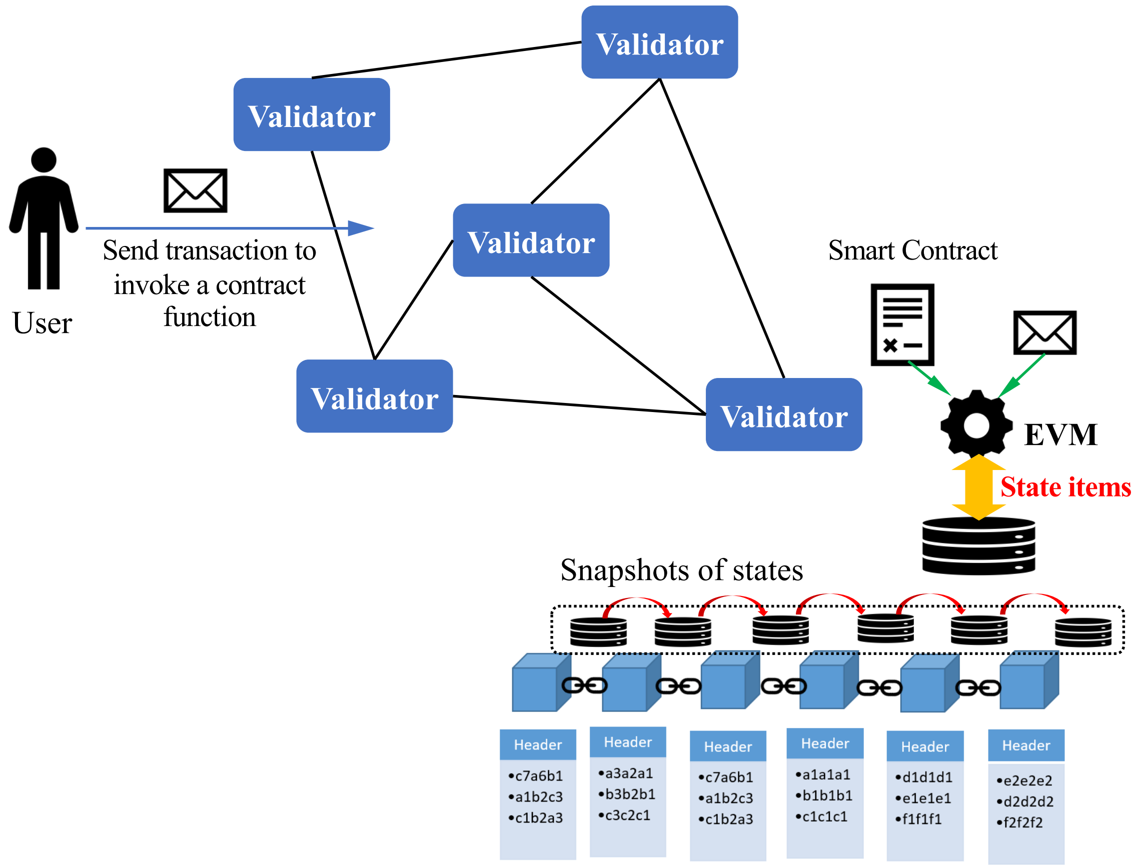
Smart Contract Parallel Execution with Fine-Grained State Accesses

Xiaodong Qi, Jiao Jiao, Yi Li
Nanyang Technological University
xiaodong.qi@ntu.edu.sg

Introduction

- Blockchain
  - Ledger maintained by every validators
- Smart contract
  - Self-enforcing computer program
  - States: persistent storage, e.g., variables
- Consensus protocols
  - Proof-of-Work

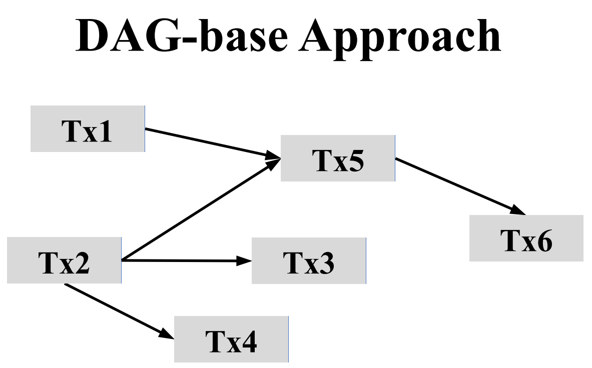


Motivation

- **Serial execution**
  - Ensure state consistency across all validators
  - No parallelism between transaction executions
  - Bottleneck shifting

Motivation

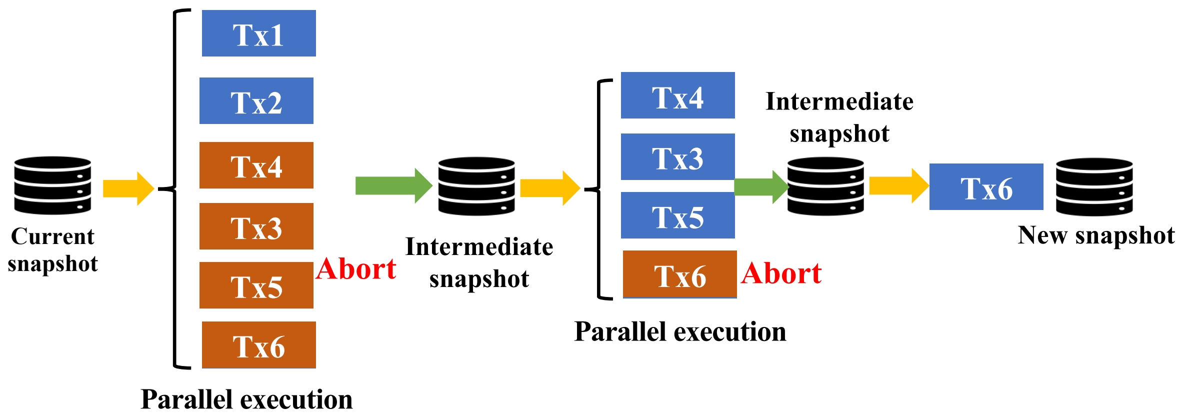
- **Serial execution**
  - Ensure state consistency across all validators
  - No parallelism between transaction executions
  - Bottleneck shifting
- **Parallel solutions**
  - Directed acyclic graph (DAG), Optimistic Concurrency Control (OCC)



Motivation

- **Serial execution**
  - Ensure state consistency across all validators
  - No parallelism between transaction executions
  - Bottleneck shifting
- **Parallel solutions**
  - Directed acyclic graph (DAG), Optimistic Concurrency Control (OCC)

OCC-base Approach



Motivation

- **Serial execution**
  - Ensure state consistency across all validators
  - No parallelism between transaction executions
  - Bottleneck shifting
- **Parallel solutions**
  - Directed acyclic graph (DAG), Optimistic Concurrency Control (OCC)
  - Unrealistic assumption of read/write set
  - Low parallelism caused by coarse-grain analysis

State of Contract

```
pragma solidity ^0.6.12;
contract Example {
 mapping(address => uint) public A;
 uint[] public B;

  function UpdateB(address x, uint y) public
   ↪ {
    uint idx = A[x];
    if(idx > 1) {
      for(uint i = idx; i > 1; i--) {
        B[i] = B[i-2] + y;
      }
    } else {
      B[0] = 0;
      assert(y <= 10);
      B[1] = B[1] + y;
    }
  }
}
```

Fig. 1: Example contract highlighting state access dependencies.

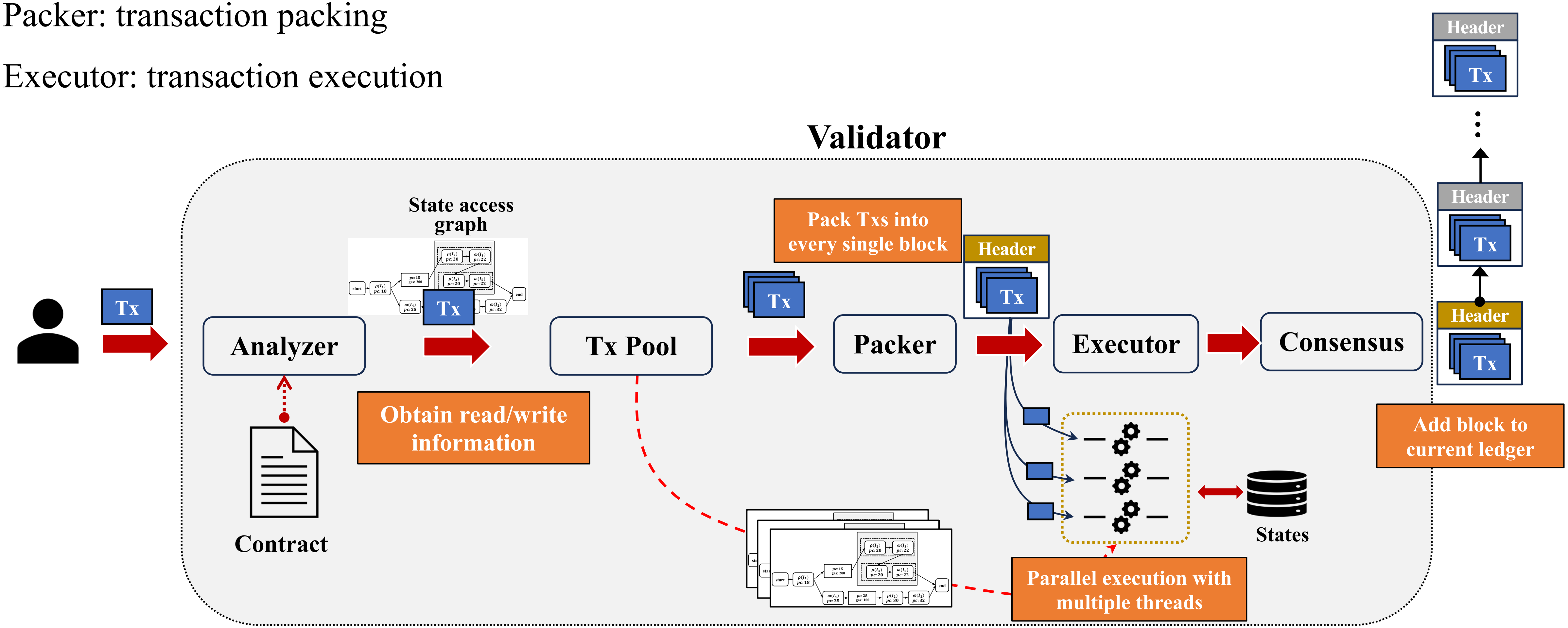
Naïve solution: access entire array B exclusively
Low parallelism!

Contribution

- **Deterministic multi-version concurrency control (DMVCC):**
  - Analyze smart contract code to determine the precise read/write sets of each program statement and enable more find-grained state accesses
  - Eliminate the write-write conflicts between transactions by preserving effects of all write operations as separate versions, which is referred to as *write versioning*
  - Allow transactions to read uncommitted writes through *early-write visibility* feature.

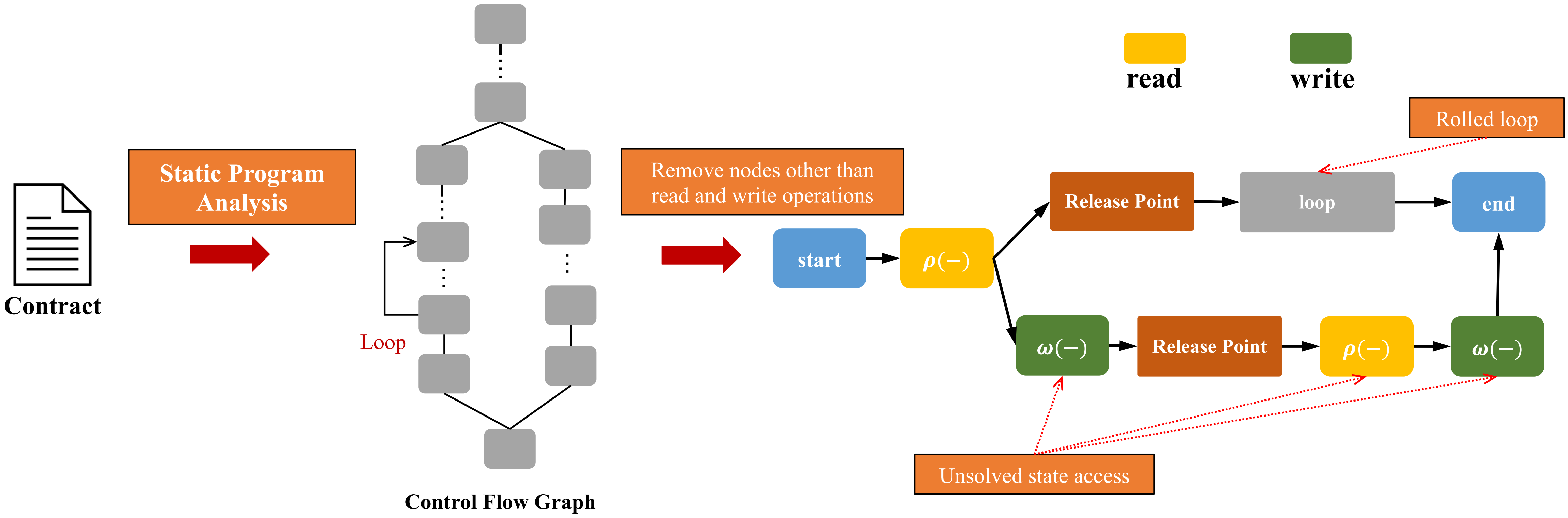
Workflow

- SAG analyzer: *state access graph* anaylsis
- Packer: transaction packing
- Executor: transaction execution



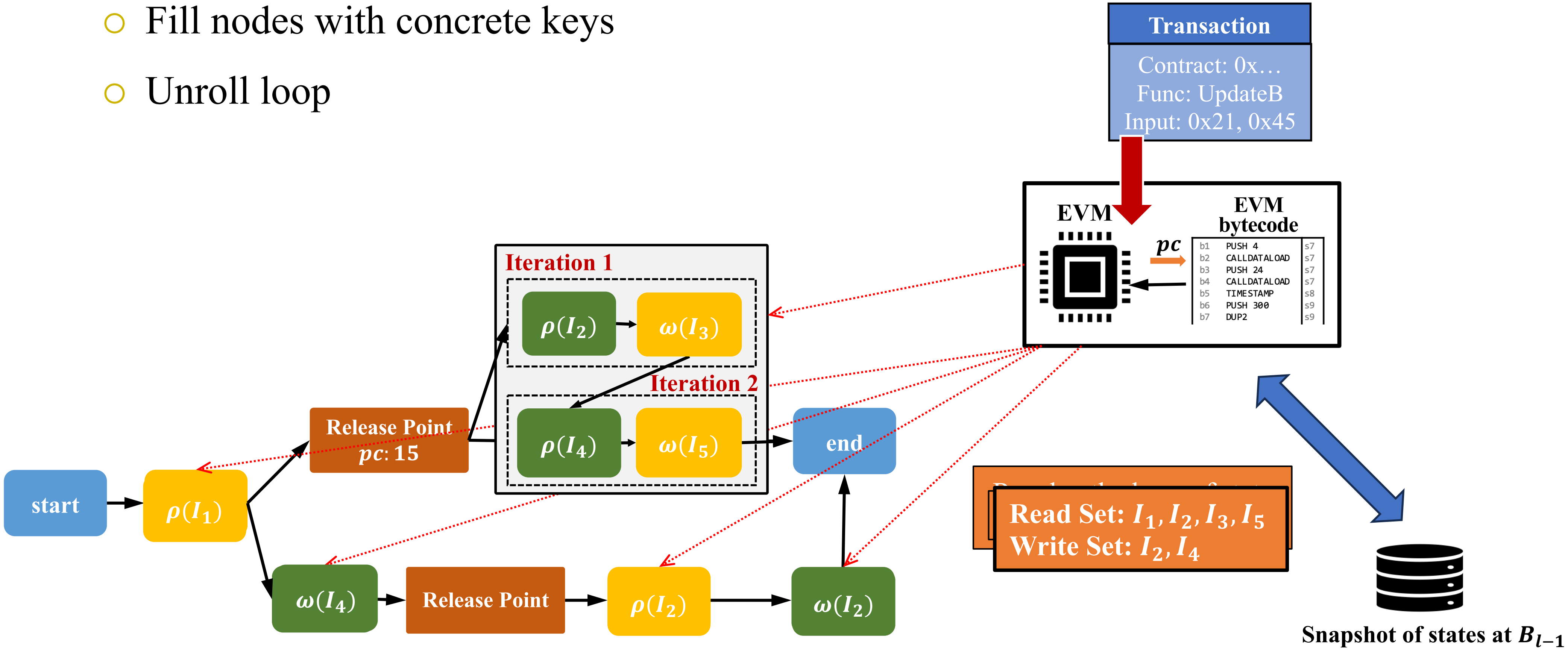
State Access Graph

- **Partial state access graph (P-SAG)**
  - A simplified control flow graph
  - Nodes: read /write, loop, release point



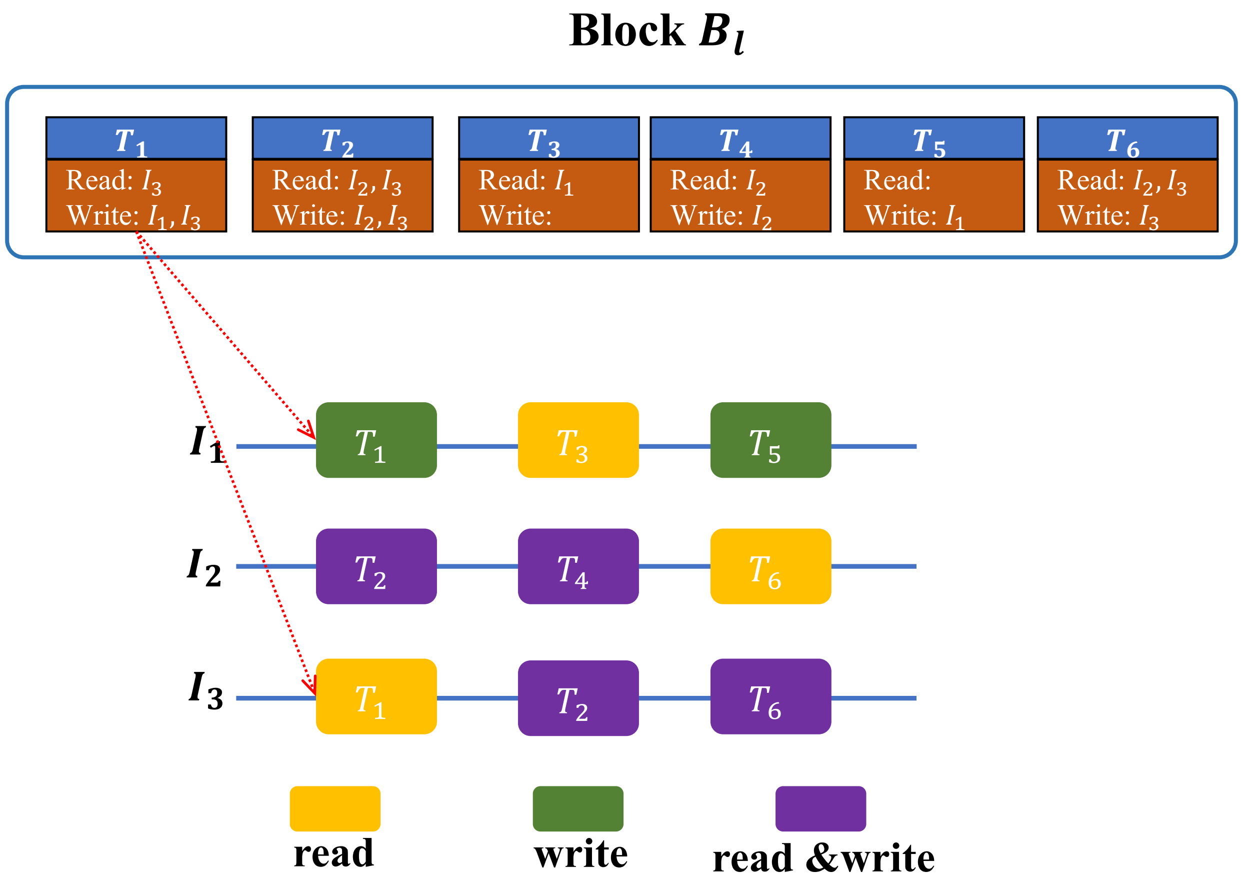
State Access Graph

- **Complete state access graph (C-SAG)**
  - Fill nodes with concrete keys
  - Unroll loop



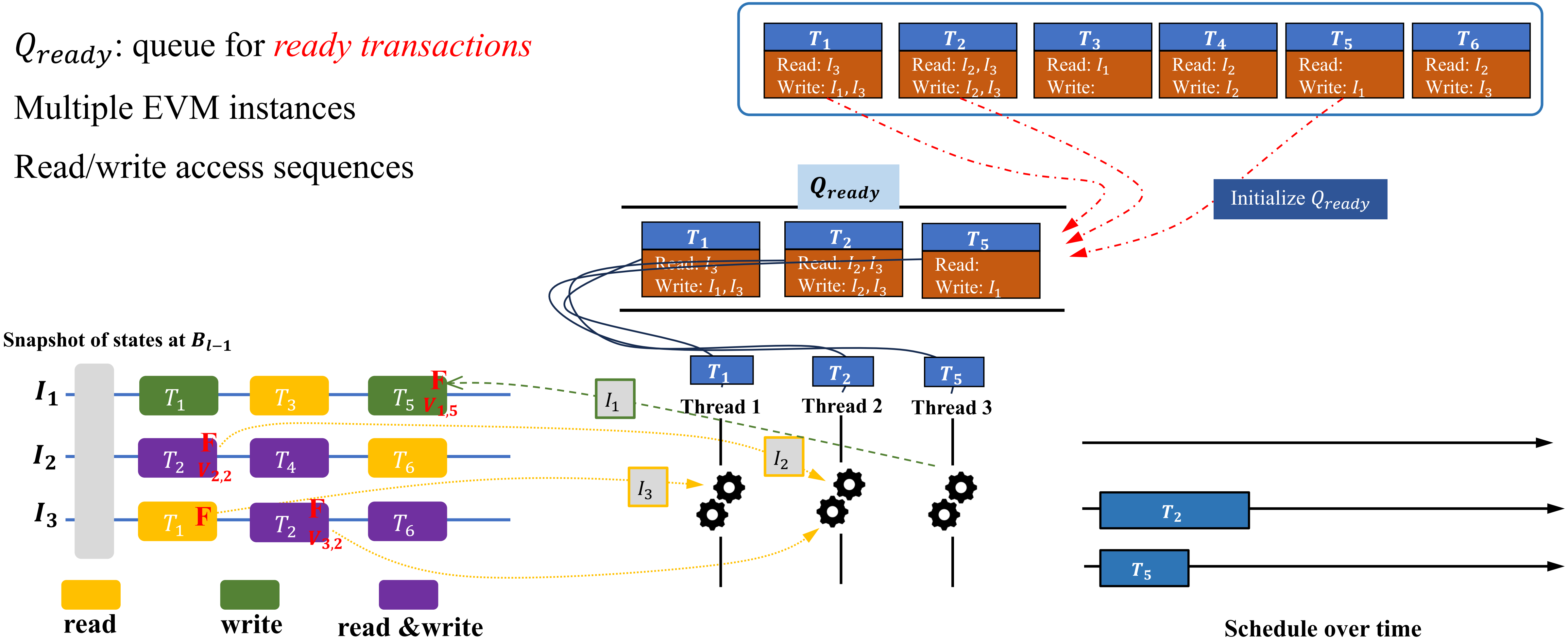
Access sequences

- **Access sequence construction**
  - Record all possible conflicts between transactions



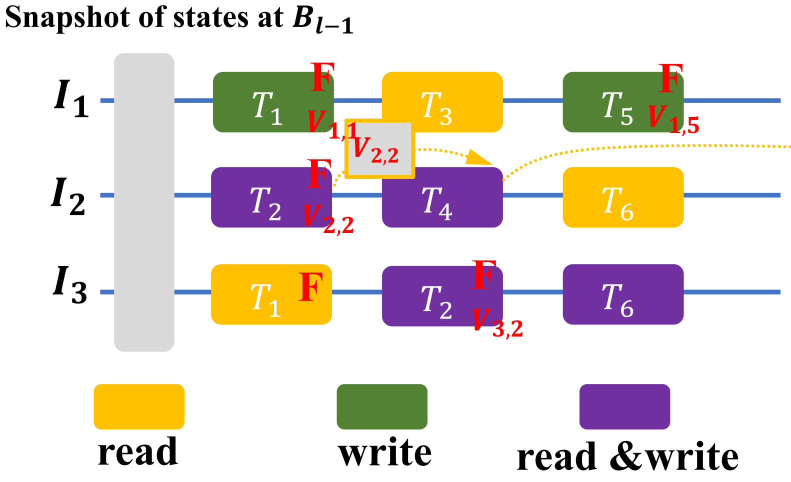
Schedule Generation

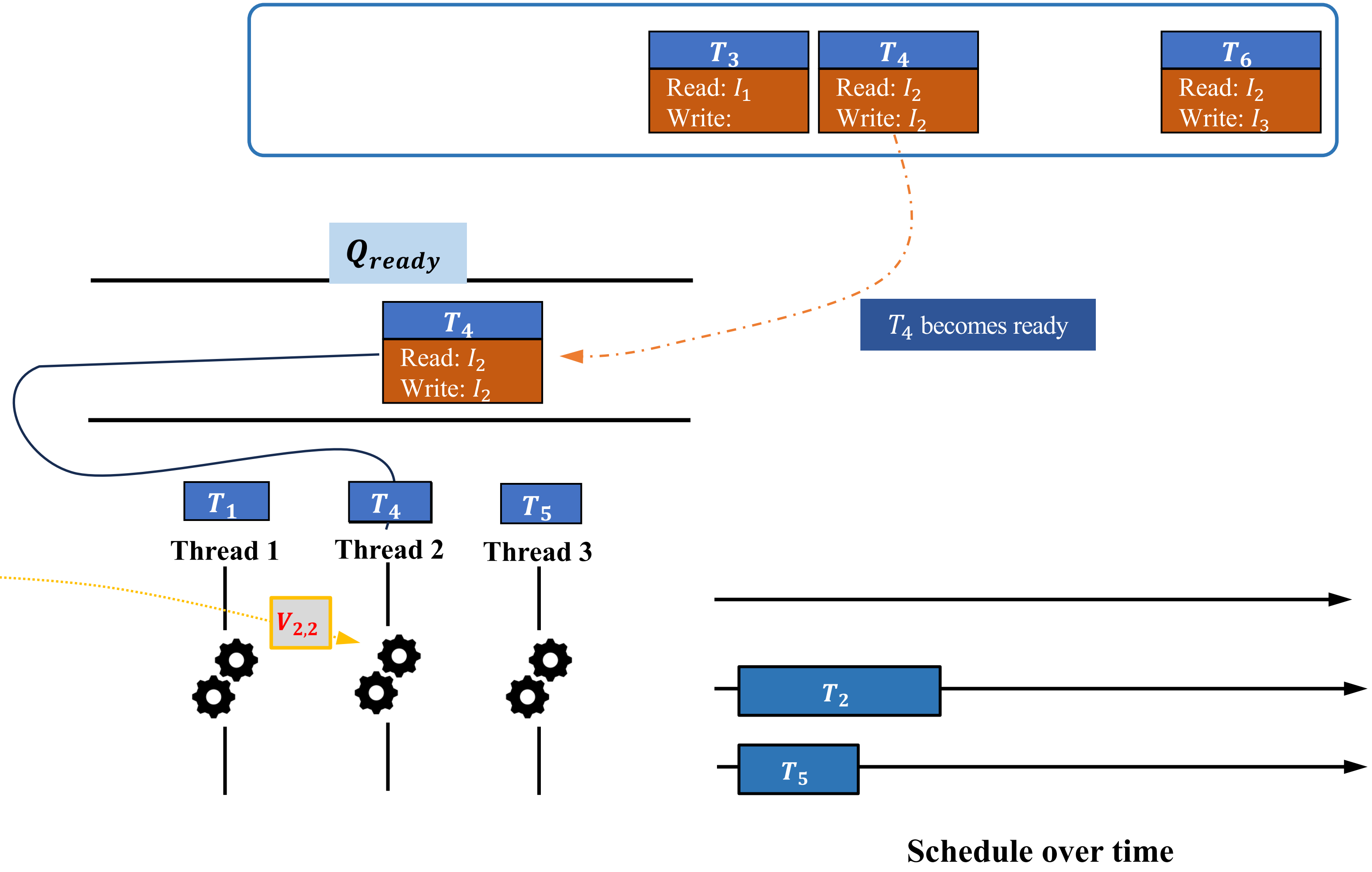
- Q_{ready}: queue for *ready transactions*
- Multiple EVM instances
- Read/write access sequences



Schedule Generation

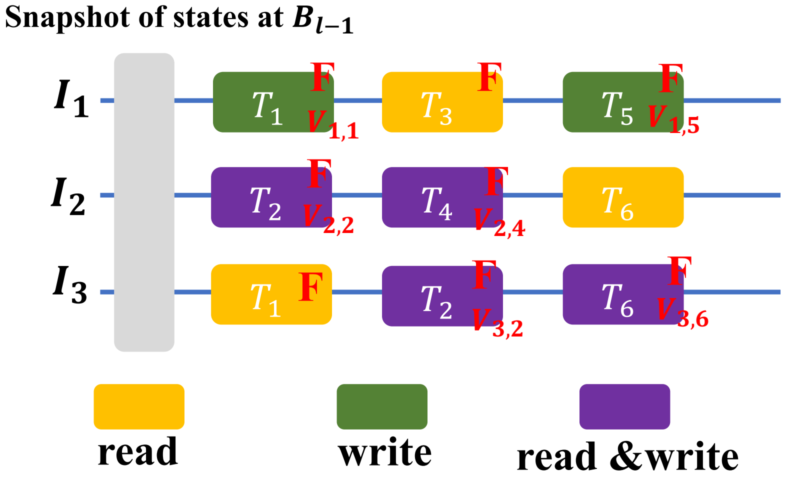
- Q_{ready}: queue for *ready transactions*
- Multiple EVM instances
- Read/write access sequences

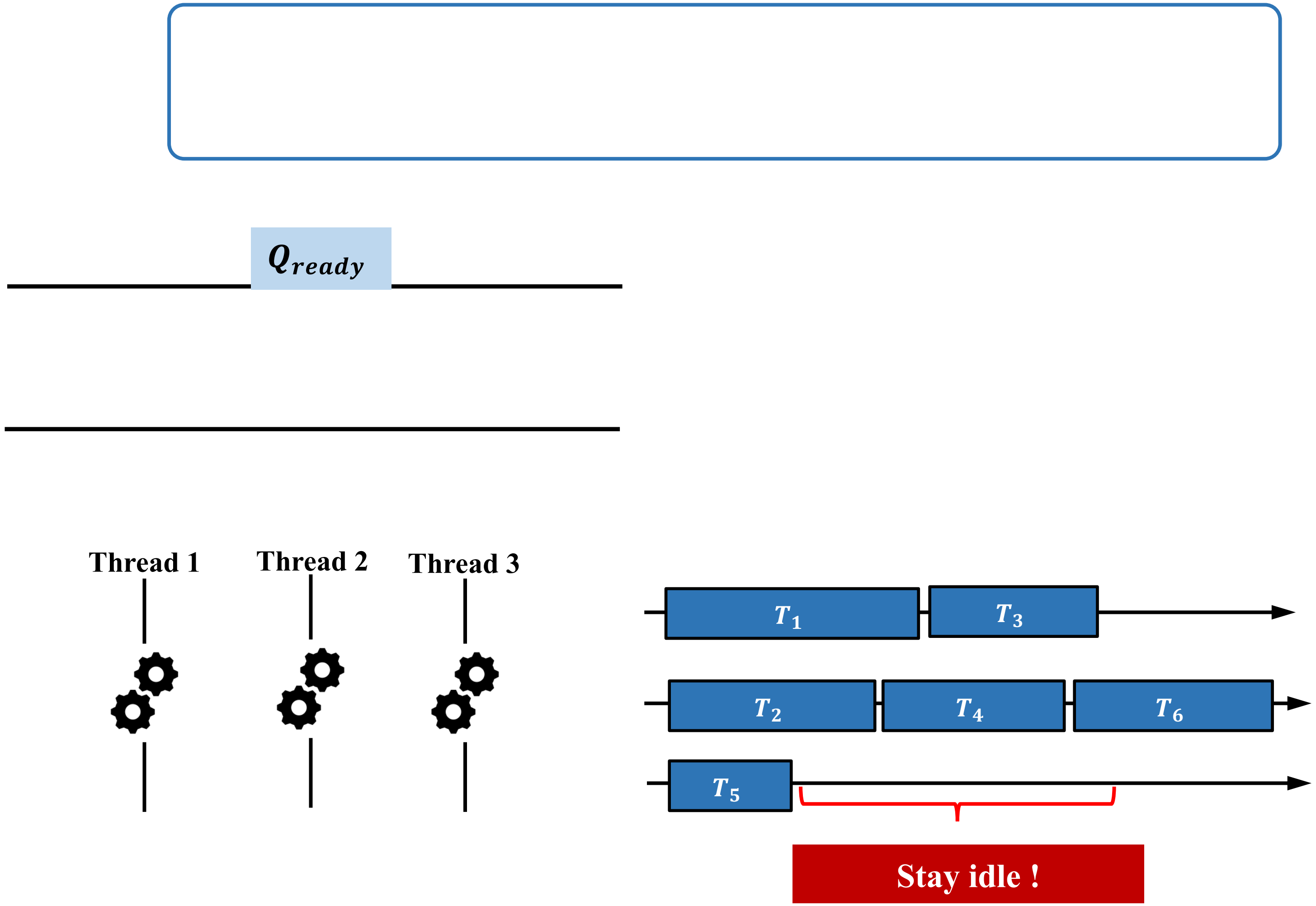




Schedule Generation

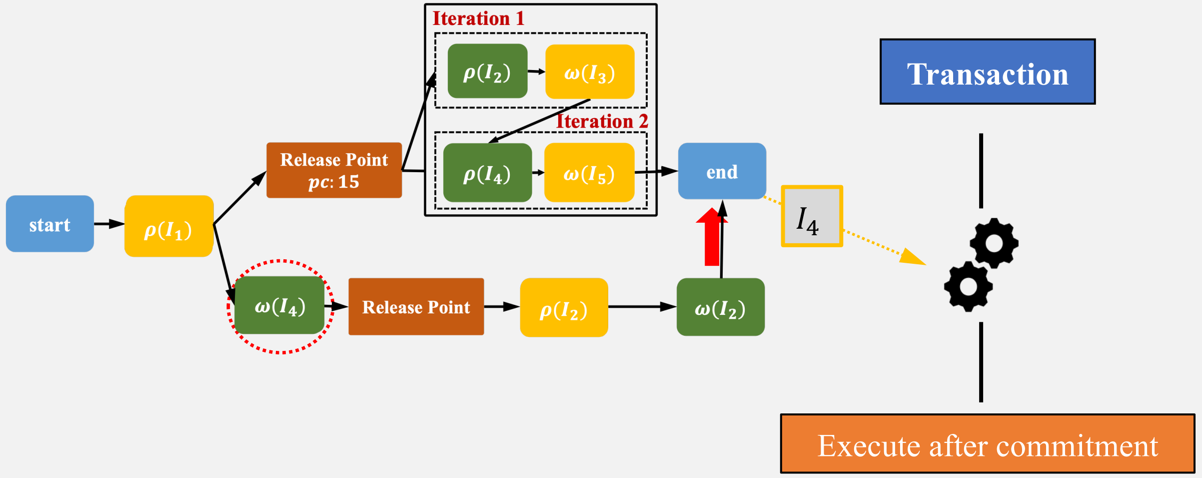
- Q_{ready}: queue for *ready transactions*
- Multiple EVM instances
- Read/write access sequences



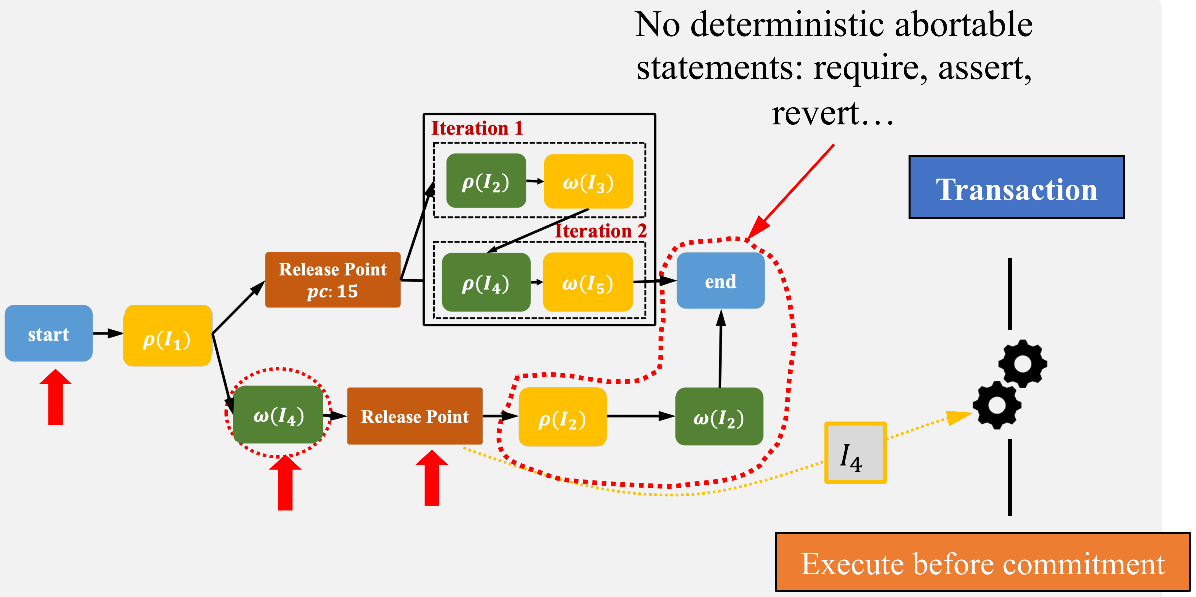


Early Write Visibility

- Transaction- vs. statement-level synchronization



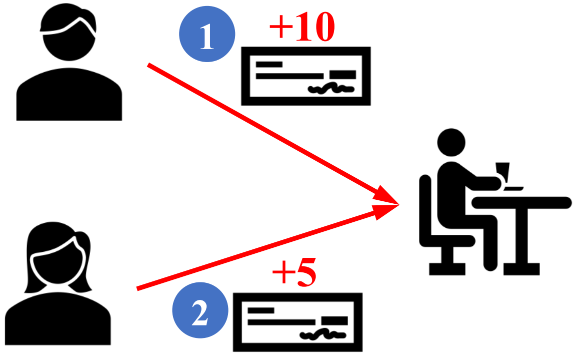
Transaction-level synchronization

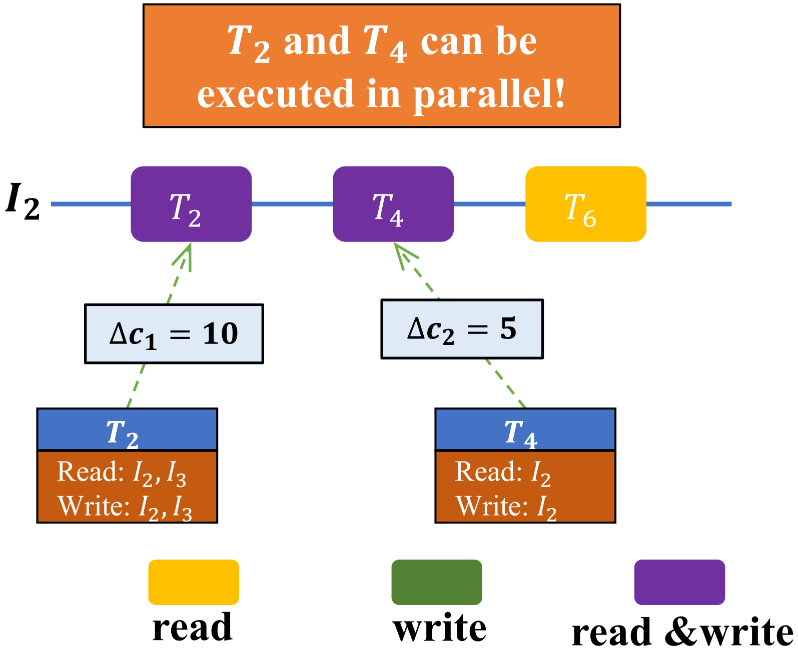


Statement-level synchronization

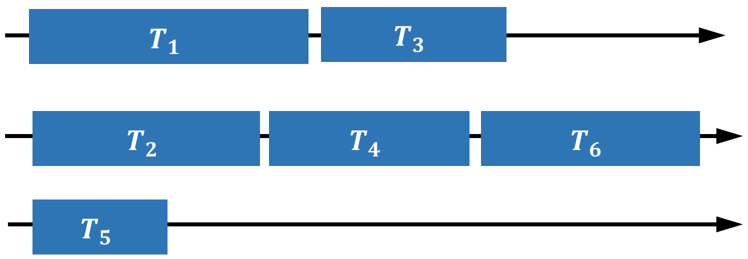
Commutative write

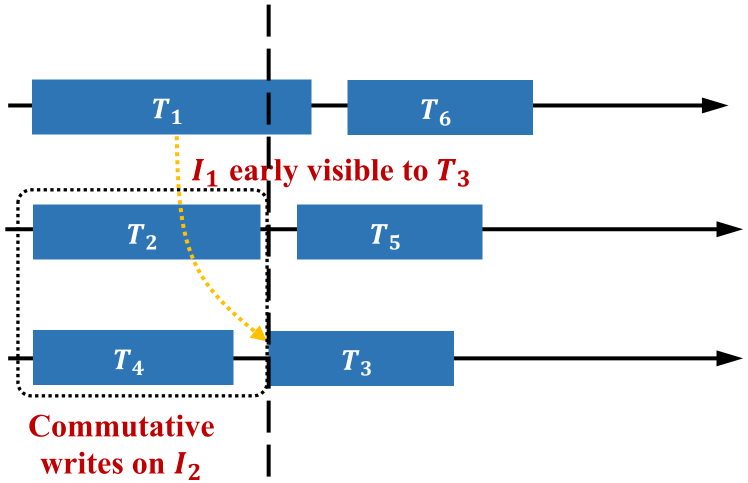
- Increment same state item without reading original value
- Perform commutative writes in parallel
- Merge increments to recover a complete value





Optimized Schedule Generation





Evaluation

- **Comparisons**
  - DAG-based approach
  - OCC-based approach
- **Workload**
  - Transactions from Ethereum Mainnet
  - Jan 1, 2022 -- April 30, 2022 (769,020 blocks in total)
  - 122 million transactions and 84 million transactions (69%) made contract calls
- **Testbed**
  - Ubuntu 18.04.3 LTS desktop equipped with an Intel Core i7 16-core and 32GB memory
  - Up to 32 threads per validator

Evaluation

- **Research questions**
  - **RQ1:** How well do the parallel execution results of DMVCC meet the deterministic serializability criteria?
  - **RQ2:** How much speedup can DMVCC achieve over the serial execution and how does it compare with other existing approaches?
  - **RQ3:** How efficient is DMVCC in a real-world blockchain environment?

Experiment

- **RQ1**: how well do the parallel execution results of DMVCC meet the deterministic serializability criteria?
  - Compare the execution results of DMVCC and serial execution
  - Matched results for 121,210 blocks

Experiment

- **RQ2:** How much speedup can DMVCC achieve over the serial execution and how does it compare with other existing approaches?
  - Performance of EVM execution without taking the impact of consensus into account
  - 1000 transactions per block
  - 21.35x (DMVCC), 11.04x (DAG), 13.86x (OCC)
  - High-contention setting: 1% hot contracts, 50% hot contract access
  - 13.73x (DMVCC), 3.05x (DAG), 3.48x (OCC)

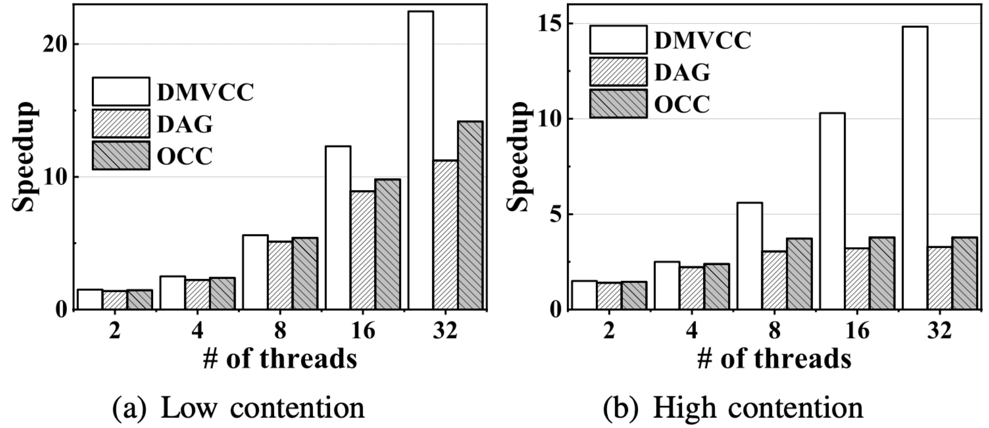
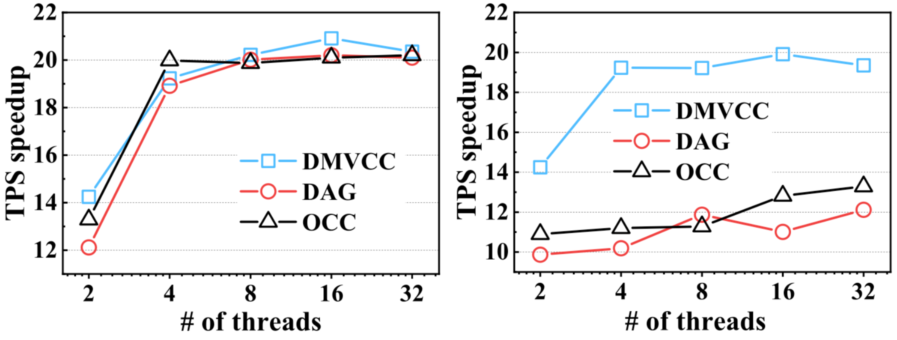


Fig. 7: Speedup of all parallel execution approaches. The x-axis shows the number of threads, and y-axis shows the speedup achieved.

Experiment

- **RQ3:** How efficient is DMVCC in a real-world blockchain environment?
  - A micro Ethereum testnet with 20 validators
  - Mining cycle 12s
  - Low-contention setting: 19.79x, execution is not the bottleneck
  - High-contention setting: 18.35x, DAG and OCC process 60% transactions of DMVCC



(a) Low contention (b) High contention

Fig. 8: Throughput speedup for blockchain of all parallel execution approaches.

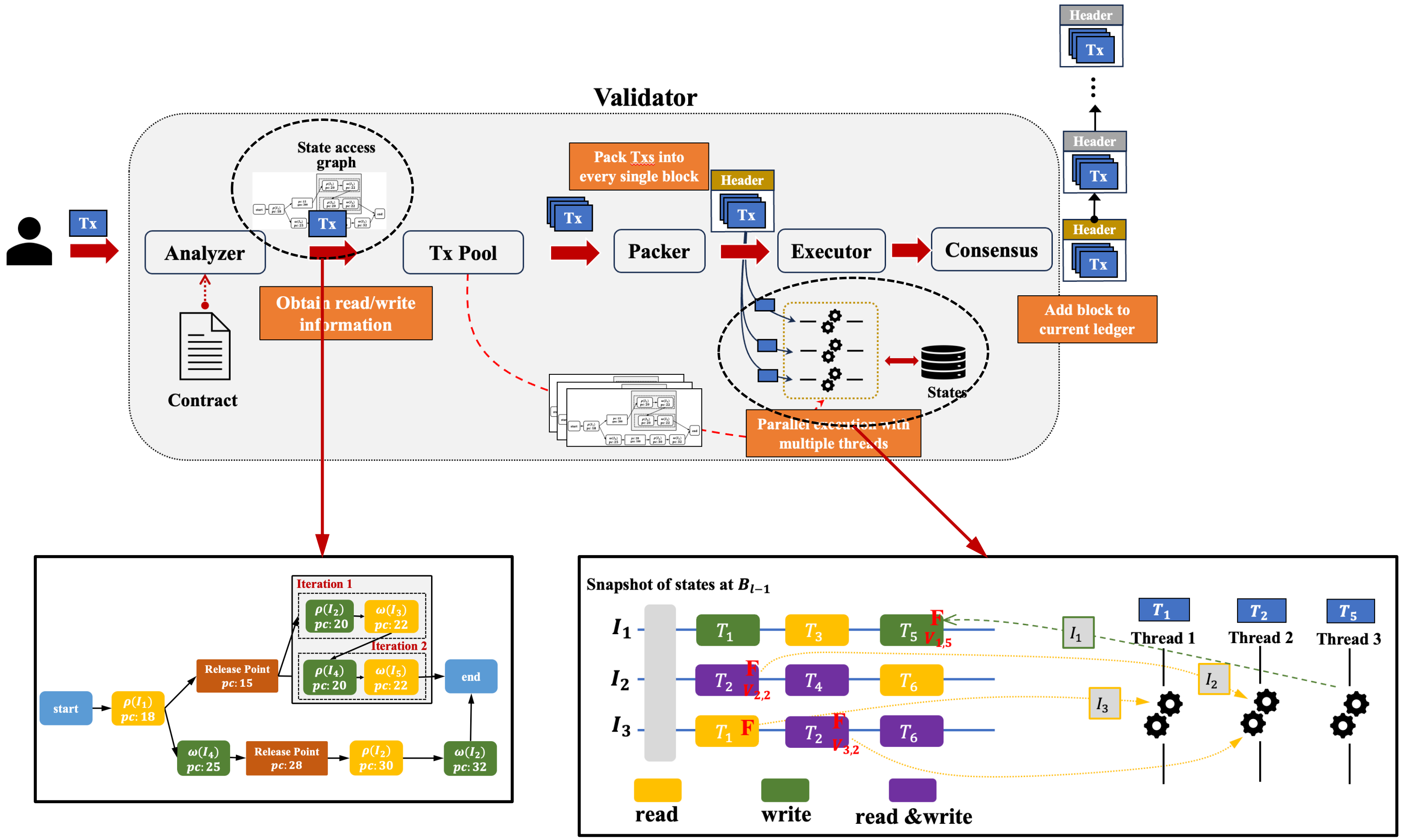
Conclusion

- Introduce a novel scheduling framework, DMVCC, which improves parallelism for high-contention transactions with more fine-grained state accesses.

- Support write versioning, which helps avoid write-write conflicts, and early write visibility, which makes writes visible to other transactions.

Thank You

Q&A



Validator
Tx
Analyzer
State access graph
Obtain read/write information
Contract
Tx Pool
Pack Txs into every single block
Packer
Header
Executor
Consensus
States
Parallel execution with multiple threads
Add block to current ledger
Iteration 1
Iteration 2
Release Point
start
end
Snapshot of states at B_{l-1}
Thread 1
Thread 2
Thread 3
read
write
read &write