

Local Reasoning via Object Versions

- Per-object **version** sequence
 - Each task advances accessed objects to exactly next version
 - Versions provide a simpler/compact abstraction for progress

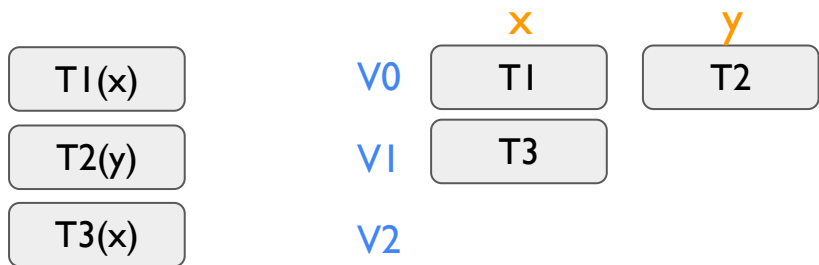
Local Reasoning via Object Versions

- Per-object **version** sequence
 - Each task advances accessed objects to exactly next version
 - Versions provide a simpler/compact abstraction for progress



Local Reasoning via Object Versions

- Per-object **version** sequence
 - Each task advances accessed objects to exactly next version
 - Versions provide a simpler/compact abstraction for progress



- Each object walks its version sequence, rather than reasoning about the huge global schedule space

Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?

Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?

Thread A

read version(x) = v

Thread B

Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?

Thread A

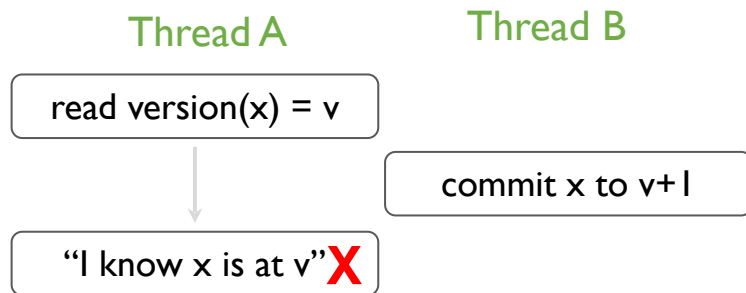
read version(x) = v

Thread B

commit x to v+1

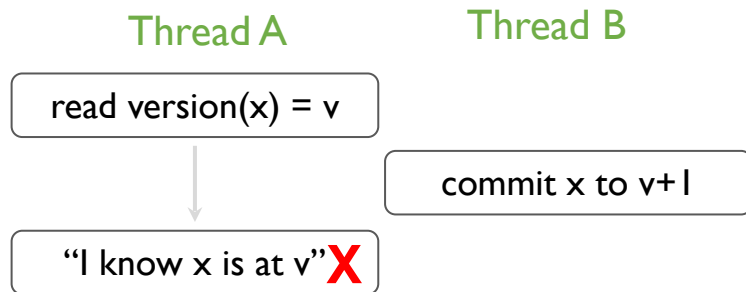
Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?



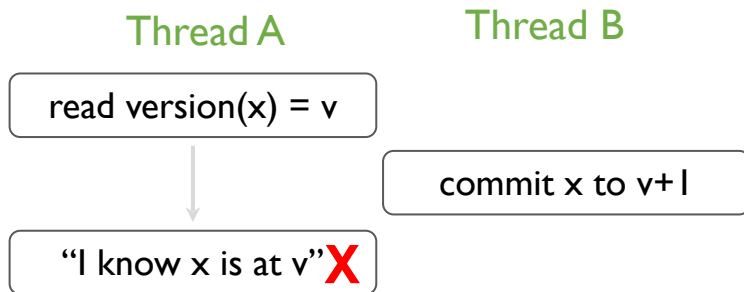
Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?
- DeterV represents this into two forms (a global truth + thread-local token/permit)



Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?
- DeterV represents this into two forms (a global truth + thread-local token/permit)



**Shared global
record**

(source of truth)
x is at v

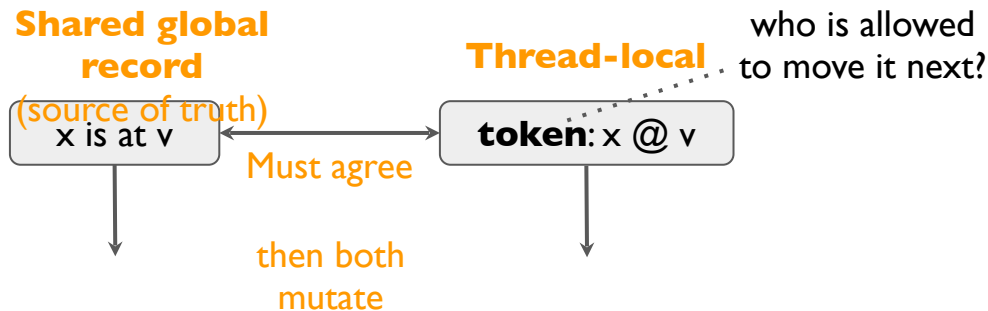
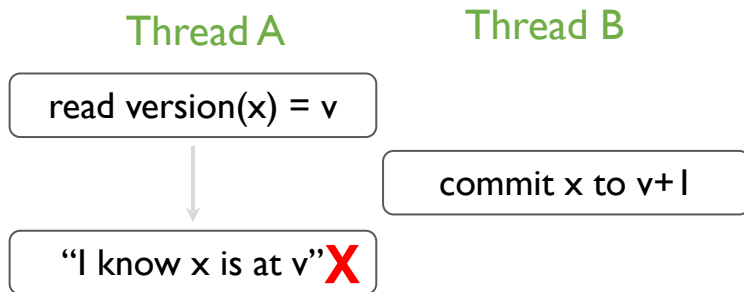
Thread-local

token: x @ v

who is allowed
to move it next?

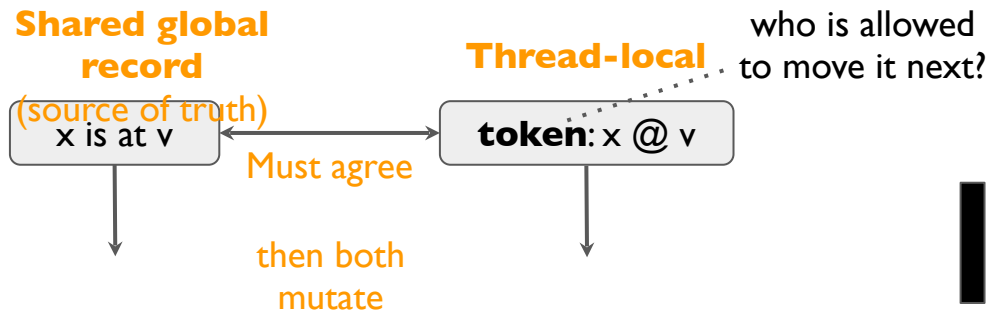
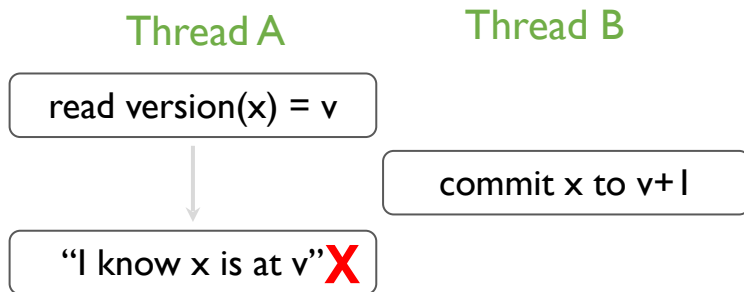
Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?
- DeterV represents this into two forms (a global truth + thread-local token/permit)



Manage Versions under Concurrency

- Versions themselves are shared mutable facts: who can safely say “x is at version v”?
- DeterV represents this into two forms (a global truth + thread-local token/permit)



Implemented as *Authoritative Resource Algebra* from Iris (POPL'15)

DeterV Design

A verified DPE runtime in Verus

Localize

each object follows its
own version order
(Insight 1)

Versions and safely
represent who may
advance a version

Enforce

Check that order at
one commit boundary
(Insight 2)



where do we check and
actually mutate the
version?

Compose

layered refinement that
composes local reasoning
into the global one

Enforcing Determinism Validation at One Effect Boundary

```
let res = task.run();  
commit_effect(res, state);
```

- **commit_effects**: tasks effects become visible, in which version order must hold

Enforcing Determinism Validation at One Effect Boundary

```
let res = task.run();  
commit_effect(res, state);
```

- **commit_effects**: tasks effects become visible, in which version order must hold
- A natural interface to integrate verification machinery - invariants checking, pre-/post-conditions

Enforcing Determinism Validation at One Effect Boundary

```
let res = task.run();  
commit_effect(res, state);
```

- **commit_effects**: tasks effects become visible, in which version order must hold
- A natural interface to integrate verification machinery - invariants checking, pre-/post-conditions

```
1 exec fn commit_effects<S: Store>(  
2   store:          &StoreHandle<S, ReadWrite>,  
3   writes:         &Vec<ObjId, S::Value>),  
4  
5  
6 )  
7  
8  
9  
10  
11 let applied: Map<ObjId, Version> = store.apply(writes);  
12  
13 }
```

Enforcing Determinism Validation at One Effect Boundary

```
let res = task.run();  
commit_effect(res, state);
```

- **commit_effects**: tasks effects become visible, in which version order must hold
- A natural interface to integrate verification machinery - invariants checking, pre-/post-conditions

```
1 exec fn commit_effects<S: Store>(  
2   store:          &StoreHandle<S, ReadWrite>,  
3   writes:         &Vec<ObjId, S::Value>,&br/>4   Tracked(inv):    Tracked<ObjInvariantsList>,  
5   Tracked(permits): Tracked<PermitsList>, Ghost  
6 ) → (advanced: Tracked<PermitsList>) state  
7  
8  
9  
10  
11 let applied: Map<ObjId, Version> = store.apply(writes);  
12  
13 }
```

Enforcing Determinism Validation at One Effect Boundary

```
let res = task.run();  
commit_effect(res, state);
```

- **commit_effects**: tasks effects become visible, in which version order must hold
- A natural interface to integrate verification machinery - invariants checking, pre-/post-conditions

```
1 exec fn commit_effects<S: Store>(  
2   store:          &StoreHandle<S, ReadWrite>,  
3   writes:         &Vec<ObjId, S::Value>,&br/>4   Tracked(inv):    Tracked<ObjInvariantsList>,  
5   Tracked(permits): Tracked<PermitsList>, Ghost  
6 ) → (advanced: Tracked<PermitsList>) state  
7  
8  
9  
10  
11 let applied: Map<ObjId, Version> = store.apply(writes);  
12 commit_effects_ghost(inv, permits, Ghost(applied))  
13 }
```

**Invariants
(matching)**

Enforcing Determinism Validation at One Effect Boundary

```
let res = task.run();  
commit_effect(res, state);
```

- **commit_effects**: tasks effects become visible, in which version order must hold
- A natural interface to integrate verification machinery - invariants checking, pre-/post-conditions

```
1 exec fn commit_effects<S: Store>(  
2   store:          &StoreHandle<S, ReadWrite>,  
3   writes:         &Vec<ObjId, S::Value>,&br/>4   Tracked(inv):    Tracked<ObjInvariantsList>,  
5   Tracked(permits): Tracked<PermitsList>, Ghost  
6 ) → (advanced: Tracked<PermitsList>) state  
7 requires  
8   forall|o| permits_match_inv(permits[o], inv[o]), Pre/Post  
9 ensures Cond  
10  forall|o| advanced@[o].v == permits[o].v + 1, {  
11    let applied: Map<ObjId, Version> = store.apply(writes);  
12    commit_effects_ghost(inv, permits, Ghost(applied))  
13 }
```

**Invariants
(matching)**

DeterV Design

A verified DPE runtime in Verus

Localize

each object follows its
own version order

(Insight 1)

Enforce

Check that order at
one commit boundary

(Insight 2)

Compose

layered refinement that
composes local reasoning
into the global one



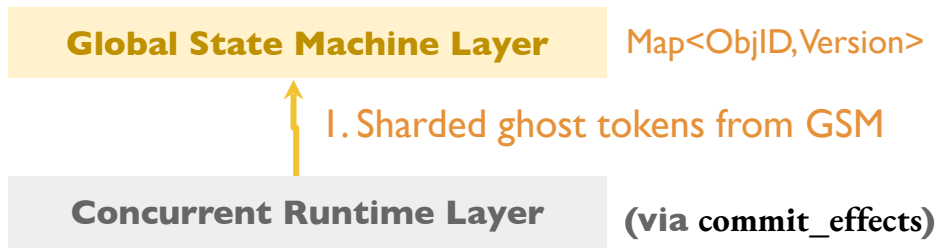
Compose Local Version Steps into Global Determinism

Concurrent Runtime Layer

(via `commit_effects`)

threads execute concurrently;
`commit_effects` is where shared state changes

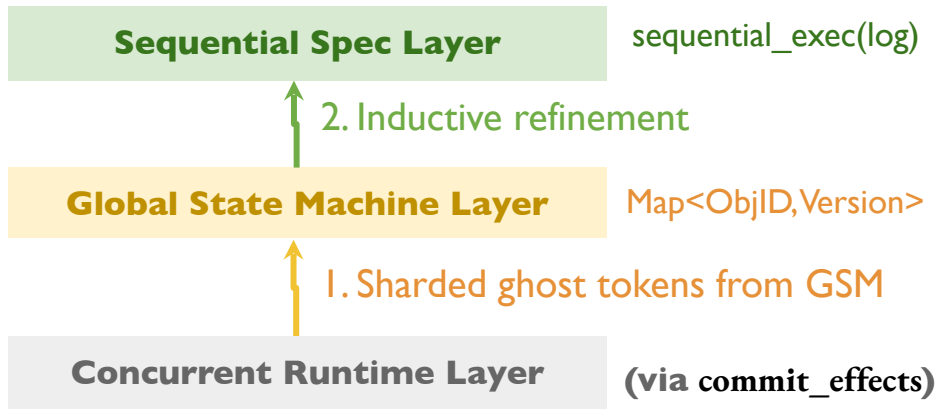
Compose Local Version Steps into Global Determinism



“Forget the schedule”; keep only each obj’s committed version.

threads execute concurrently;
`commit_effects` is where shared state changes

Compose Local Version Steps into Global Determinism



pointwise equality indicates the final state equivalence

“Forget the schedule”; keep only each obj’s committed version.

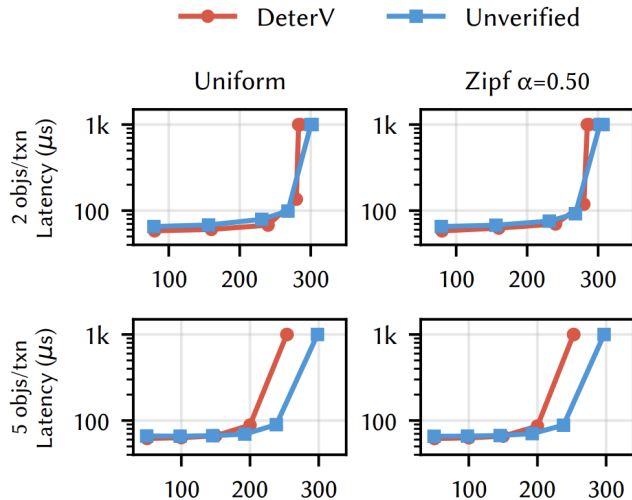
threads execute concurrently;
`commit_effects` is where shared state changes

Implementation & Evaluation

- 3:1 proof-code ratio in Verus
- Only verify safety, not liveness
- No async support
 - thread pool -> crossbeam-channel + std::thread
 - Notify -> Mutex and CondVar

Implementation & Evaluation

- 3:1 proof-code ratio in Verus
- Only verify safety, not liveness
- No async support
 - thread pool -> crossbeam-channel + std::thread
 - Notify -> Mutex and CondVar
- No latency penalty, less than 20% throughput drop due to the heavier signal impl



Takeaways

- Deterministic parallel execution boosts the replication performance, while leaving the correctness unchecked.
- DeterV presents localize-then-compose reasoning which can simplify the verification, instead of global reasoning
- Effects commitment provides a natural boundary and a reusable interface to hook in verification

Takeaways

- Deterministic parallel execution boosts the replication performance, while leaving the correctness unchecked.
- DeterV presents localize-then-compose reasoning which can simplify the verification, instead of global reasoning
- Effects commitment provides a natural boundary and a reusable interface to hook in verification

Scofield Zhengqing Liu 



scofield.liu@imperial.ac.uk