SlimChain: Scaling Blockchain Transactions through Off-Chain Storage and Parallel Processing

Cheng Xu[1,2], Ce Zhang[2], Jianliang Xu[2], and Jian Pei[1]

[1]Simon Fraser University, Canada [2]Hong Kong Baptist University, Hong Kong

{chengxu, cezhang, xujl}@comp.hkbu.edu.hk jpei@cs.sfu.ca

Motivation and System Model

- **Issues of Current Blockchain System**:
  - Every node keeps a full replication of transaction history and ledger states.
  - Every node needs to validate each transaction in block.
  - High storage (ETH full node: 870GB) and execution overhead.
- **Stateless design**:
  - Move ledger states and transaction executions off-chain to a subset of nodes.
  - Reduce the on-chain load.
- **Challenges**:
  - Transaction contains arbitrary logic
    ⇒ Novel proof techniques to ensure integrity of transaction execution
  - Transaction introduces arbitrary sized read/write set
    ⇒ Extra design to support on-chain commitment updates
  - Transaction should be processed in parallel
    ⇒ New method for validating and committing concurrent transactions
- **Transaction Processing Workflow**:
  ❶ Send TX ❷ Verifiable TX execution ❸ Broadcast
  ❹ Validate & append to ledger ❺ Synchronize

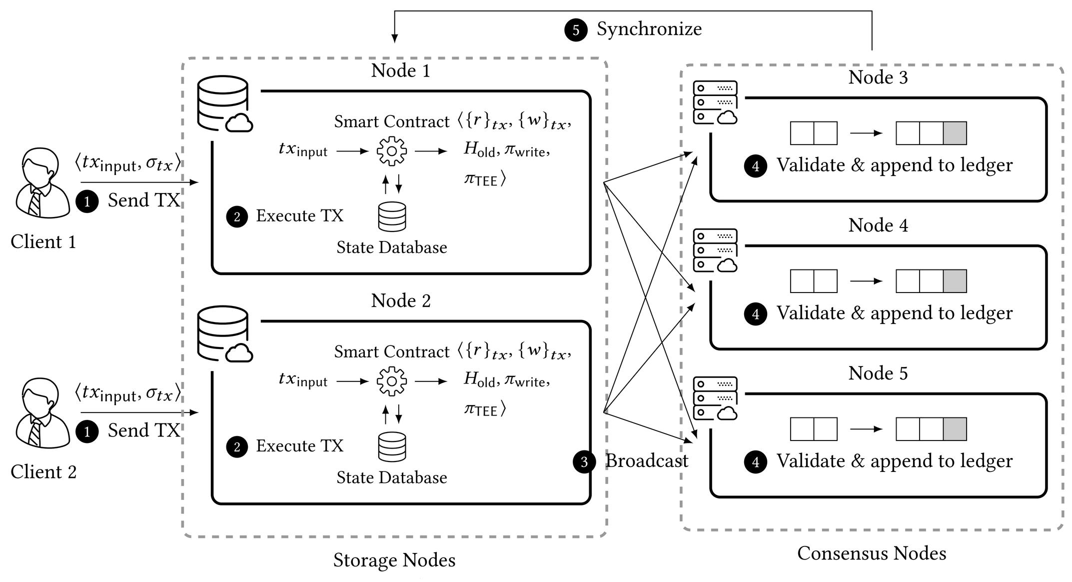

Fig. 1: System Model

Off-chain Transaction Execution

- **Inside TEE**:
  - Generate the read/write set $\{r\}_{tx}, \{w\}_{tx}$ w.r.t. the current state H_{old}.
  - Get the read set Merkle proof π_{read} and verify it w.r.t. $\{r\}_{tx}$.
  - Compute the TEE proof π_{TEE} w.r.t. $\{r\}_{tx}, \{w\}_{tx}, H_{\text{old}}$.
- **Outside TEE**:
  - Get the write set Merkle proof π_{write}.
- **Broadcast** $tx_{\text{submit}} = \langle tx_{\text{input}}, \{r\}_{tx}, \{w\}_{tx}, H_{\text{old}}, \pi_{\text{TEE}}, \pi_{\text{write}} \rangle$:
  - π_{TEE} ensures the execution integrity and the read integrity.
  - $\{r\}_{tx}, \{w\}_{tx}, H_{\text{old}}, \pi_{\text{write}}$ provide enough information for on-chain validation and commitment.

On-chain Transaction Commitment

- **Challenges**:
  - How to update the state commitment without access to the full tree?
  - How to check conflict among transactions and ensure serializability?
- **Our Solution**: Keep track of temp state of recent k blocks.
  - $\mathcal{T}_w$: a partial Merkle tree w.r.t. the write set in the past k blocks.
  - $M_{i \mapsto r}, M_{i \mapsto w}$: map between block height to read, write addresses.
  - $M_{r \mapsto i}, M_{w \mapsto i}$: map between read, write addresses to an ordered list of block heights.
- **Procedure**:
  - Discard TX is older than recent k blocks.
  - Validate $\pi_{\text{TEE}}, \pi_{\text{write}}$.
  - Check conflict of $\{r\}_{tx}, \{w\}_{tx}$.
    - OCC: Check whether other committed transactions have modified the data that the current transaction accessed.
    - SSI: Check *write-write conflict* and whether there are *rw-dependencies* both pointing to and originating from the current transaction.
  - Update ledger state commitment over $\mathcal{T}_w$ and generate new block.
    - Update $\mathcal{T}_w$: take the Merkle proof π_{write} and write set $\{w\}_{tx}$ to apply the writes from the transaction.
    - Tidy $\mathcal{T}_w$: remove the write addresses whose age is more than k blocks.

Example of Transaction Commitment

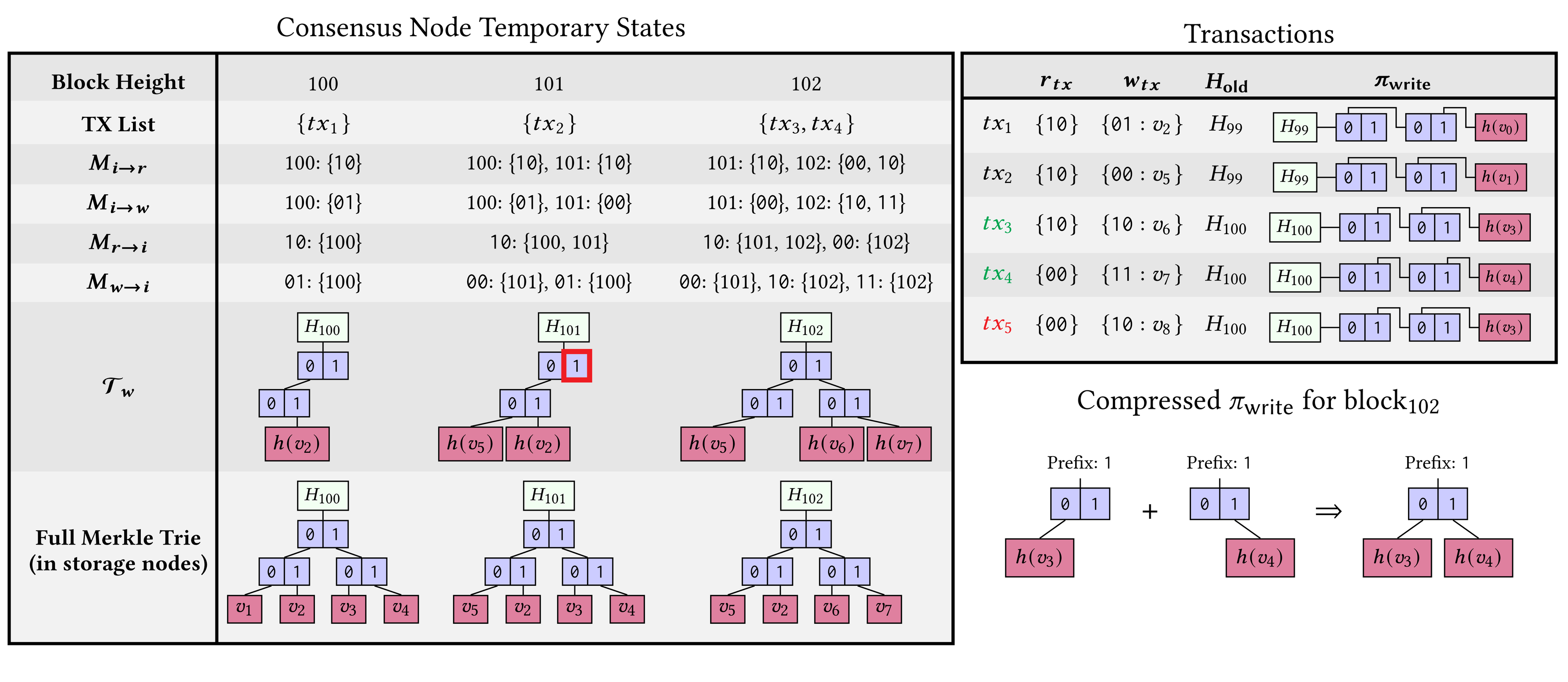

Fig. 2: Example of Transaction Commitment

Node Synchronization

- **Block Observer**
  - Validate and log blocks created by the block proposers.
  - Compress π_{write} to reduce network transmission.
- **Storage Node**
  - Execute the similar procedure as on-chain transaction commitment.
  - Keep transaction data and state data.
  - Maintain full Merkle tree instead of partial tree $\mathcal{T}_w$.

Implementation

- Implement in Rust program language (LOC: 26,000).
- Two consensus protocols are implemented: PoW, Raft.
- Source code is available at `https://git.io/slimchain`.

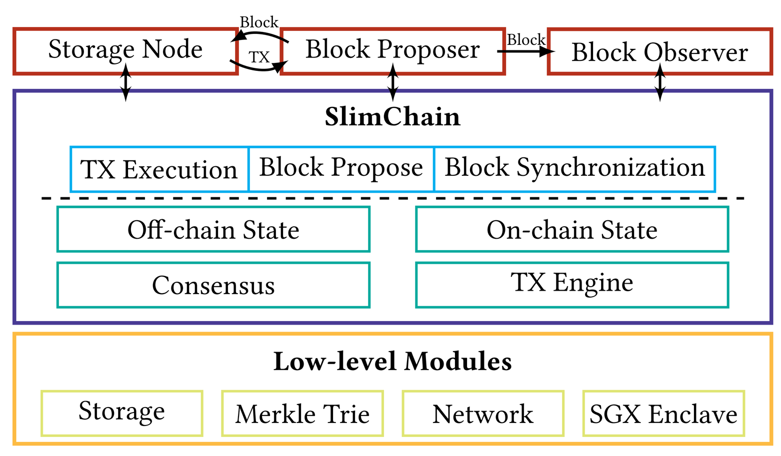

Fig. 3: System Architecture of SlimChain

Performance Evaluation

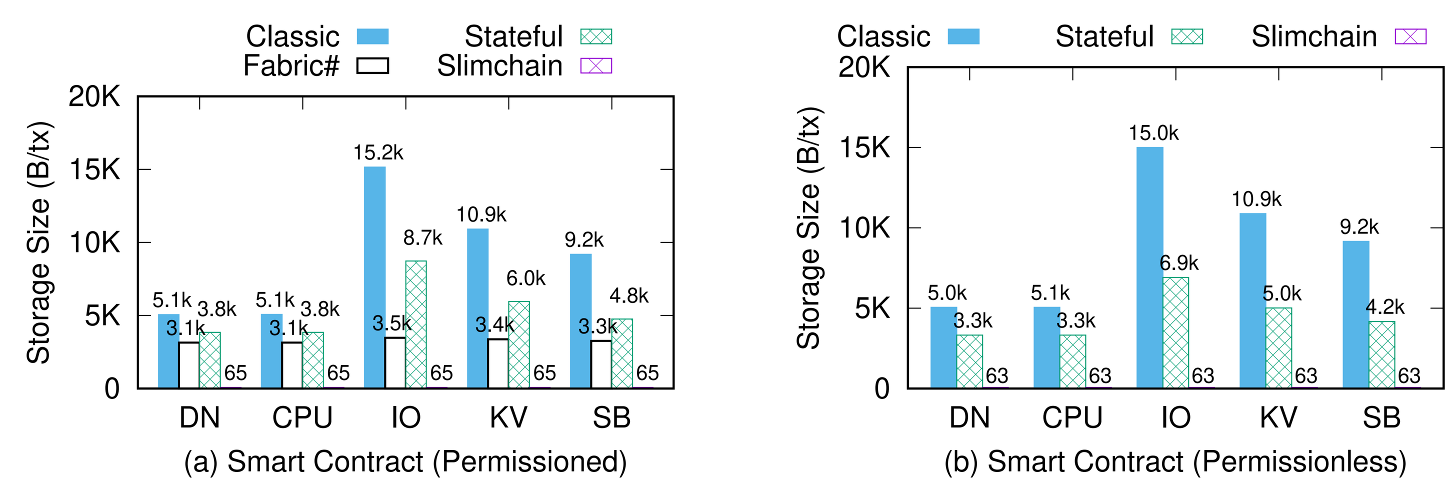

Fig. 4: Consensus Node Storage Size (B/tx) vs. Smart Contract

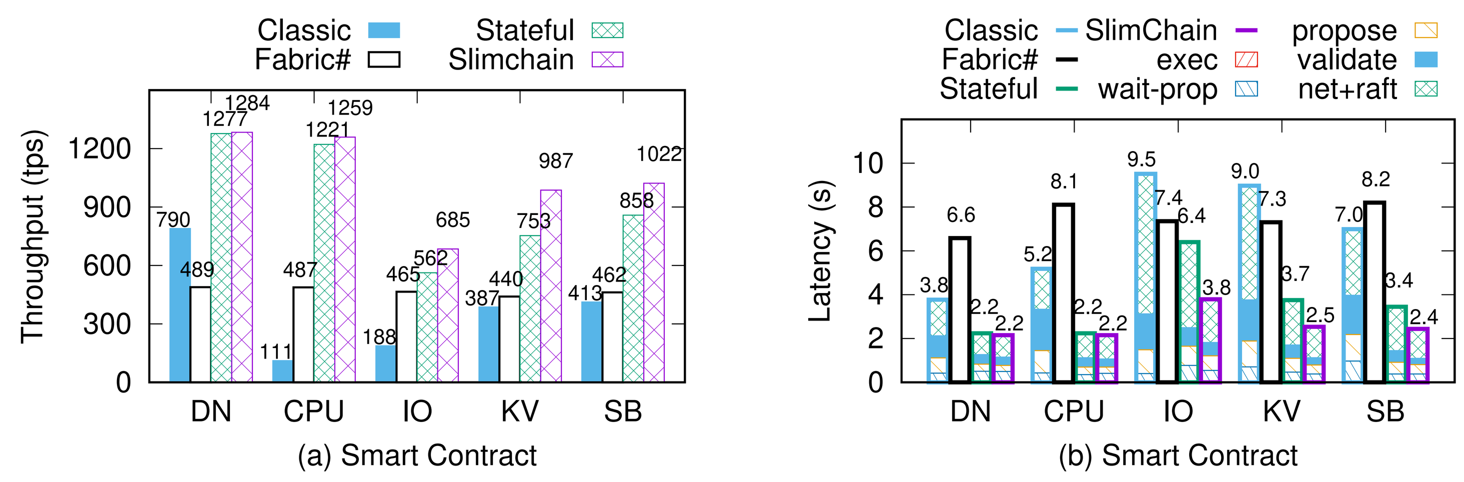

Fig. 5: Throughput/Latency vs. Smart Contract (Permissioned)

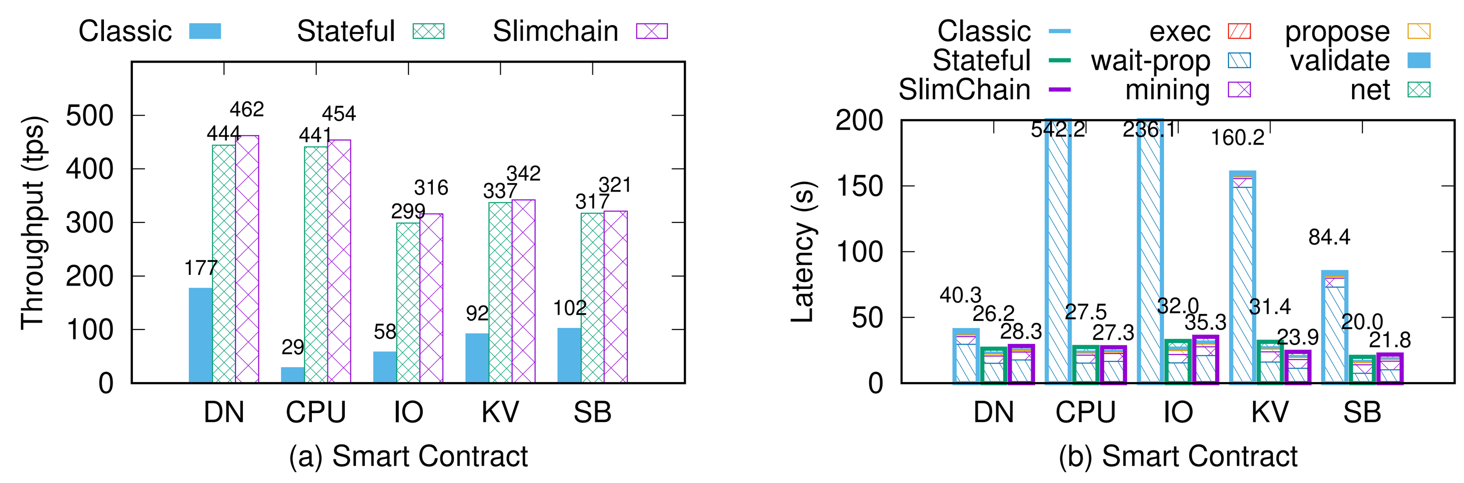

Fig. 6: Throughput/Latency vs. Smart Contract (Permissionless)