VLDB2024
GUANGZHOU

# Spectrum: Speedy and Strictly-Deterministic Smart Contract Transactions for Blockchain Ledgers


***Zhihao Chen***, Tianji Yang, Yixiao Zheng, Zhao Zhang,
Cheqing Jin, Aoying Zhou

East China Normal University


Presented at VLDB 2024

華東師範大學
EAST CHINA NORMAL UNIVERSITY

SCHOOL OF DATA SCIENCE & ENGINEERING
数据科学与工程学院

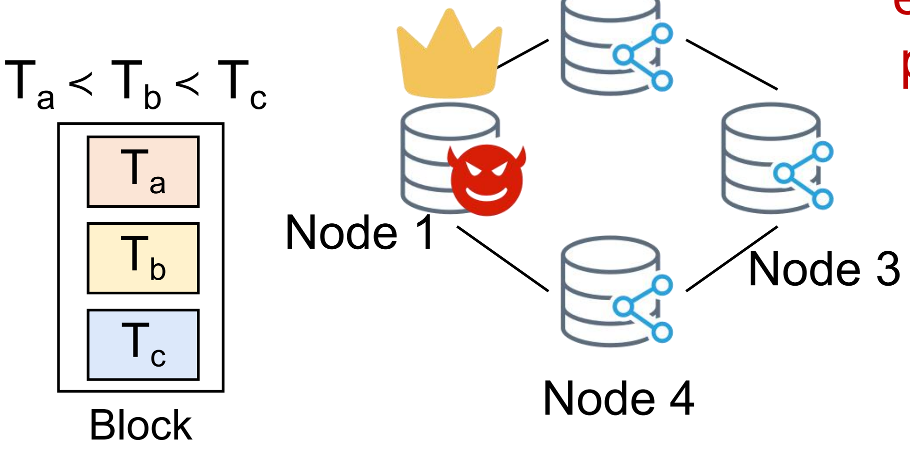
# Outline

- Motivation
- Background
- Goals & Contributions
- Methodology
- Evaluation
- Conclusion & Future work

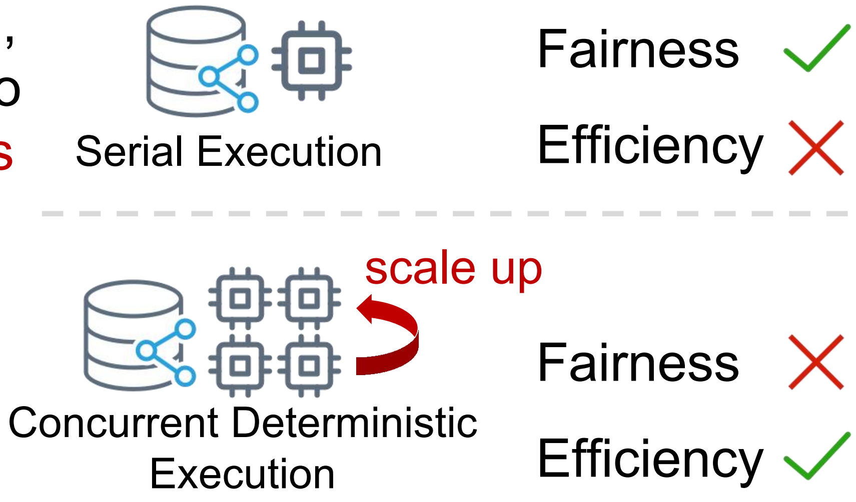
# Motivation

- Unlike tranditional databases, blockchain ledgers concern ordering fairness
- Existing deterministic execution schemes fail to preserve both ordering fairness and high performance

Prevents transaction ordering manipulation




Given the fair ordering, execution is required to preserve such fairness (in the OE paradigm)




Modern Byzantine consensus, e.g., Pompe [OSDI' 20]、Themis [CCS' 23] incorporate fairness designs

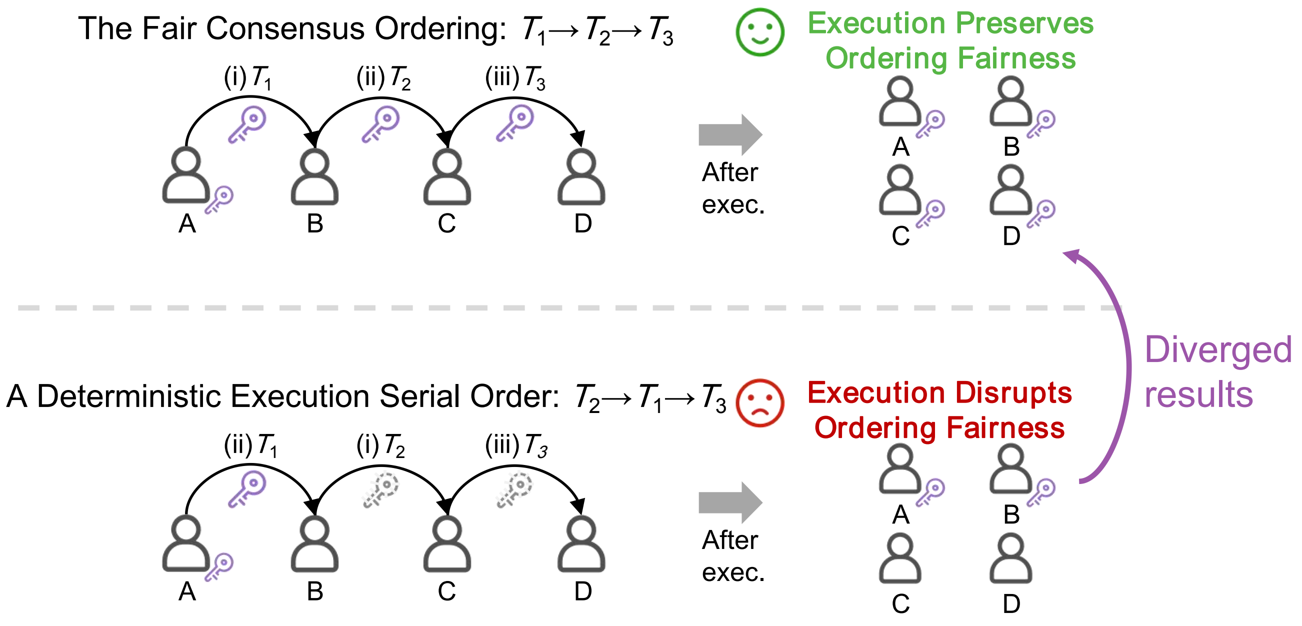
# Motivation

- They merely guarantee deterministic serializability, which can lead to a deterministic (consistent across replicas), yet different serial order than the agreed-upon ordering

e.g., AccessControl contract: A caller with access permission can grant it to a specified address

```
contract AccessControl {
    mapping(address => bool) public access;
    function grantAccess(address to) public {
        if (access[msg.sender] != false)
            access[to] = access[msg.sender];
    }
}
```

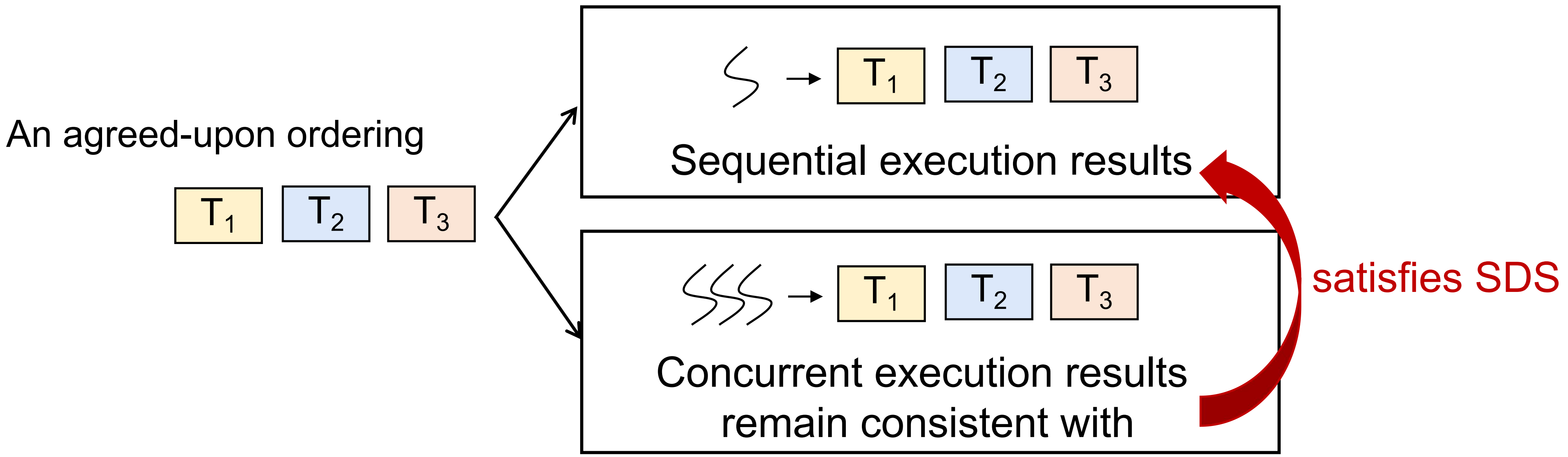
Initially, only A has permission; $T_1$: A->B, $T_2$: B->C, $T_3$: C->D




Merely Determinism Does NOT Preserve Ordering Fairness

# Background

- Problem Definition: Strictly-Deterministic Serializability (SDS)
  - Given an agreed ordering of transactions, *O*: {$T_1$, ..., $T_n$}, an execution schedule of transactions *S* satisfies **strictly-deterministic serializability** iff its effect is equivalent to the sequential execution of *O*, which adheres to the transactions commit order, {$T_1$, ..., $T_n$}



# Background

- For blockchain ledgers, ensuring SDS in an execution scheme preserves ordering fairness
- But most Deterministic Concurrency Control (DCC) schemes fail to ensure SDS when processing smart contracts with runtime-determined accesses

Kindly refer to the paper for an in-depth analysis

| Deterministic Execution Schemes | Execution Paradigm | No Complete R/W Sets | Strict Determinism | Scaling in Contention |
|---|---|---|---|---|
| ▲ HyperLedger Fabric [10] | EOV | ✓ | ✗ | ✗ |
| ▲ Fabric(++, #) [47, 49] | EOV | ✓ | ✗ | ✓ |
| ▲ NeuChain [40] | EV | ✓ | ✗ | ✗ |
| • Calvin [55], PWV [22] | OE | ✗ | ✗ | ✓ |
| • Bohm [23], Caracal [45] | OE | ✗ | ✗ | ✓ |
| • QueCC [43] | OE | ✗ | ✗ | ✓ |
| • Aria, AriaFB [38] | OE | ✓ | ✗ | ✓ |
| • Sparkle [36] | OE | ✓ | ✓ | ✗ |
| ▲ Primary-Follower [29] | OE | ✓ | ✗ | ✓ |
| ▲ SSI [39], OCC-DA [24] | OE | ✓ | ✗ | ✗ |
| ▲ OXII [9], PEEP [15] | OE | ✗ | ✗ | ✓ |
| ▲ Harmony [33] | OE | ✓ | ✗ | ✓ |
| ▲ **Spectrum (Ours)** | OE | ✓ | ✓ | ✓ |

• Scope of Databases ▲ Scope of Blockchains

Deterministic Concurrency Control (Database)

Calvin [SIGMOD' 12] | Orthrus [SIGMOD' 16] | Sparkle [DSN' 19] | QStore [EDBT' 20] | Caracal [SOSP' 21]

Bohm [VLDB' 15] | PWV [VLDB' 17] | QueCC [Middleware' 18] | Aria [VLDB' 20] | Lotus [VLDB' 22]

Before 2020 → 2021 → 2022 → 2023 → 2024

Parallelizing Transactional Smart Contracts (Blockchain)

Fabric & Fabric(++,#) [EuroSys' 18] [SIGMOD' 19, 20] | Block-SSI [VLDB' 19] | SChain [VLDB' 21, ICDE'23] | NeuChain [VLDB' 22] | Harmony [SIGMOD' 23] | **Spectrum (Ours)**

OXII [ICDCS' 19] | Primary-Follower [TKDE' 21] | OCC-DA [ICSE' 22] | Block-STM [PPoPP' 23]

# Background

- Quasi-Turing-complete smart contracts have runtime-determined access patterns

```
pragma solidity >=0.8.2 <0.9.0;
contract MutableRW {
  mapping(uint256 => uint256) public store;
  function add(uint256 key0, uint256 key1) public {
    uint256 bal0 = store[key0];
    uint256 bal1 = store[key1];
    if (bal0 < bal1) store[key0] += bal1;
    else store[key1] += bal0;
  }
}
```

$T_i$ —invoke→ add(key0, key1) —branch→ bal0 < bal1 ?

| Branch 1 | | Branch 2 |
|---|---|---|
| $Snapshot_1$ → (bal0 < bal1) | | $Snapshot_2$ → (bal0 >= bal1) |
| $T_i$'s R_Set: { store[key1] } | ⟷ Mutable R/W Sets | $T_i$'s R_Set: { store[key0] } |
| $T_i$'s RW_Set: { store[key0] } | | $T_i$'s RW_Set: { store[key1] } |

A Transaction with mutable read/write sets

Runtime-determined nature

↓

whose read/write sets can vary across different snapshots
( Mutable read/write sets )

↓

Most DCC schemes CAN NOT guarantee SDS when handling mutable r/w sets

For inaccurate pre-acquisition or inherent scheme limitations (reordering, violations, etc.)

# Goals & Contributions

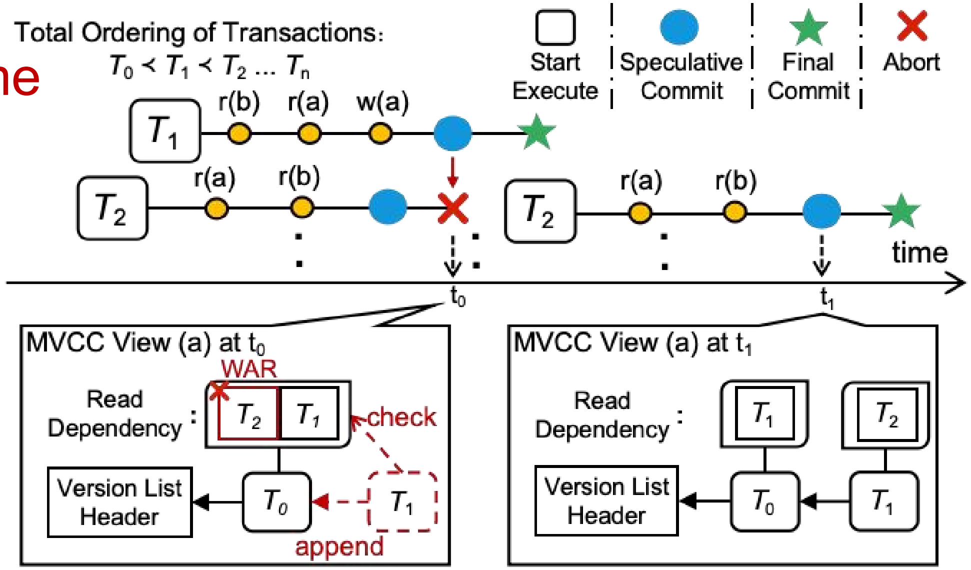
- Design goal
  - Spectrum, a DCC scheme that ensures both strict determinism and high performance (across diverse workloads) for blockchain ledgers

- Key points and contributions
  - Leverages speculative execution to ensure SDS for concurrent smart contract transactions
  - Proposes a partial rollback mechanism with efficient impl.
  - Designs a predictive transaction scheduling method
  - Evaluates by running EVM-Based smart contracts on YCSB, SmallBank and TPC-C alike benchmarks

Two novel optimizations to maintain high performance under contention

# Methodology

- #1 Speculative transaction execution
  - Multi-Version Concurrency Control with runtime conflict detection
  - Lets each thread independently execute Txns in speculation => high inter-thread concurrency
  - Transaction lifecycle: 1) Execution; 2) Speculative Commit; 3) Final Commit
  - Detects and aborts any order-violating Txns at runtime
  - Re-executes them with their original seq. numbers
  - Upholds SDS

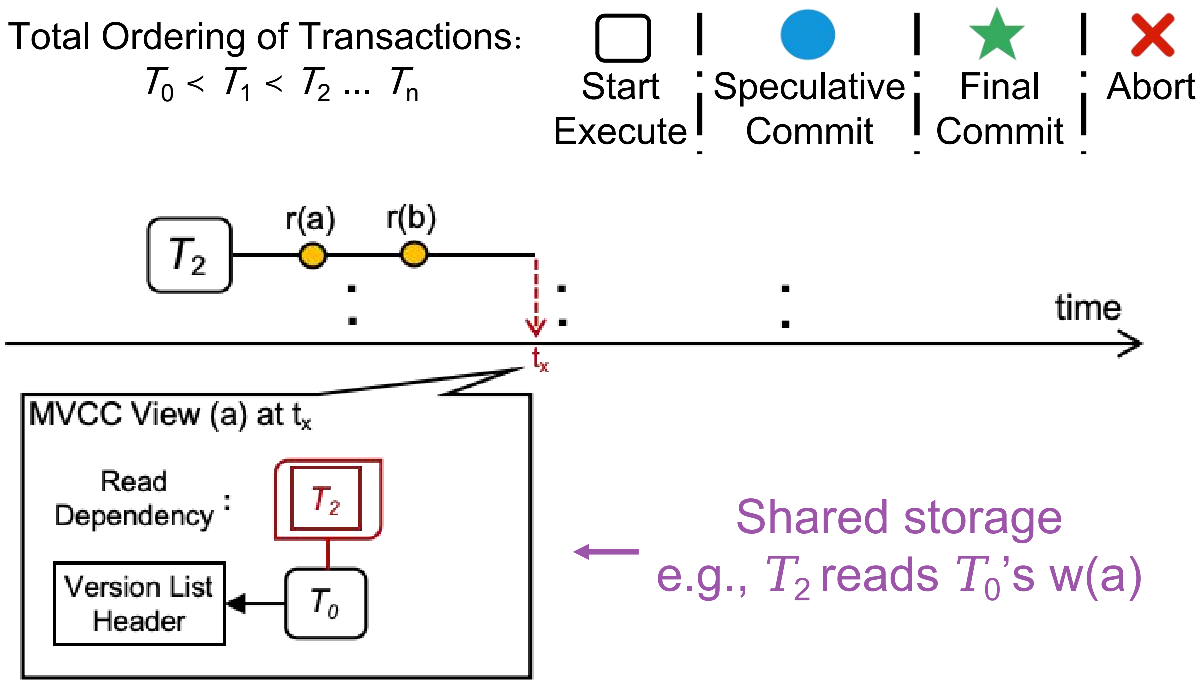


Speculative transaction execution

# Methodology

- #1 Speculative transaction execution
  - Transaction lifecycle: 1) Execution; 2) Speculative Commit; 3) Final Commit

1) Execution: runs OPs, reads from shared storage (if not found locally), writes to $T$'s local storage




Speculative transaction execution

Read operations:
$T$'s local storage -> shared storage

(reads the largest preceding version before $T$ and records read dep.)

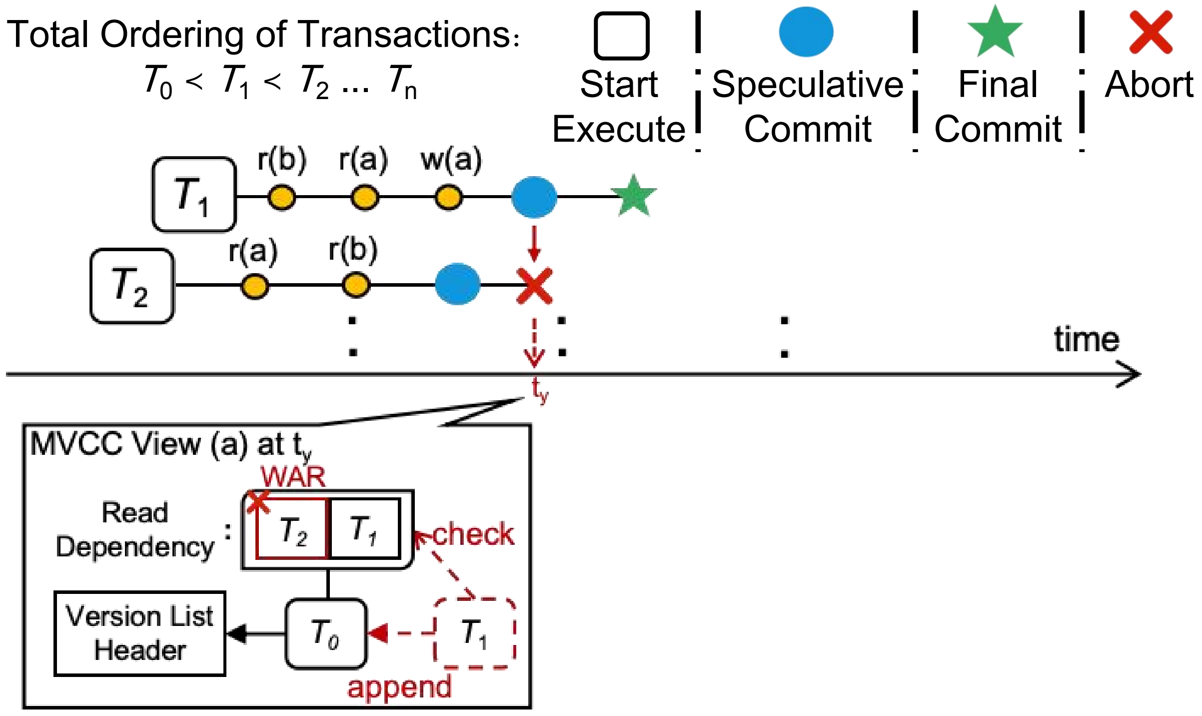
Enables aborting $T$ if a conflict is detected later

Write operations:
merely inserts to local storage

# Methodology

- #1 Speculative transaction execution
  - Transaction lifecycle: 1) Execution; 2) Speculative Commit; 3) Final Commit

2) Specuative Commit: makes writes visible & detects conflicts & aborts order-violating Txns




Speculative transaction execution

Inserts all local writes to shared storage

Detects missed reads on newly-inserted writes

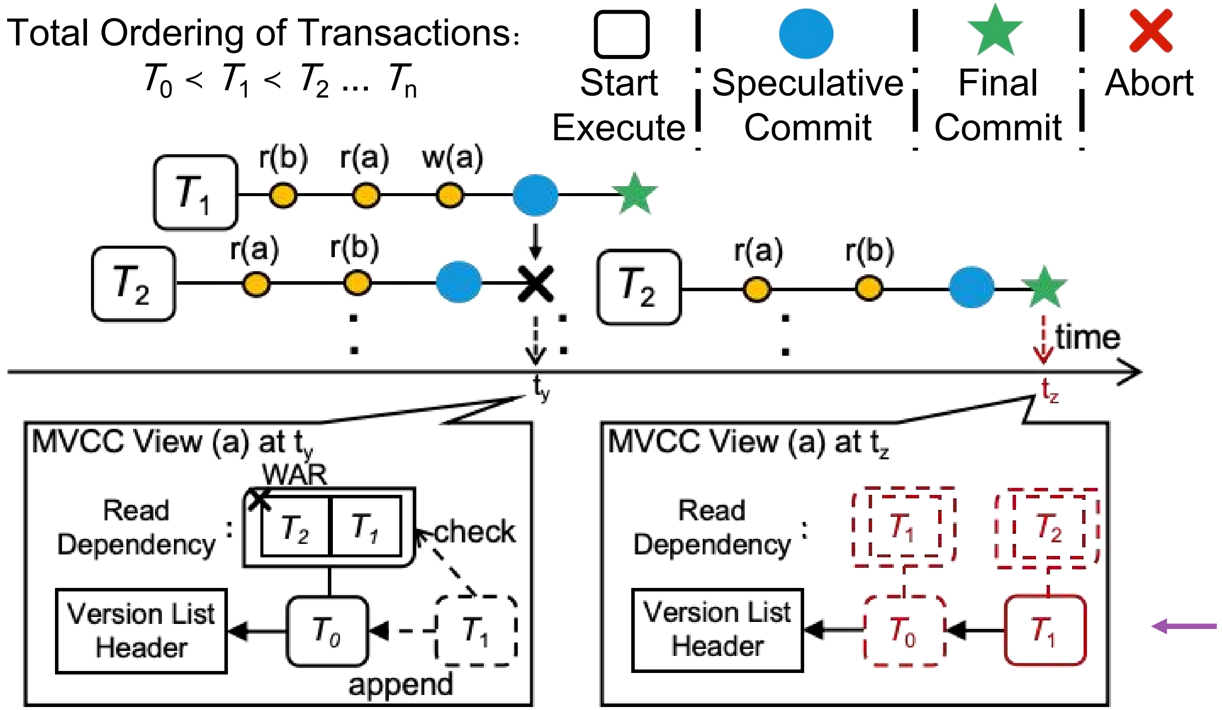
e.g., $T_2$'s read missed $T_1$'s w(a)

Aborts those order-violating Txns

# Methodology

- #1 Speculative transaction execution
  - Transaction lifecycle: 1) Execution; 2) Speculative Commit; 3) Final Commit

3) Final Commit: updates the tracking counter, cleans redundant versions & deps




Speculative transaction execution

FC condition (i): all Txns preceding $T$ have been finally committed

FC condition (ii): $T$ remains unaborted

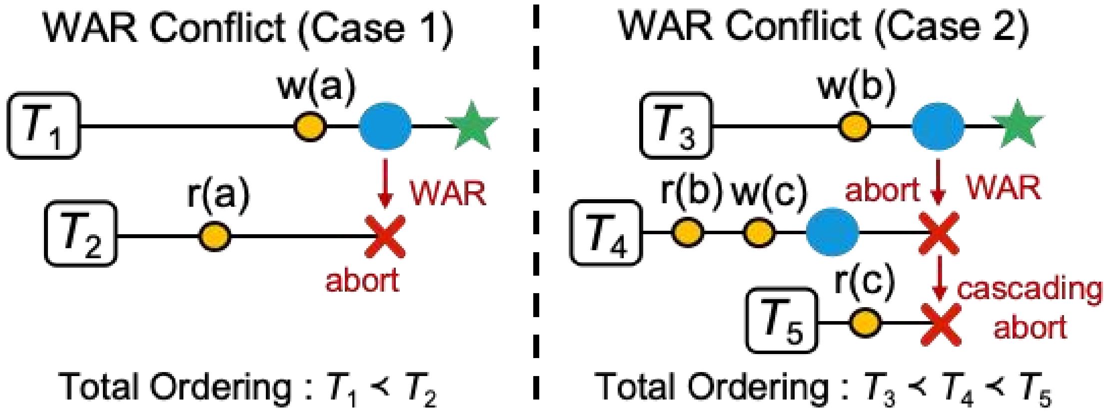
Promises $T$ no longer to be aborted

← After $T_1$, $T_2$ finally committed

# Methodology

- Conflict types
  - Mis-speculations lead to write-after-read (WAR) conflict (but no WAW or RAW conflicts)
  - WAR conflict: Txn $T_i$'s write $W_i$ (visible til $T_i$'s spec. commit) is missed by $T_j$'s read $R_j$, where $W_i$ conflicts with $R_j$ and $i < j$
  - Become scaling bottlenecks under contended workloads

both the **Overhead** and the **Number** of mis-speculations




Exemplifying the WAR Conflict

# Methodology

- #2 Fine-Grained transaction rollback

Traditionally: Complete ↓ Complete ↓

  - When a WAR conflict occurs, the aborted transaction(s) need rollback and re-execute

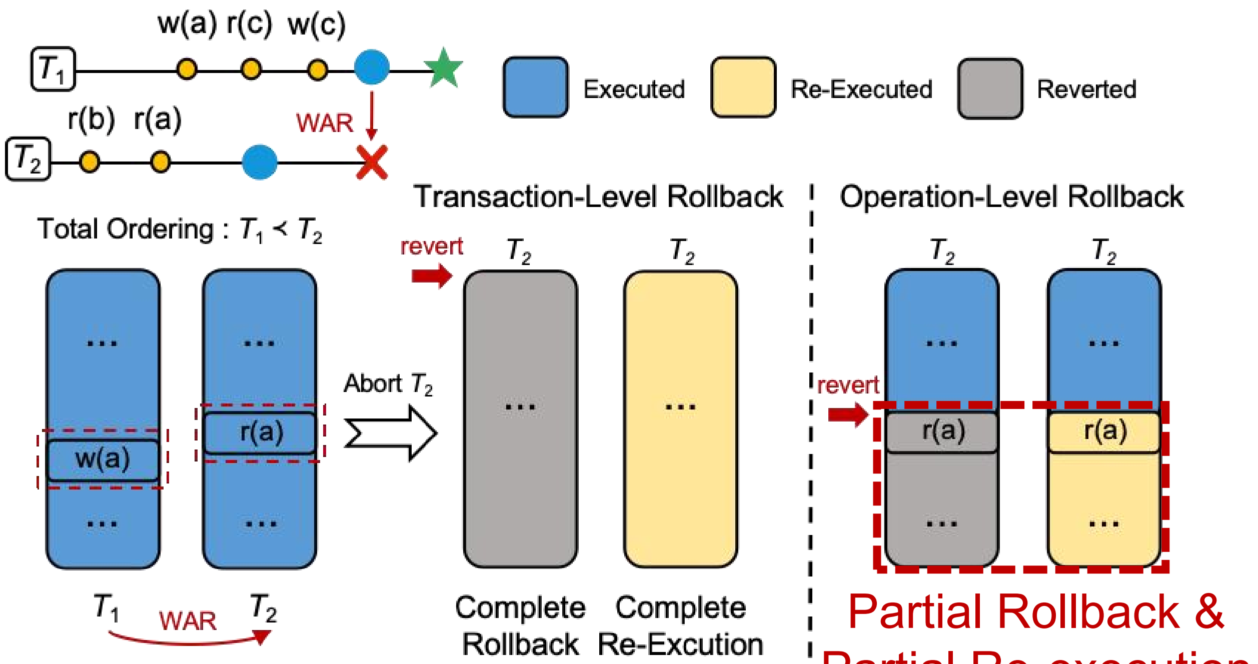
Expensive and wasteful for smart contract Txns !!!

  - Partial rollback mechanism:

Only the operations impacted by the conflict need to be rollbacked and re-executed

  - Avoids wasting CPU resources & Saves re-execution overhead

Reduces the overhead per mis-speculation



Comparison of transation- and operation-level rollback

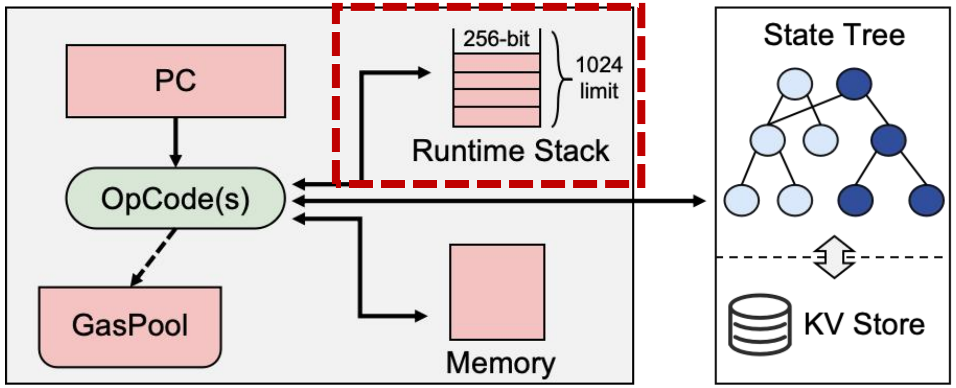
# Methodology

- Implementing partial rollback in stack-based EVM
  - pStack, a persistent stack with Copy-on-Write (CoW) mechanism <=> replacing EVM's Stack
  - Efficiently restores and operates on multiple stack snapshots, (i.e., checkpoints)

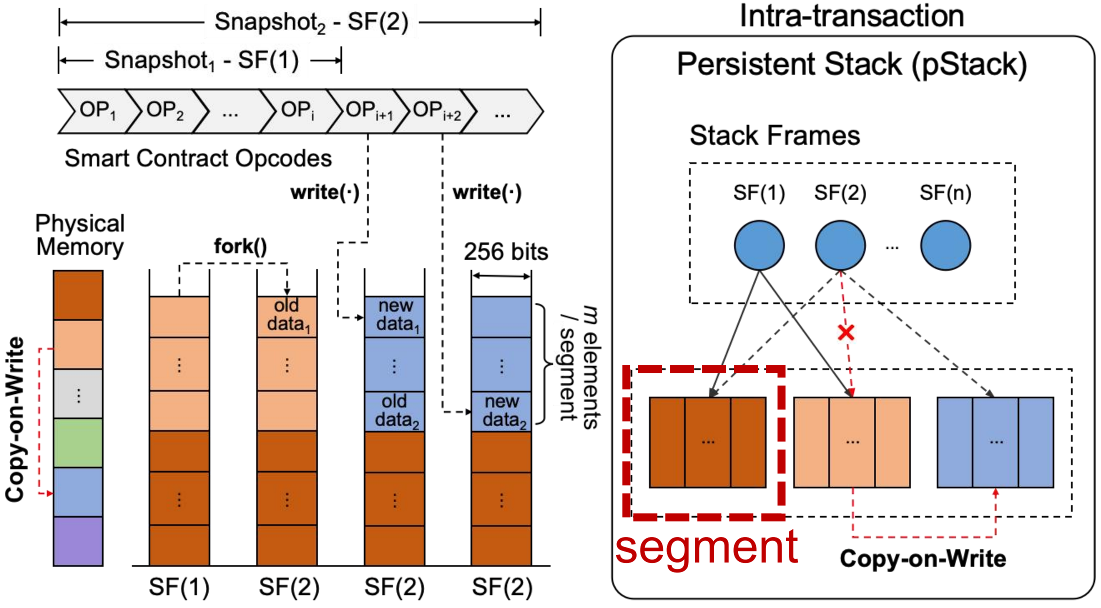
CoW at the 'segment' level (consecutive stack elements)

Exposes three primitives: 1. write(elementIndex, value); 2. fork(); 3. rollback(snapshotID)

Replaces EVM's Stack with pStack for efficient checkpointing



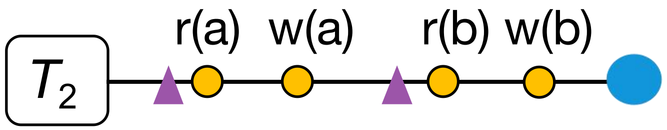

Ethereum Virtual Machine (EVM) structure




pStack-based stack frame management

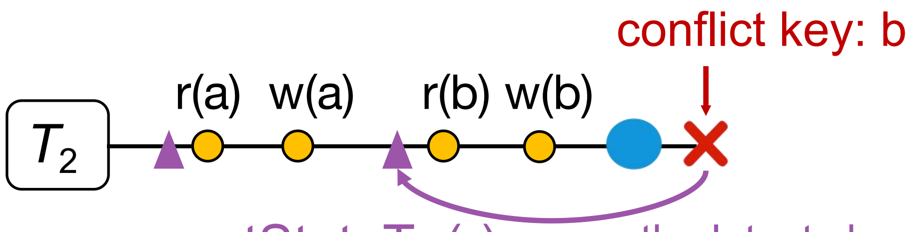
# Methodology

- Integrates partial rollback implementation with speculative execution
- Makes a checkpoint before an external read, i.e., SLOAD with the read not in local storage
- Rollbacks to a checkpointed state right before the conflict occurs
- Low checkpoint memory cost & cache-friendly




▲ makeCheckpoint(·):

- EVM.pStack.fork()
- Records PC, Memory
- EVM.addCheckpoint(·)



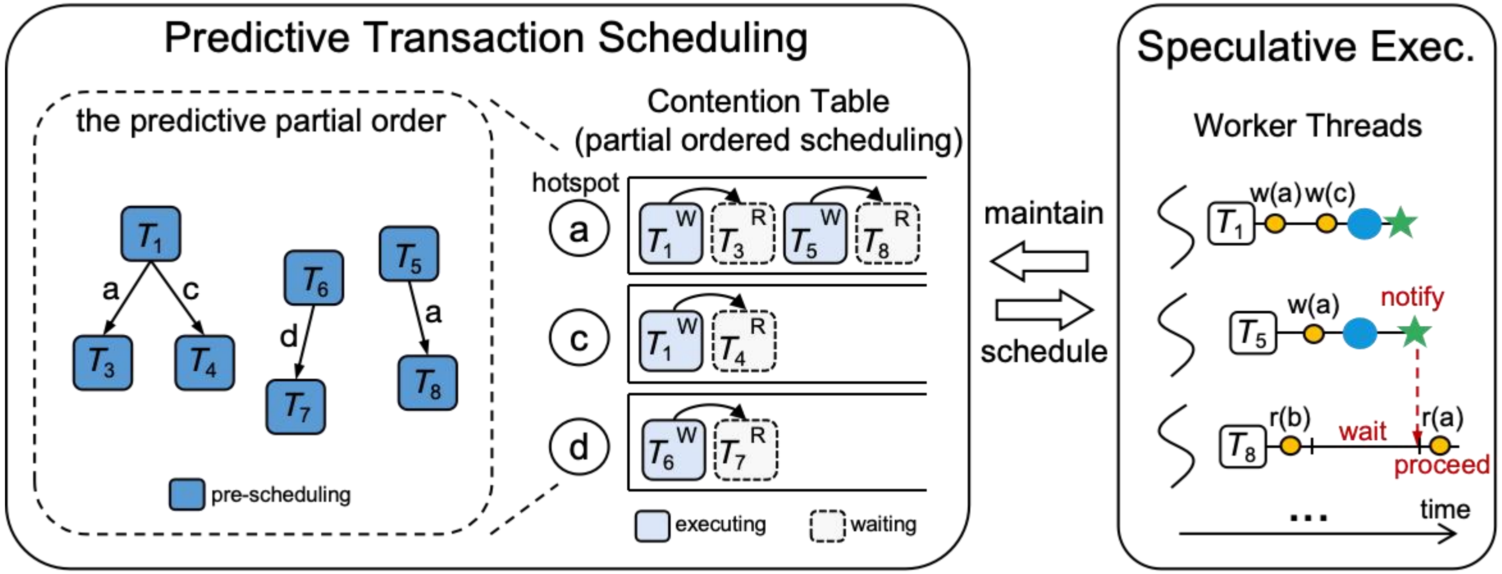

revertStateTo(·)

the latest checkpoint prior to any of the conflicts

- EVM.getCheckpoint(·)
- Sets PC, Memory
- EVM.pStack.rollback(·)

# Methodology

- #3 Predictive transaction scheduling ← serving as 'hints'
  - Only requires partial a-priori read/write sets to pre-schedule potentially conflicting transactions
  - If the prediction is correct, it effectively reduces the number of runtime mis-speculations
  - Independent scheduling across replicas (ensures SDS despite different predictions)




Predictive scheduling optimization

# Methodology

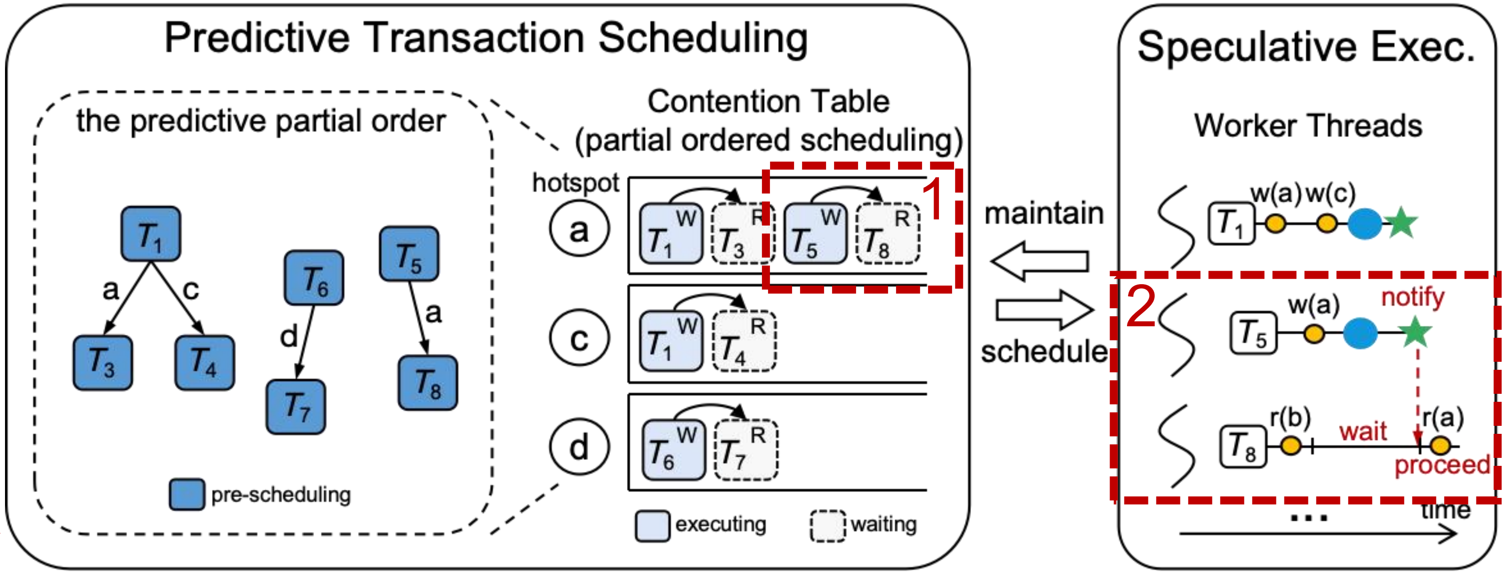
- Key intution: wait for preceding conflicting Txns to be finally committed => avoid WAR conflicts
- Sched. Algorithm (efficiency and correctness): concurrent maintenance and sequential granting
- Schedules only highly-conflicted keys and predicted WAR conflicts
- Superior parallelism than Calvin's single-threaded ordered lock pre-scheduling

2. Runtime scheduling

Example:
pre-acquired rw_set

$T_5$ : w(a)
$T_8$ : r(a)

1. Pre-maintenance




Schedules reads, finer than Txns

$T_8$'s r(a) waits for $T_5$'s final commit

Predictive scheduling optimization

# Evaluation

- Testbed
  - 2x Intel Xeon Gold 6330 CPU (28C56T each) with 256GB DRAM
- Benchmarks
  - YCSB & SmallBank & TPC-C alike smart contracts (written in Solidity)
  - Uniform workload and Skewed workload (controlled by *Zipf*)
- Baselines (incl. SOTA DCC schemes)
  - Serial
  - Calvin
  - Aria, AriaFB
  - Sparkle

| Benchmark | Transaction Type (# of reads/writes) | call ratio |
|---|---|---|
| YCSB | Get&Set (5r5w) | 100% |
| SmallBank | Six standard functions (2r0w, 2r1w, 1r1w * 2, 2r2w * 2) | equal probability |
| TPC-C | NewOrder, Delivery, Payment (4r5w/orderline, deliver 10 orderlines, 1r1w) | N:D:P = 11:11:1 |

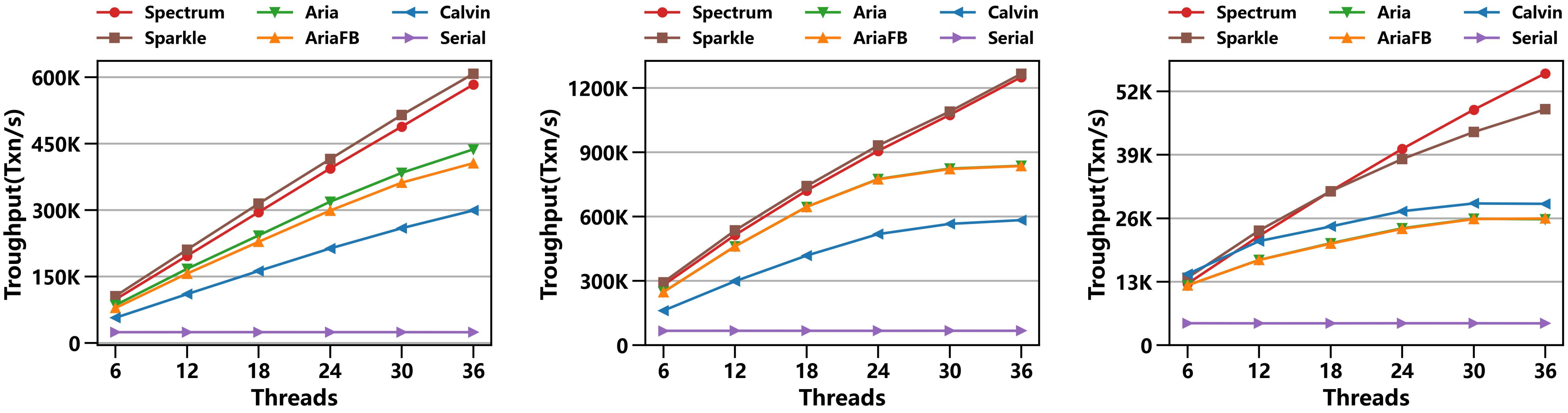
**Kindly refer to our paper for further impl. and experiments**

# Evaluation

- Throughput of varying threads

  superior parallelism, no locking and phase synchronization

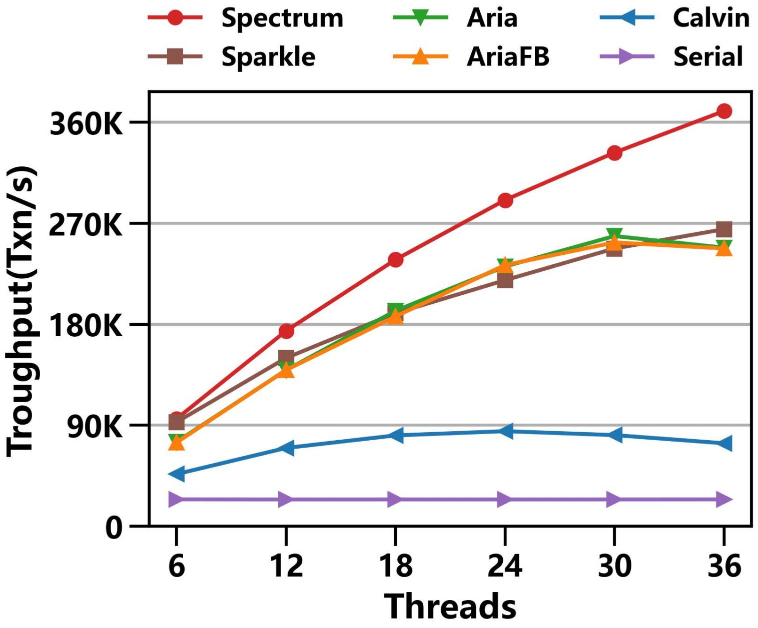
  - Under the uniform workload, both Spectrum and Sparkle scale well within 36 threads, and outperform Aria (1.3x~2.1x), AriaFB (1.4x~2.1x), Calvin (1.7x~1.9x) and Serial (12.7x~24.2x)



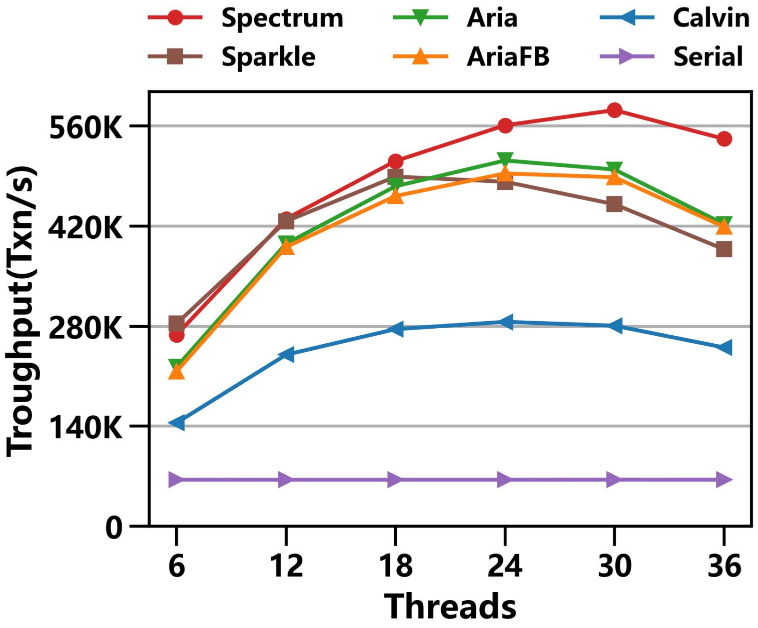

YCSB (Uniform) SmallBank (Uniform) TPC-C (10 orderlines/order)

# Evaluation

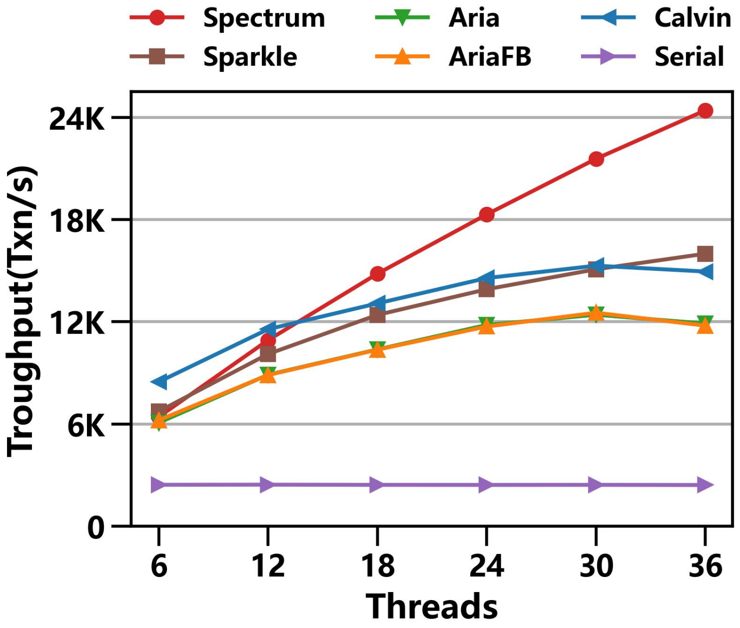
- Throughput of varying threads
  - Under the skewed workload, Spectrum achieves the highest peak throughput (only applies partial rollback)
  - Up to 1.6x, 2.1x and 5.0x higher throughput than Sparkle, Aria, and Calvin




YCSB (Zipf = 0.9)



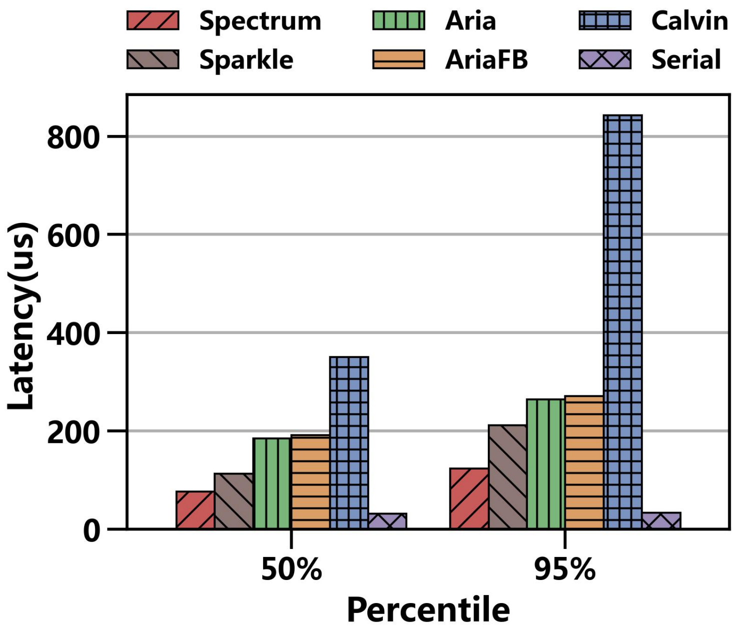

SmallBank (Zipf = 1.1)



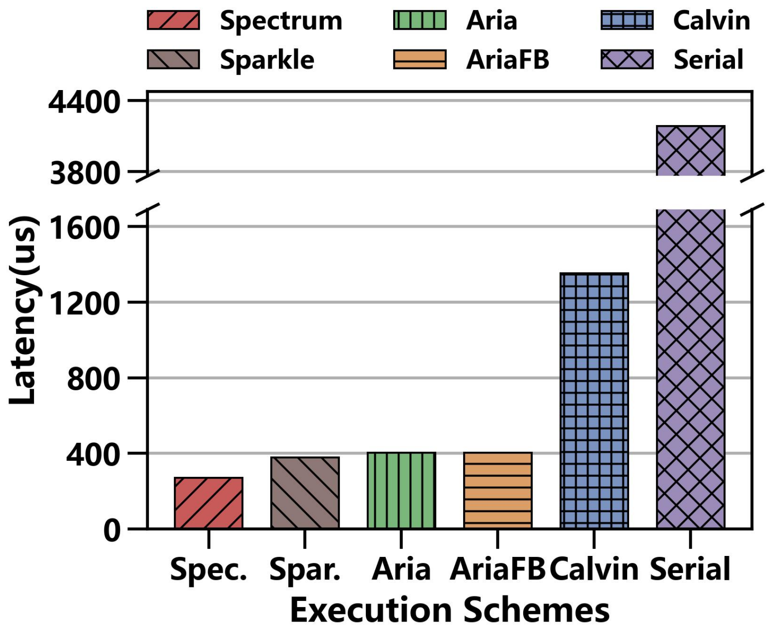

TPC-C (20 orderlines/order)

# Evaluation

- Transaction and Block latency
  - Serial has the lowest tx latency, then Spectrum, Sparkle, Aria variants, and Calvin (the highest)
  - Spectrum realizes the lowest block latency, reducing it by 28.3%, 32.8%, 33.1%, 80.0%, and 93.5% compared to Sparkle, Aria, AriaFB, Calvin, and Serial, respectively




p50, p95 Tx latency (YCSB, Zipf = 0.9)



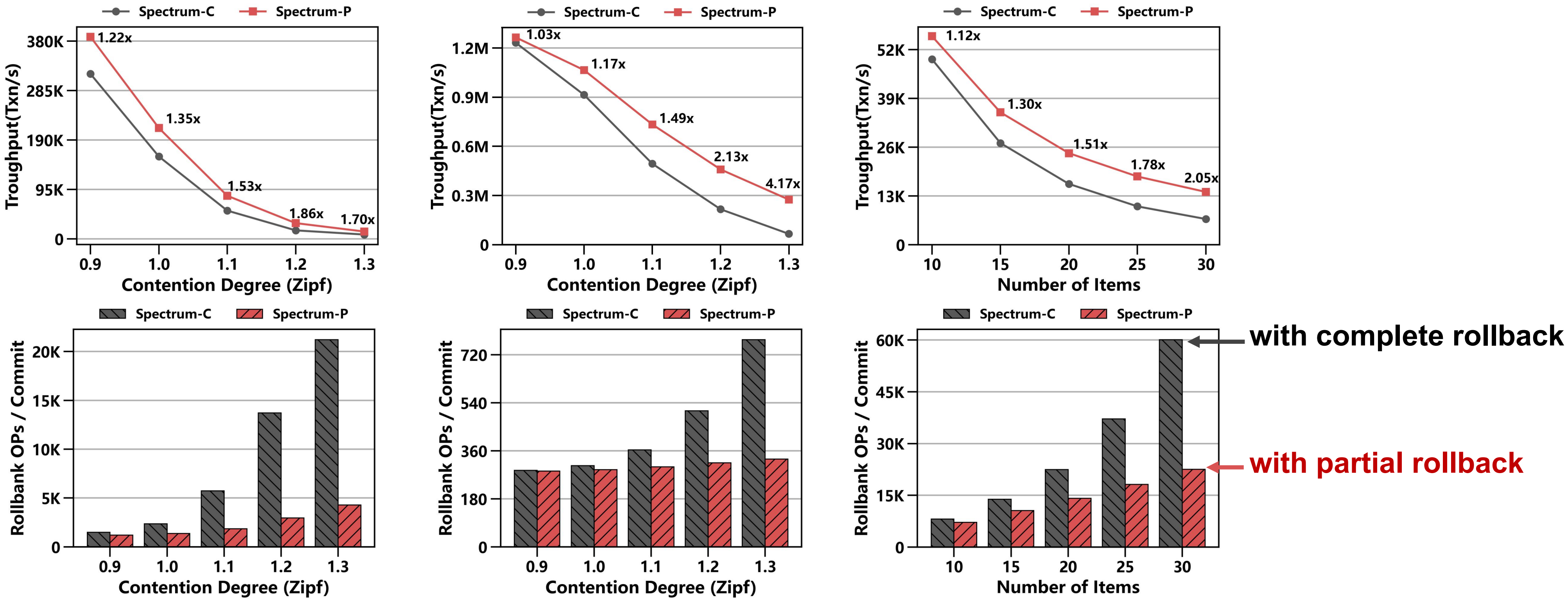

Block latency (YCSB, Zipf = 0.9)

# Evaluation

- Evaluation of partial rollback (re-execution overhead)
  - Partial rollback effectively reduces the number of operations to be rollbacked and re-executed, by 79.8%(YCSB), 57.6%(Smallbank) and 64.9%(TPC-C), compared to complete rollback

Scheme throughput in varying Zipf

Rollback operations per committed tx



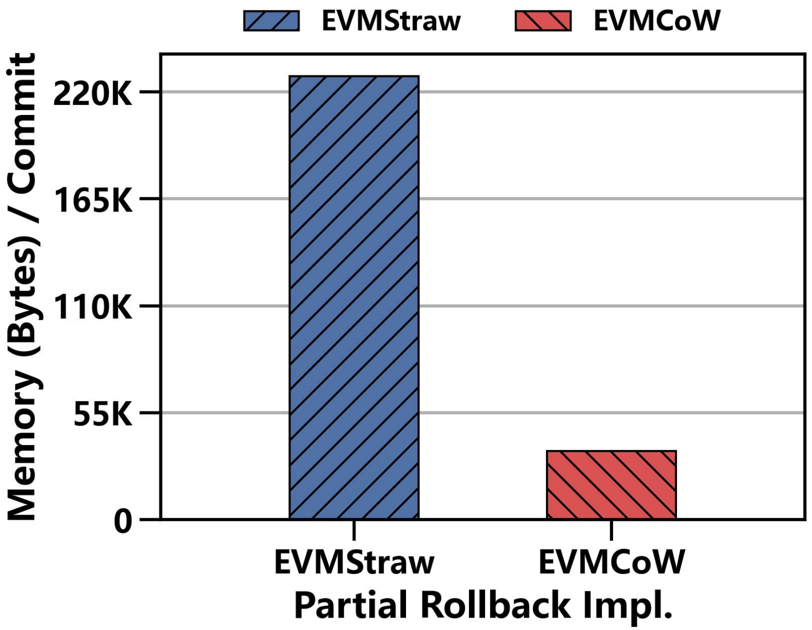

YCSB SmallBank TPC-C

# Evaluation

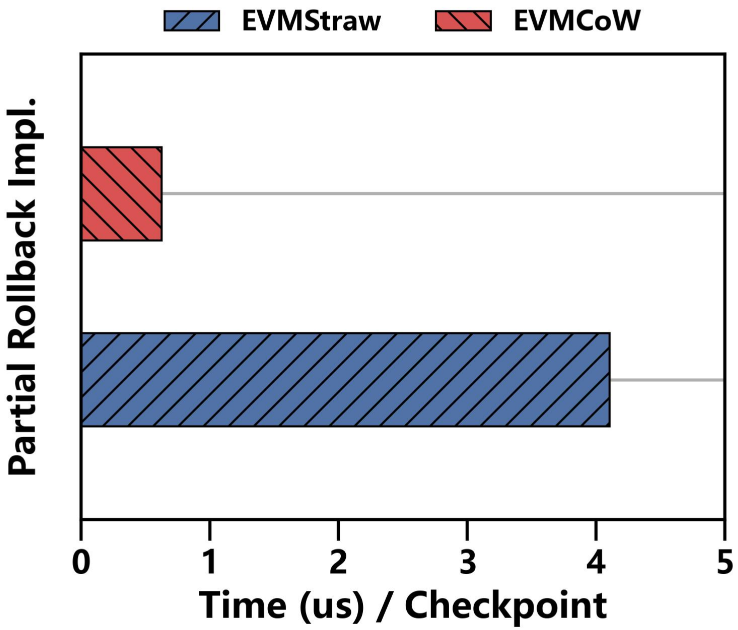
**strawman: copies all stack elements when checkpointing**
**pStack: uses Copy-on-Write mechanism for checkpointing**

- Evaluation of partial rollback (efficiency of pStack)
  - pStack saves 84.8% checkpointing memory cost and largely reduces creation time
  - pStack enhances throughput by 31.2% compared to the strawman approach

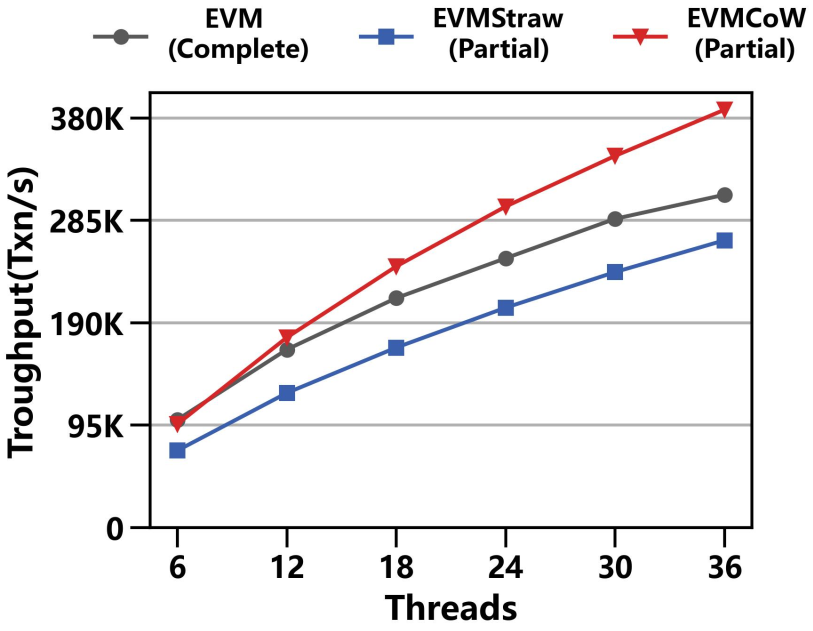
efficient impl. and cache-friendness



Memory Cost



Checkpointing Time Cost



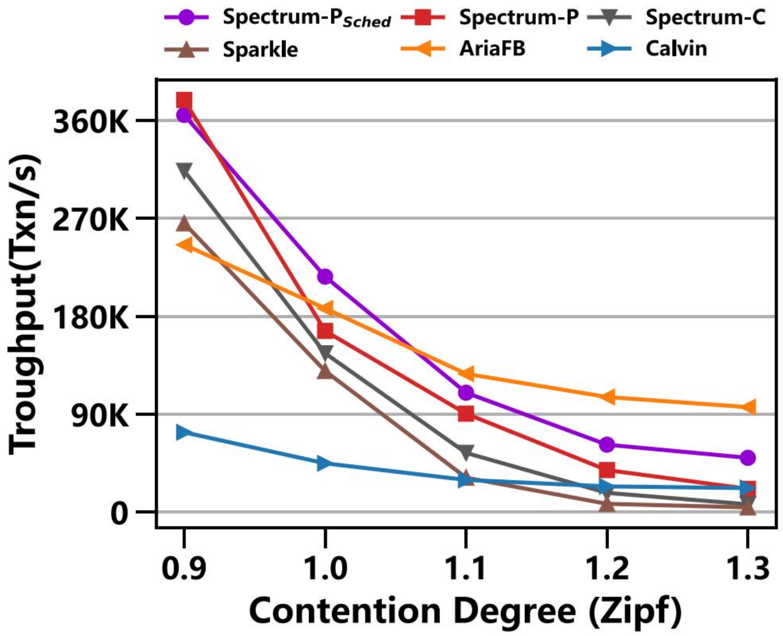
Integrated YCSB Throughput

# Evaluation

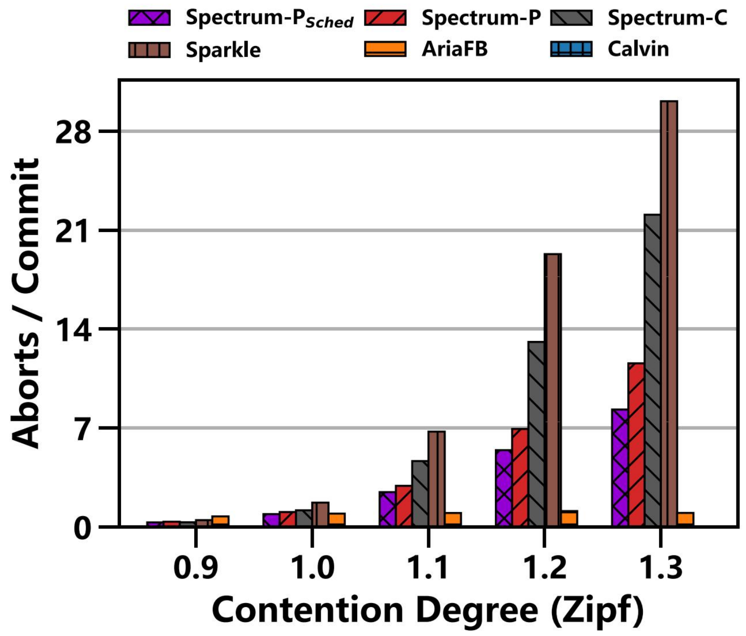
- Evaluation of predictive scheduling

Spectrum-$P_{Sched}$ implements both predictive scheduling and partial rollback
Spectrum-P only applies partial rollback
Spectrum-C employs neither of these optimizations

  - Spectrum-$P_{Sched}$ outperforms Spectrum-P by 2.3x and Spectrum-C by 6.8x
  - Predictive scheduling further reduces aborts (number of mis-speculations) by 28.2% to 62.3%



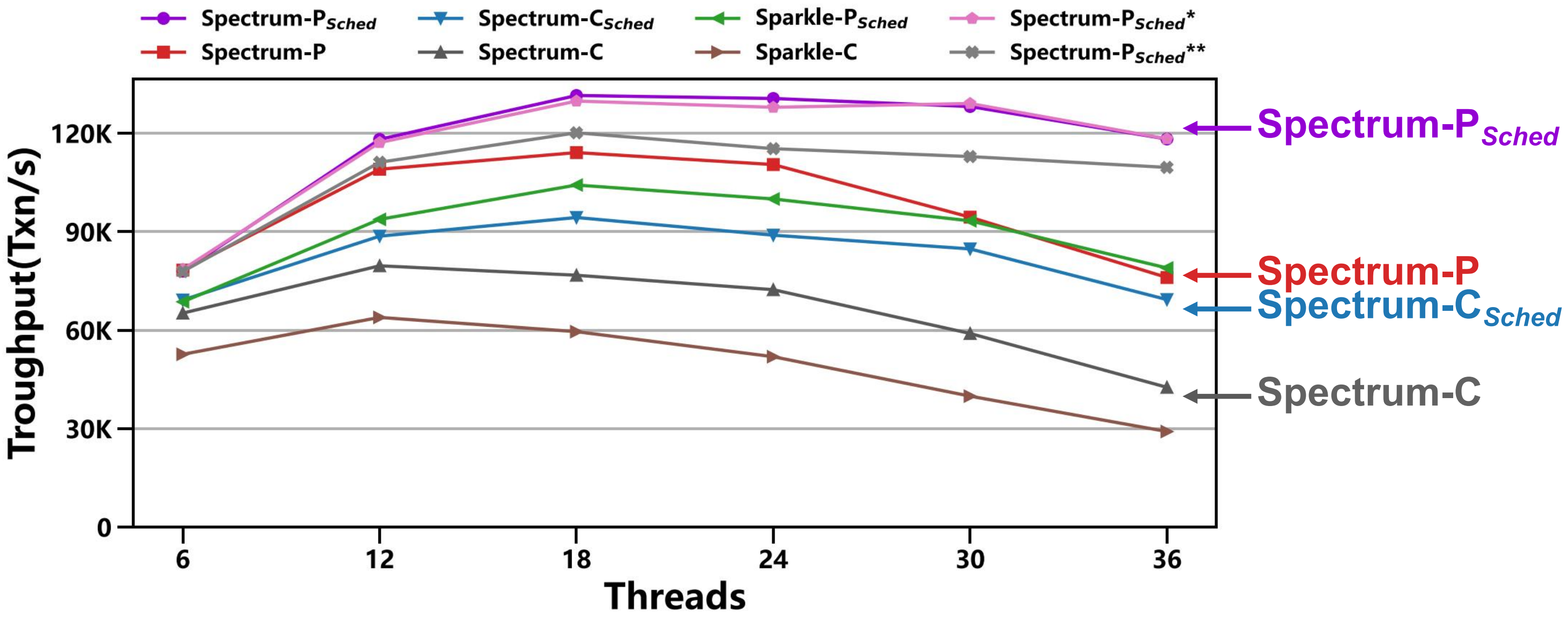

Throughput of varying Zipf (YCSB)




Aborts/Commit in Varying Zipf (YCSB)

# Evaluation

- Ablation Studies on Integrated Optimizations
  - The improvement of using only pre-scheduling (Spectrum-$C_{Sched}$, 1.6x) is less than that of using only partial rollback (Spectrum-P, 1.8x), but leveraging both yields the best performance (Spectrum-$P_{Sched}$, 2.8x than Spectrum-C).




Throughput of Varying Threads (YCSB, Zipf = 1.1)

# Conclusion & Future work

- Spectrum achieves both fair smart contract execution (by ensuring strict determinism) and high performance for blockchain ledgers
  - Speculative execution produces the same agreed-upon serial order with superior parallelism
  - Two novel optimizations: operation-level rollback & predictive scheduling
  - 1.4x ~ 4.1x higher throughput than SOTA DCC schemes (on YCSB,SmallBank,TPC-C alike smart contracts)
- Future work
  - Intra-transaction parallelism
  - Data-partitioned sharding settings

https://github.com/jacklightChen/spectrum

Paper Details

Source Code

Thank you for listening!

Q&A chenzh@stu.ecnu.edu.cn