SlimChain: Scaling Blockchain Transactions through Off-Chain Storage and Parallel Processing (Technical Report)

Cheng Xu[1,2,†], Ce Zhang[2,†], Jianliang Xu[2], and Jian Pei[1]
[1]Simon Fraser University [2]Hong Kong Baptist University
{chengxu, cezhang, xujl}@comp.hkbu.edu.hk jpei@cs.sfu.ca

ABSTRACT

Blockchain technology has emerged as the cornerstone of many decentralized applications operating among otherwise untrusted peers. However, it is well known that existing blockchain systems do not scale well. Transactions are often executed and committed sequentially in order to maintain the same view of the total order. Furthermore, it is necessary to duplicate both transaction data and their executions in every node in the blockchain network for integrity assurance. Such storage and computation requirements put significant burdens on the blockchain system, not only limiting system scalability but also undermining system security and robustness by making the network more centralized. To tackle these problems, in this paper, we propose SlimChain, a novel blockchain system that scales transactions through off-chain storage and parallel processing. Advocating a stateless design, SlimChain maintains only the short commitments of ledger states on-chain while dedicating transaction executions and data storage to off-chain nodes. To realize SlimChain, we propose new schemes for off-chain smart contract execution, on-chain transaction validation, and state commitment. We also propose optimizations to reduce network transmissions and a new sharding technique to improve system scalability further. Extensive experiments are conducted to validate the performance of the proposed SlimChain system. Compared with the existing systems, SlimChain reduces the on-chain storage requirements by 97% ~ 99%, while also improving the peak throughput by 1.4X ~ 15.6X.

1 INTRODUCTION

Blockchain is an emerging technology that is considered to have the potential to revolutionize database systems [1, 2]. It has been the cornerstone of many decentralized applications in a wide range of domains, such as finance [3], healthcare [4], and supply chain [5]. Under the hood, blockchain is a secure append-only data structure built upon the incoming transactions that are agreed by a set of untrusted nodes in a P2P network. It utilizes cryptographic signatures, hash chains, and consensus protocols to create a trusted distributed system upon a foundation of otherwise untrusted peers. While first-generation blockchain systems such as Bitcoin [3] are built specifically to support cryptocurrencies, second-generation blockchains such as Ethereum [6] extend their functionality to support general-purpose transactions in what is known as smart contracts. Smart contracts are user-defined, trusted programs that allow users to process data in the blockchain. They can be deployed in the blockchain and triggered for execution by future transactions.

In order to maintain the same total order of transactions, ensure execution integrity, and support data provenance, existing blockchain systems often require every node in the network to keep a full replication of the transaction history and the ledger states. These ever-growing data structures, however, have become too large after a while. For example, as of July 2021, the entire blockchain ledger is around 350GB for Bitcoin[1] and has exceeded 7TB for Ethereum.[2] To mitigate this problem, most blockchain nodes usually maintain only a compact index called validation states, which, being substantially smaller than the entire ledger, is sufficient for determining transactions' validity. However, the validation states are still in the order of GBs (e.g., Ethereum's validation states are around 870GB[3]). Additionally, blockchain nodes are required to replay all transactions locally based on the replicated states. Such cumbersome stateful data poses significant storage and computation costs, limiting system scalability. Moreover, it undermines system security and robustness by making the network more centralized as fewer and fewer nodes are capable of handling such a large amount of data.

In attempting to solve these problems, it became clear that it is a huge waste of storage and computation resources to require every blockchain node to keep the same replica of the data and repeat the same transaction executions. One solution is sharding [7, 8]. Sharding horizontally partitions the blockchain into multiple parallel chains, each of which is managed by only a subset of the nodes. This is an effective means of reducing storage and computation duplications among different shards. However, it is also clear that this only alleviates the problem by a constant factor (i.e., the number of shards). Within each shard, it is still necessary for blockchain nodes to duplicate storage and computation. Furthermore, existing sharding solutions often introduce new problems, such as cross-shard transaction processing and attacks by slowly-adaptive Byzantine adversaries [7]. Another solution is the use of light nodes. Unlike full nodes, which store full states, light nodes keep only the block headers. However, although light nodes are able to follow the consensus protocols, they alone are unable to verify the validity of the transactions. Therefore, using light nodes does not address the issue of centralization due to the state maintenance burden.

More recently, the concept of stateless blockchain has been proposed [9, 10]. This is an off-chain scaling approach that moves ledger states and transaction executions off-chain to a subset of the nodes, thereby reducing the on-chain load. However, existing stateless blockchain systems [9, 10, 11] are designed particularly for cryptocurrencies. Attempting to develop a general-purpose stateless blockchain that supports smart contracts presents several new challenges. First, a fundamental issue is that in general applications, transactions supported by smart contracts may contain arbitrary logic. This demands novel proof techniques to attest to the integrity of off-chain executions. Second, a smart contract transaction would

[†]These authors have contributed equally to this work.

[1]https://www.blockchain.com/charts/blocks-size
[2]https://etherscan.io/chartsync/chainarchive
[3]https://etherscan.io/chartsync/chaindefault

introduce read and write sets of arbitrary size. This is significantly more complex than simple cryptocurrency transactions and requires an extra design to support on-chain commitment updates. Third, the cryptocurrency exchange methods proposed in existing studies offer very limited support for parallel transaction executions. To improve system throughput, new transaction processing methods are needed to support validating and committing concurrent transactions arrived from an asynchronous network despite of the stateless design.

To address these challenges, in this paper, we present SlimChain, a stateless blockchain system that scales transactions through off-chain storage and parallel processing. The main idea is to leverage off-chain storage nodes to store ledger states and simulate the execution of smart contracts, allowing the blockchain to maintain only the short commitments of ledger states. We start by designing a verifiable transaction execution algorithm for the storage nodes. It executes transactions off-chain in parallel and computes some auxiliary information to attest to the integrity of the execution as well as to facilitate subsequent transaction commitment by the on-chain consensus nodes. We then develop a novel on-chain temporary state, which provides minimal yet enough information to enable stateless on-chain transaction validation, concurrency control, and commitment. In particular, a novel partial Merkle trie structure is proposed to enable stateless consensus nodes to maintain the root of on-chain states without using a full Merkle trie. Our system can handle parallel transactions submitted from an asynchronous network even if they arrive in any arbitrary order and ensure that the on-chain data can be collectively maintained and perfectly synchronized among all nodes in the blockchain network.

Since the network layer is often a bottleneck in a blockchain system [12, 13], we also propose several optimizations for SlimChain to reduce its network transmissions during node synchronization. To further improve the scalability of SlimChain, we propose a brand new way to support sharding under stateless settings, which addresses many weaknesses of the existing sharding solutions. Our method is independent of the consensus protocols and can be used in both permissionless and permissioned settings.

To summarize, this paper's contributions are as follows:

- We present SlimChain, a novel stateless blockchain system for scalable transaction processing with smart contract capabilities. To the best of our knowledge, this is the first of its kind in the literature.
- We propose new off-chain smart contract execution, stateless on-chain transaction validation, and novel state commitment schemes to realize SlimChain in both permissionless and permissioned settings.
- We further propose optimization techniques for reducing network transmissions during node synchronization and a novel storage sharding technique to improve system scalability further.
- We build an end-to-end prototype and conduct extensive experiments to validate the performance of SlimChain system. Compared with the existing systems, SlimChain reduces the on-chain storage requirements by 97% ~ 99%, while also improving the peak throughput by 1.4X ~ 15.6X.

The rest of the paper is organized as follows. We present the background and related work in Section 2. Section 3 gives a system overview of SlimChain. Section 4 presents transaction processing and sharding techniques in our design. Sections 5 and 6 present the implementation details and the experimental results, respectively. Finally, we conclude our paper in Section 7.

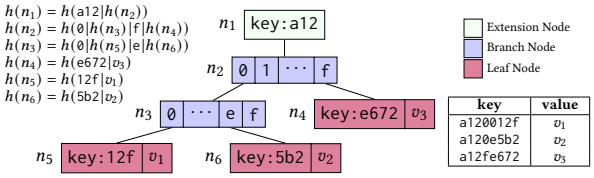

Figure 1: Merkle Patrica Trie

2 BACKGROUND AND RELATED WORK

In this section, we give some background necessary for introducing SlimChain, including cryptographic preliminaries, blockchain basics, and concurrency control methods. We also provide a review of related blockchain systems and highlight the novelty of SlimChain.

2.1 Cryptographic Preliminaries

Cryptographic Hash Function. A cryptographic hash function $h(\cdot)$ accepts an arbitrary-length string as its input and returns a fixed-length bit string. It is collision resistant and difficult to find two different messages m_1 and m_2 such that $h(m_1) = h(m_2)$. Classic cryptographic hash functions include the SHA-2 and SHA-3 families.

Merkle Patricia Trie. A Merkle Hash Tree (MHT) [14] is an authenticated data structure for storing key-value pairs $\langle k_i, v_i \rangle$. It supports verifiable membership testing with logarithmic complexity. The MHT stores the data in an index tree, where each node is assigned with a digest based on its hashed data value as well as its child nodes. Subsequently, the root digest of the MHT can be used to authenticate all the underlying data. For example, in Fig. 1, a proof to attest to value v_1 (resp. v_3) consists of the tree path $\{n_1, n_2, n_3, n_5, h(n_4), h(n_6)\}$ (resp. $\{n_1, n_2, n_4, h(n_3)\}$). During the verification, one can reconstruct the root digest and compare it with the published value.

In blockchain systems, a trie is often used as the index tree structure to reduce the storage cost [6]. As shown in Fig. 1, there are three kinds of nodes in the trie: (i) *extension node*, which stores a slice of the search key and a child node; (ii) *branch node*, which branches out to a fixed number of child nodes; and (iii) *leaf node*, which stores the remaining search key and its corresponding value. If a search key is not present in the trie, the corresponding value is defined as the default value, such as 0. This enables us to store the values with a huge key space (e.g., 2^{256} for address space) in a compact size. The Merkle Patricia trie is a specialized MHT that uses the aforementioned trie as the index structure. For brevity, hereafter we refer to Merkle Patricia trie as Merkle trie.

Merkle Multiproof. When accessing multiple values in a Merkle trie, a Merkle multiproof is usually used to reduce the proof size. Instead of returning a Merkle tree path for each accessed value, we can combine them based on their common ancestors. For example, considering a case where we want to access values v_1 and v_3 simultaneously in Fig. 1, the Merkle multiproof will consist of $\{n_1, n_2, n_3, n_4, n_5, h(n_6)\}$. There is no need to return the hash values $h(n_3)$ and $h(n_4)$, which can be computed locally during the verification.

Verifiable Computing. Verifiable computing (VC) is a method for securing the integrity of computations performed by untrusted parties. Formally, a VC scheme consists of the following algorithms:

- KeyGen$(1^\lambda, F) \rightarrow (EK_F, VK_F)$: The key generation algorithm takes the computation task (function) F and the security parameter λ as input, and outputs an evaluation key EK_F and a public verification key VK_F.
- Compute$(EK_F, u) \rightarrow (y, \pi_y)$: The compute algorithm uses the evaluation key EK_F and an input u to compute the result $F(u) \rightarrow y$ and a proof π_y for verifying y's integrity.
- Verify$(VK_F, u, y, \pi_y) \rightarrow \{0, 1\}$: Given the verification key VK_F, the input u, the result y, and the proof π_y, the verification algorithm outputs 1 if $F(u) = y$, and 0 otherwise.

There are two general approaches to implementing a VC scheme: (i) cryptography-based solutions and (ii) secure-hardware-based solutions. For cryptography-based solutions, pioneered by the SNARKs scheme [15, 16, 17], a cryptographic proof is generated using a circuit-based structure derived from the computation task. Owing to the cryptographic primitives, the proof generation usually suffers from high computation overheads. In contrast, secure-hardware-based solutions are more efficient. Arbitrary computation tasks can be executed inside a trusted execution environment (TEE), such as the Intel Software Guard eXtension (SGX), in an integrity-assured and privacy-preserving manner [18, 19]. Specifically, EK_F encodes a private key secretly embedded in the TEE hardware, and VK_F encodes the TEE's public key. The proof is computed as the result's attestation signature signed by EK_F, and the client can verify the signature using VK_F, to ensure the computation integrity. There are several existing works which propose to utilize VC schemes in blockchain off-chain transaction execution [20, 21].

2.2 Blockchain Basics

Blockchain Data Structure. A blockchain consists of a chain of blocks that maintain a set of world states and record the transactions that modify those states. While blockchain nodes are mutually untrusted, a consensus protocol (e.g., Proof of Work [3], Proof of Stake [22], and PBFT [23]) is used to order transactions globally so that each node has the same view of the world states.

Figure 2 shows the block data structure in a classic (stateful) blockchain system. In a nutshell, the header of each block consists of four fields: (i) $H_{\text{prev\_blk}}$, which is the hash of the previous block; (ii) π_{cons}, which is the data corresponding to the consensus protocol (e.g., *nonce* computed by the miners in Proof-of-Work systems); (iii) $H_{\text{tx\_root}}$, which is the root hash of the transactions in the current block; and (iv) $H_{\text{state\_root}}$, which is the root hash of the Merkle trie corresponding to the current world states. In addition to the block header, each block also stores the original transactions and the state Merkle trie. These data are replicated in every node in the blockchain network, which leads to significant storage overheads. Furthermore, every node needs to replay all the transactions in order to update the current world states, which incurs a high maintenance cost. To alleviate these problems, as mentioned in Section 1, this paper designs SlimChain to offload as much data as possible to off-chain nodes. More details will be given in Section 3.

Permissionless vs. Permissioned Blockchain. Blockchains can be broadly categorized into two types, i.e., permissionless and permissioned. In a permissionless blockchain (e.g., Bitcoin [3] and Ethereum [6]), anybody is allowed to participate in the network and the consensus process. On the other hand, a permissioned blockchain (e.g., Hyperledger Fabric [12], R3 Corda [24], and Quorum [25]) regulates who can initiate transactions and participate in the consensus mechanism. Compared to permissionless blockchains, permissioned blockchains usually achieve higher throughput and lower consensus latency, but are less decentralized.

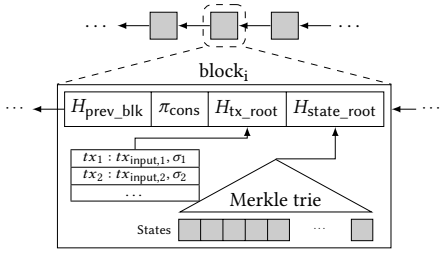

Figure 2: Stateful Block Data Structure

2.3 Concurrency Control Methods

Concurrency control is to maintain data consistency and serializability despite concurrent execution of transactions. SlimChain supports the following two prevalent concurrency control methods.

Optimistic Concurrency Control (OCC). OCC assumes that multiple transactions can often complete without interfering with each other. Thus, it simply checks whether other committed transactions have modified the data that the current transaction accessed (read or wrote). If so, the current transaction is aborted. OCC is more suitable for workloads with low data contention.

Serializable Snapshot Isolation (SSI). Snapshot Isolation (SI) executes a transaction in a consistent snapshot of the database. The commit will be successful only if the values updated by the transaction have not been changed externally since the snapshot was taken. However, the serializability may not be guaranteed due to SI anomalies [35]. SSI remedies this by constructing a dependency graph [36]. In the graph, each vertex represents a transaction; an edge from T_1 to T_2 indicates that T_1 is preceded by T_2 in serial order. There are three types of edges: (i) *rw-dependency*, (ii) *wr-dependency*, and (iii) *ww-dependency*. In practice, the latter two can be ignored due to the use of locks during writes. As long as there is no cycle in the graph, the transactions are serializable. To improve the efficiency of serializability checking, instead of using the graph, each transaction is augmented with two flags to track whether there is a *rw-dependency* pointing to or originating from the transaction [37]. If both flags are set, the transaction is aborted.

2.4 Related Work

Table 1 shows a comparison of several related systems alongside our own proposed system, SlimChain. In the section below, we briefly review these systems in terms of stateless design, support of concurrent transactions, and other features, such as smart contracts, permissionlessness, and sharding.

Most existing blockchain systems are based on a stateful design that stores all state data on-chain. Ethereum [6], the first permissionless blockchain that supports smart contracts, does not allow parallel transaction execution and suffers from low scalability. To improve the concurrency of transaction processing, Hyperledger Fabric [12], a permissioned blockchain with smart contract func-

Table 1: Comparison of SlimChain with Existing Blockchain Systems

System	Stateless	Transaction Execution			Features		
		Parallel Execution	Concurrency Control	Serializability[1]	Smart Contract	Permissionless	Sharding
Ethereum [6]	✗	✗	N/A	Strongly Serializable	✓	✓	✗
Fabric [12]	✗	✓	OCC	Strongly Serializable	✓	✗	✗
Fabric++ [26]	✗	✓	OCC + TX reordering	Strongly Serializable	✓	✗	✗
AHL [27]	✗	✓	OCC	Strongly Serializable	✓	✗	✓
[28]	✗	✓	SSI	Serializable	✓	✗	✗
Fabric# [29]	✗	✓	SSI + TX reordering	Serializable	✓	✗	✗
FastFabric [30]	✗	✓	OCC	Strongly Serializable	✓	✗	✗
XOXFabric [31]	✗	✓	SSI	Serializable	✓	✗	✗
[32, 33]	✗	✗	N/A	Strongly Serializable	✓	✗	✓
[9, 10, 11]	✓	✗	N/A	Strongly Serializable	✗	✓	✗
[34]	✓	✗	N/A	Strongly Serializable	✓	✓	✗
SlimChain	✓	✓	OCC or SSI	Strongly Serializable[2]	✓	✓	✓

[1] Strongly serializable means that the on-chain transaction commit order is the same as its serialization order.
[2] SlimChain achieves strong serializability when OCC is used and normal serializability when SSI is used.

tionality, employs the execute-then-order paradigm, in which multiple endorsing peers are allowed to execute transactions in parallel, and the execution results are then ordered and committed sequentially by committing peers. However, this system supports only optimistic concurrency control (OCC), which suffers from a high abort rate when there is a high data contention. To remedy this, Sharma et al. [26] propose transaction reordering to improve OCC, and Nathan et al. [28] consider another concurrency control method, Serializable Snapshot Isolation (SSI). Ruan et al. [29] theoretically analyze the execute-then-order paradigm and suggest SSI with transaction reordering to further boost performance. On the other hand, FastFabric [30] suggests several novel architectural optimizations to reduce I/O and computation overheads in Fabric. It is further enhanced by XOXFabric [31], where a re-execution phase is introduced to handle aborted transactions. There are also studies proposing improved consensus protocols [32, 33].

Recently, a stateless blockchain paradigm has been proposed for permissionless settings, in which the validation states are moved off the blockchain [9, 10, 11, 34]. However, [9, 10, 11] all target on cryptocurrency applications and their methods cannot be applied to our setting with smart contracts. Specifically, they can support only simple cryptocurrency transfer transactions, but not the arbitrary computation logic required by smart contracts. Additionally, they only consider the state maintenance for account balances, which is insufficient for smart contracts as an unbound number of states may be read or write during smart contract executions. Gorbunov et al. [34] propose a scheme called Pointproofs, which can be used for stateless blockchains with smart contract functionality. However, all of these works only support sequential transaction processing in their stateless design. In comparison, in SlimChain, smart contracts can be concurrently executed by off-chain nodes with a VC scheme to ensure execution integrity. They can then be validated and committed by stateless on-chain nodes even if they are submitted asynchronously.

Furthermore, sharding is a viable approach to reduce state maintenance overheads by partitioning the blockchain ledger into different shards [7, 8, 27, 32]. In existing systems, sharding is often implemented by creating several parallel sub-chains, where each chain concerns only a subset of the smart contracts. However, such a design has at least three disadvantages. First, existing sharding techniques cannot completely address the state maintenance problem owing to the limited number of shards one can create. Second, handling cross-shard transactions requires a cumbersome two-phase commit protocol (2PC), which introduces long latency and cannot well handle forking events. Third, since sharding also partitions the consensus among the nodes in some existing systems, this actually leads to degraded security in the consensus layer. For example, in RapidChain [7], a costly reconfiguration protocol needs to take place to prevent newly joined nodes from breaching the threshold of faulty nodes in some shard. In contrast, as we will show later, a brand new sharding method can be integrated with SlimChain to optimize off-chain storage performance in a novel way.

In summary, existing blockchain systems do not support stateless data storage, smart contract functionality, parallel transaction execution, and sharding techniques all at the same time. SlimChain is the first system that supports all of these desired features.

3 SLIMCHAIN OVERVIEW

In this section, we provide an overview of SlimChain, the proposed blockchain system for the enabling of scalable transaction processing. We focus on general-purpose blockchain systems with smart contract capabilities.

3.1 Design Goals

We aim to achieve the following design goals in SlimChain:

- *Minimizing the maintenance burden of blockchain nodes.* As discussed in the previous sections, reducing the storage and computation overheads of blockchain nodes is essential for improving system scalability and robustness.
- *Supporting parallel transaction execution.* To maximize system throughput, we should allow transactions to be executed in parallel by different nodes. Meanwhile, we should be able to resolve conflicts and establish a consensus on the order of transactions among all nodes.
- *Supporting effective sharding.* Sharding has been demonstrated to be an effective solution to maintaining high performance in classic stateful blockchain systems. We should integrate sharding into the stateless blockchain design.
- *Retaining system security.* System security should not be impaired. We should follow the same security assumptions that

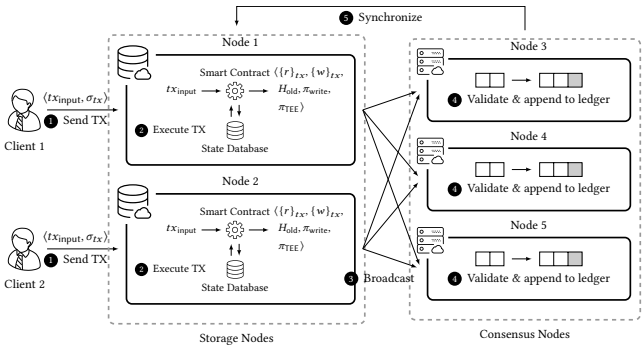

Figure 3: System Model

existing blockchain systems do. For example, for a permissioned blockchain, as long as the ratio of Byzantine nodes does not exceed the threshold of the underlying consensus protocol, the integrity of the blockchain should be guaranteed among the honest nodes. On the other hand, we will allow off-chain nodes to behave arbitrarily.

3.2 System Overview

To meet the design goals, SlimChain maintains only the short commitments of ledger states on-chain, whereas the stateful data is stored off-chain in dedicated nodes. Figure 3 presents an overview of the system model, which consists of three types of nodes connecting each other in an asynchronous network.

- *Clients*, which can invoke smart contracts by sending transaction requests to the blockchain network.
- *Consensus nodes*, which run a consensus protocol and collaboratively maintain a consensus view of the blockchain ledger. They can take two different roles: (i) block proposers, a.k.a. miners, which are tasked to generate new blocks; and (ii) block observers, which participate in the consensus by only observing and validating new blocks.
- *Storage nodes*, which are off-chain nodes with relatively high storage and computation capabilities. In addition to synchronizing on-chain commitments, they are also dedicated to maintaining off-chain stateful data as well as executing transactions.

Block Data Structure. To reduce the storage burden of the consensus nodes and offload as much data as possible to the storage nodes, we design a new stateless block data structure, as shown in Fig. 4. Compared to the classic block structure shown in Fig. 2, there are two main differences. First, we replace transactions with their corresponding digests, H_{tx_i}, as the on-chain transaction data. The storage nodes are instead responsible for keeping the full transaction data. Owing to the cryptographic hash function, anyone can use the on-chain digests to ensure that the transaction data is not tampered with when being retrieved from the untrusted storage nodes. Second, the entire world states and the corresponding Merkle trie are also moved to the storage nodes. We keep only the root hash of the Merkle trie, $H_{\text{state\_root}}$, on-chain for integrity assurance. Similar to the transaction data, anyone can use this root hash $H_{\text{state\_root}}$ to verify the integrity of the state data retrieved from the storage nodes. In the above design, the consensus nodes in SlimChain are lightweight and comparable to light nodes in existing blockchain systems in terms of resource demands. However,

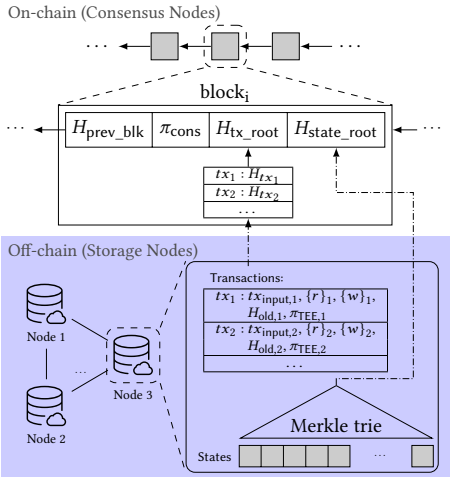

Figure 4: Stateless Block in SlimChain

unlike existing systems, the consensus nodes in SlimChain are still capable of validating the integrity of transaction executions and maintaining the same level of security guarantee as that of full nodes in classic blockchains.

Transaction Processing Workflow. As illustrated in Fig. 3, the transaction processing in SlimChain generally consists of the following steps:

- ❶ To invoke a smart contract, the client sends the transaction request to one of the storage nodes. The request is a tuple $\langle tx_{\text{input}}, \sigma_{tx} \rangle$, where tx_{input} and σ_{tx} represent the transaction input and the corresponding digital signature, respectively.
- ❷ Upon receiving the transaction, the storage node simulates the smart contract execution locally to obtain its read set $\{r\}_{tx}$ and write set $\{w\}_{tx}$.
- ❸ The execution results along with an execution proof π_{tx} and some other auxiliary data are broadcast to the consensus nodes (more details will be discussed in Section 4.1).
- ❹ After validating the execution results, the consensus nodes append the transaction and update the state commitment in the blockchain. This includes proposing a block by block proposers and validating the block by block observers.
- ❺ Finally, the storage nodes will commit to their local state storage, once they observe that the transaction has been included in the blockchain.

In what follows, we propose novel algorithms to enable transaction execution and validation in SlimChain.

4 SLIMCHAIN TRANSACTION PROCESSING

In this section, we discuss how SlimChain processes parallel transactions with stateless blocks. There are three main challenges. First, transactions can no longer be executed by the consensus nodes since they do not maintain the ledger states. At the same time, we cannot simply trust the storage nodes to execute transactions faithfully. Second, the consensus nodes are not able to update the on-chain commitments in a straightforward way due to a lack of necessary information. Third, we need to ensure the ACID properties of committed transactions, since they are executed by the off-chain storage nodes concurrently. We address these three challenges in Sections 4.1 to 4.3. Afterwards, we discuss how to extend

Algorithm 1: Storage Node Transaction Execution (TEE)

Input: transaction request $\langle tx_{\text{input}}, \sigma_{tx} \rangle$, state root in the latest block H_{old}.

1 **if** verify(tx_{input}, σ_{tx}) *failed* **then abort**;
2 $\langle \{r\}_{tx}, \{w\}_{tx} \rangle \leftarrow$ execute(tx_{input}, H_{old});
3 $\pi_{\text{read}} \leftarrow$ get_merkle_proof($\{r\}_{tx}$);
4 **if** verify($\{r\}_{tx}, H_{old}, \pi_{read}$) *failed* **then abort**;
5 $\pi_{\text{TEE}} \leftarrow$ TEE.sign($\langle tx_{input}, \{r\}_{tx}, \{w\}_{tx}, H_{old} \rangle$);
6 **return** $\langle \{r\}_{tx}, \{w\}_{tx}, \pi_{TEE} \rangle$;

our SlimChain system to support sharding in Section 4.4.

4.1 Off-chain Transaction Execution

Because the ledger states are not stored on-chain, a smart contract can only be executed off-chain with the help of storage nodes. To ensure that executions are done faithfully, we can leverage forms of public verifiable systems. That is, the storage nodes are required to supply some additional proof to attest to the integrity of smart contract executions. Given such proof, anyone can verify that the transaction execution results are indeed correct using only publicly available information. There are three ways to achieve this:

- *Cryptography-based solution*: Cryptographically verifiable computation techniques such as SNARKs [16] can be used to construct a verifiable Turing machine. Although they are usually less efficient, they offer the strongest security guarantee.
- *Secure-hardware-based solution*: A trusted execution environment (TEE) is a special secure area of a processor. A TEE provides a security-isolated world for trustworthy program executions on an otherwise untrusted hardware platform. It is highly efficient but relies on additional security assumptions compared with pure cryptography-based solutions.
- *Policy-based solution*: For permissioned blockchains, a policy-based approach can also be used. For example, one can assume that the execution is faithful as long as there are enough endorsers or auditors to digitally sign the results. This approach offers the highest level of performance, though at the expense of some security protections.

In this paper, we mainly consider using the secure-hardware-based solution, specifically Intel SGX powered TEE [19]. It has the advantage of both offering high performance and requiring less trust. Nevertheless, it is worth noting that our transaction commitment and node synchronization schemes proposed in Sections 4.2 and 4.3 can work with either of the three solutions mentioned above.

To support verifiable off-chain transaction execution and provide enough information to be used by the consensus nodes for stateless on-chain transaction commitment, we develop a novel TEE-based storage node transaction execution algorithm. As shown in Algorithm 1, it accepts two inputs: (i) the transaction request from a client, and (ii) the state root in the latest block observed by the current storage node (denoted as H_{old}). H_{old} represents a snapshot of the current world states. It serves two purposes. On one hand, it creates a snapshot isolation environment w.r.t. H_{old} for the smart contract execution. On the other hand, as to be shown later, H_{old} serves as an anchor point to attest to the integrity of the transaction read set. To execute the smart contract, the transaction request is firstly verified against its digital signature (Line 1). Then, the smart contract is executed normally with respect to the input and

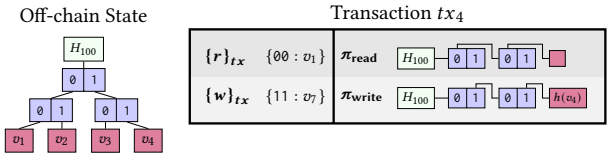

Figure 5: An Example of Transaction Execution

the state root. During the execution, we record the read set $\{r\}_{tx}$ and the write set $\{w\}_{tx}$ (Line 2). To ensure that the read values obtained from outside TEE are correct, a Merkle multiproof with respect to the read set is generated outside TEE (Line 3). Inside TEE, the Merkle root is reconstructed using this proof and the recorded read set $\{r\}_{tx}$. The verification passes if this computed Merkle root matches with the state root from the input H_{old} (Line 4).[4] Finally, TEE signs a digital signature π_{TEE} using the technique known as remote attestation [19] (Line 5). Note that the signature is created with respect to the transaction input, the read/write sets, and the original state root H_{old} to prevent any tampering.

Outside TEE, the storage node also computes an additional Merkle proof (denoted as π_{write}) with respect to the written addresses in the write set $\{w\}_{tx}$. This is needed to facilitate the stateless consensus nodes to update the on-chain state root. We will discuss this in more detail in the next section. Putting all these things together, the off-chain execution result tx_{submit}, which will be broadcast to the consensus nodes for final commitment, has the following components:

$$tx_{\text{submit}} = \langle tx_{\text{input}}, \{r\}_{tx}, \{w\}_{tx}, H_{\text{old}}, \pi_{\text{TEE}}, \pi_{\text{write}} \rangle.$$

Example. *Figure 5 shows an example of transaction execution. Suppose that the current off-chain state trie is the one shown in the left. After verifying the signature of the transaction request, the transaction tx_4 is executed with respect to the current state root H_{100}. Assume that we obtain the read set $\{00 : v_1\}$ and the write set $\{11 : v_7\}$. A Merkle proof π_{read} as shown in the figure is computed from the current off-chain state and read set. Using this proof and the read value, the TEE recomputes the Merkle root H_{100} to verify the integrity of the transaction reads. After verification, the value in the read set is discarded, leaving the read set to be $\{00\}$. Then, TEE signs the execution result in the form of π_{TEE} = TEE.sign($\langle tx_{input}, \{00\}, \{11 : v_7\}, H_{100} \rangle$). Finally, the storage node computes another Merkle proof π_{write} with respect to the write set as shown in the figure, and broadcasts the final execution result as $tx_{submit} = \langle tx_{input}, \{00\}, \{11 : v_7\}, H_{100}, \pi_{TEE}, \pi_{write} \rangle$.*

4.2 On-chain Transaction Commitment

After collecting a certain number of off-chain transaction execution results, the block proposers of the consensus nodes can bundle them together to generate a new block. In other words, they are responsible for ordering and committing the transactions on-chain.

4.2.1 Solution Overview. There are two major obstacles for stateless transaction commitment. On one hand, the consensus nodes need to validate then commit the transactions despite of being stateless. In particular, the root of the Merkle trie $H_{\text{state\_root}}$ needs to be updated by the consensus nodes without access to the full trie. On the other hand, due to the nature of the asynchronous network,

[4] After the verification of π_{read}, we only need to keep track of the read addresses in the read set $\{r\}_{tx}$. In comparison, the write set $\{w\}_{tx}$ includes both the written addresses and written values.

transactions from the storage nodes may arrive in any arbitrary order. This entails us to design algorithms such that they are able to not only handle concurrency control to ensure ACID properties among parallel transactions, but also ensure the on-chain data being collectively maintained and perfectly synchronized among all nodes in the blockchain network. To this end, we propose to maintain some minimal yet enough auxiliary information on the consensus nodes to follow the state of the Merkle trie and track the dependencies between different transactions.

To make the aforementioned on-chain auxiliary information as small as possible, we make two important observations. First, the longer the latency between the transaction execution and the arrival on the consensus nodes, the higher the probability of a conflict. Second, there is no need to keep the stateful data forever. Thus, we only need the information for a few of the most recent blocks. We let each consensus node keep track of a minimal amount of temporary stateful data for the most recent k blocks. The setting of on-chain temporary state length k is a system parameter shared by all parties in the network. If a transaction received by the consensus nodes was executed based on a block whose age is more than k blocks, the consensus nodes simply discard the transaction. Note that the temporary data serves two purposes: detecting read/write conflicts and updating the on-chain state root. Furthermore, it should also support incremental updates since the on-chain transaction commitment happens in sequential order and has a big impact on the system performance. Therefore, we design the temporary state to include five components: (i) A map between the block height to the corresponding read addresses set, denoted as $M_{i\mapsto r}$; (ii) A map between the block height to the corresponding written addresses set, denoted as $M_{i\mapsto w}$; (iii) A map between the read addresses to an ordered list of block heights, denoted as $M_{r\mapsto i}$; (iv) A map between the written addresses to an ordered list of block heights, denoted as $M_{w\mapsto i}$; and (v) A partial Merkle trie which only records the write set that happened in the past k blocks along with their Merkle paths, denoted as $\mathcal{T}_w$.

Algorithm 2 describes the overall procedure for committing the transactions. We will go through the pending transactions one by one. Consider one transaction. First, it checks that the transaction is indeed recent enough (Line 4) and has a valid TEE signature (Line 5) as well as a valid write proof π_{write} (Line 6). Next, it checks whether the transaction has any read or write conflict with other committed transactions (Line 7). Then, we merge π_{write} and $\{w\}_{tx}$ into the partial Merkle trie $\mathcal{T}_w$ (Line 8). At the same time, we can update the four maps stored in the temporary states to include the new transaction. Upon applying the above algorithm to all the transactions, a new block is computed. The new block includes a list of digests of the valid transactions and a new state root computed from $\mathcal{T}_w$ (Line 13). This proposed new block will later be passed to the consensus protocol for ordering. Finally, we remove the information related to the k-th recent block from the temporary states. This is done by removing the k-th block from $M_{r\mapsto i}$ and $M_{w\mapsto i}$. If an address has no corresponding block in these two maps, it is removed completely from the maps. For the removed written addresses, we also remove their associated tree paths from $\mathcal{T}_w$ (Line 22).

4.2.2 Partial Merkle Trie Maintenance. The core design of our proposed temporary state is a novel partial Merkle trie $\mathcal{T}_w$, which

Algorithm 2: Consensus Node Transaction Commitment

Input: transactions executed by the storage nodes $\{tx_{\mathsf{submit}}\}$, current block height i.

1 **for** $tx \in \{tx_{submit}\}$ **do**
2 $\quad\langle tx_{\mathsf{input}}, \{r\}_{tx}, \{w\}_{tx}, H_{\mathsf{old}}, \pi_{\mathsf{TEE}}, \pi_{\mathsf{write}}\rangle \leftarrow tx$;
3 $\quad j \leftarrow$ `get_block_height`(H_{old});
4 $\quad$ **if** $i - j > k$ **then continue**;
5 $\quad$ **if** `verify`$(\langle tx_{input}, \{r\}_{tx}, \{w\}_{tx}, H_{old}\rangle, \pi_{TEE})$ *failed* **then continue**;
6 $\quad$ **if** `verify`(H_{old}, π_{write}) *failed* **then continue**;
7 $\quad$ **if** `check_conflict`$(\{r\}_{tx}, \{w\}_{tx}, j)$ *failed* **then continue**;
8 $\quad$ Update $\mathcal{T}_w$ using π_{write} and $\{w\}_{tx}$; `// See Alg. 3`
9 $\quad M_{i\mapsto r}[i+1]$.`append`$(\{r\}_{tx})$;
10 $\quad M_{i\mapsto w}[i+1]$.`append`$(\{w\}_{tx})$;
11 $\quad$ **for** $r \in \{r\}_{tx}$ **do** $M_{r\mapsto i}[r]$.`append`$(i+1)$;
12 $\quad$ **for** $w \in \{w\}_{tx}$ **do** $M_{w\mapsto i}[w]$.`append`$(i+1)$;
13 Compute $H_{\mathsf{state\_root}}$ from $\mathcal{T}_w$, generate new block block_{i+1};
14 **for** $r \in M_{i\mapsto r}[i-k+1]$ **do**
15 $\quad M_{r\mapsto i}[r]$.`remove`$(i-k+1)$;
16 $\quad$ **if** $M_{r\mapsto i}[r]$ *is empty* **then** $M_{r\mapsto i}$.`remove`(r);
17 $M_{i\mapsto r}$.`remove`$(i-k+1)$;
18 **for** $w \in M_{i\mapsto w}[i-k+1]$ **do**
19 $\quad M_{w\mapsto i}[w]$.`remove`$(i-k+1)$;
20 $\quad$ **if** $M_{w\mapsto i}[w]$ *is empty* **then**
21 $\quad\quad M_{w\mapsto i}$.`remove`$(w)$;
22 $\quad\quad \mathcal{T}_w$.`remove`$(\textit{path associated to } w)$; `// See Alg. 4`
23 $M_{i\mapsto w}$.`remove`$(i-k+1)$;

enables the consensus nodes to update the state root digest without accessing the full Merkle trie. The partial trie has an identical structure to the full trie stored by the storage nodes, so they share the same root digest. However, the majority of tree nodes in $\mathcal{T}_w$ are suppressed. Instead, only the tree nodes corresponding to the written values happening in the past k blocks as well as their Merkle paths are materialized. The partial trie supports two operations, namely update and tidy. The update operation takes the Merkle proof π_{write} and write set $\{w\}_{tx}$ to apply the writes from the transaction. Note that the proof contains the original Merkle trie paths corresponding to the written addresses when the transaction is executed, whereas the write set records the written values. A simplified version of the algorithm is shown in Algorithm 3, which assumes that there are only branch nodes and leaf nodes without the remaining search key in the Merkle trie. It traverses both the partial trie and the proof in a top-down fashion to find the proper places to insert the missing subtrees and the written values. For normal Merkle trie, the full version of the algorithm is shown in Appendix A . On the other hand, the tidy operation is used to remove the write addresses whose age is more than k blocks. This is needed to keep the size of partial Merkle trie $\mathcal{T}_w$ small. Its algorithm is shown in Algorithm 4. We start by comparing the removing address with the rest of the addresses in the write set map $M_{w\mapsto i}$ to find the maximum common prefix length (Line 1). Then, we simply remove all nodes who are on the Merkle path corresponding to the removing address and have a tree depth larger than the previous found prefix length.

Example. *Figure 6 shows a full example of transaction commitment. In this example, the on-chain temporary state length k is 2, which means that the consensus nodes only keep track of the partial states for the last two blocks. Assume that the current block height is*

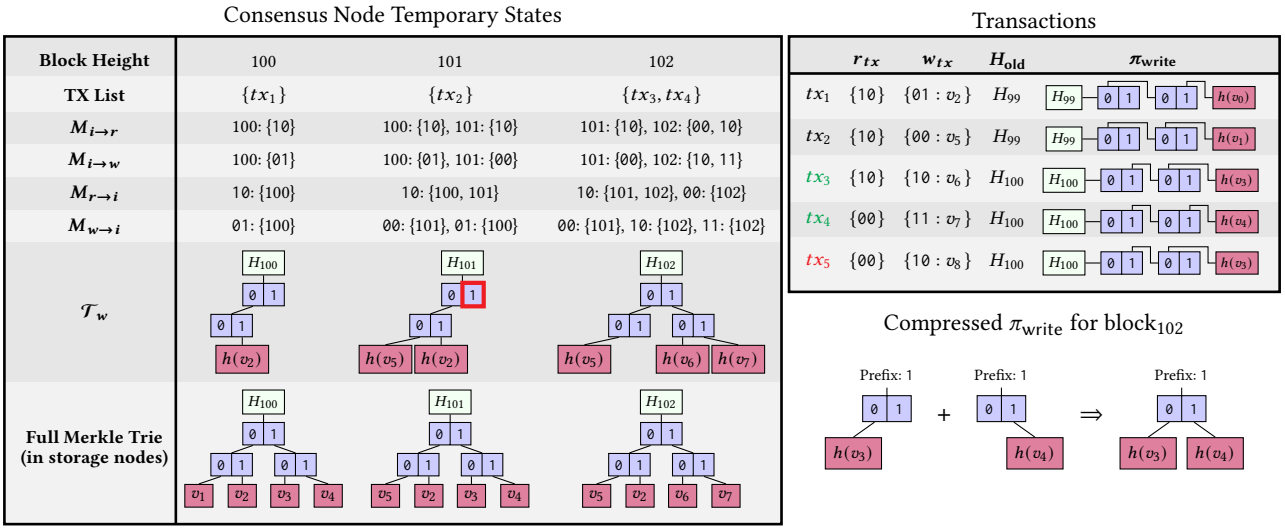

Figure 6: An Example of Transaction Commitment

Algorithm 3: Update Partial Merkle Trie $\mathcal{T}_w$ (Simplified)

Input: write set Merkle proof π_{write}, write set $\{w\}_{tx}$.

```
Q ← init_queue();
Q.enqueue(⟨π_write.root(), T_w.root()⟩);
while not Q.is_empty() do
    ⟨n_π, n_T⟩ ← Q.dequeue();
    for ⟨n_1, i⟩ ∈ n_π.child_nodes_with_index() do
        n_2 ← n_T.child[i];
        if n_2 is suppressed then
            Update n_1's subtree using values from {w}_tx;
            Replace n_2 with n_1;
        else Q.enqueue(⟨n_1, n_2⟩);
```

Algorithm 4: Tidy Partial Merkle Trie $\mathcal{T}_w$

Input: address to be removed w, write set map $M_{w\mapsto i}$.

```
len ← max{common_prefix_len(w, w') | ∀w' ∈ M_{w↦i}};
node ← T_w.root(); depth ← 0;
while depth < len do
    node ← node.child w.r.t. address w; depth ← depth + 1;
Replace node with its digest value;
```

101, and a consensus node is working to create block 102. There are three candidate transactions $\{tx_3, tx_4, tx_5\}$, all of which are executed by the storage nodes using block 100 as the starting point.

Suppose that tx_3 and tx_4 are the only transactions passing both the validation of π_{TEE}, π_{write} and the concurrency control, a consensus node now needs to commit them into the partial Merkle trie $\mathcal{T}_w$ for the purpose to compute the state digest for the new block 102. First, the consensus node uses π_{write}'s of tx_3 and tx_4 to materialize the missing tree nodes in the partial trie $\mathcal{T}_w$. The missing tree nodes all lie in the Merkle paths corresponding to keys 10 and 11. Next, the write sets $\{10 : v_6\}$ and $\{11 : v_7\}$ of these two transactions are applied to update the written values $\{v_6, v_7\}$. This updated partial trie is then used to compute the new state root H_{102} and subsequently to create the new block 102. Finally, the consensus node removes the unneeded tree path associated with keys whose age is more than 2 blocks from the partial trie to keep the trie size small. In our example, it is the path associated

Algorithm 5: Check Read/Write Conflict (OCC)

Input: read set $\{r\}_{tx}$, write set $\{w\}_{tx}$, block height j which corresponds to H_{old}.

```
for r ∈ {r}_tx do
    if ∃i s.t. i ∈ M_{w↦i}[r] ∧ i ≥ j then return failed;
for w ∈ {w}_tx do
    if ∃i s.t. i ∈ M_{w↦i}[w] ∧ i ≥ j then return failed;
return success;
```

to key 01 (i.e., leaf node $h(v_2)$).

4.2.3 Concurrency Control. Our system supports two methods of concurrency control, i.e., Optimistic Concurrency Control (OCC) and Serializable Snapshot Isolation (SSI) using the heuristic from [37]. Both of them only rely on the minimal amount information stored in the temporary state. When OCC is used, we simply check whether any read or write set of the transaction has been updated by other more recently committed transactions. In comparison, we check two criteria in SSI: (i) whether any part of the write set is also updated by other transactions; and (ii) whether there are *rw-dependencies* both pointing to and originating from the current transaction. Similar to traditional DBMS, we do not need to keep track of either *wr-dependency* or *ww-dependency*. This is because the transactions are committed in a sequence with no concurrent writes. It is also worth noting that the order of transactions in the blockchain may not represent their execution order in SSI. Instead, each transaction is considered to start at the block corresponding to H_{old} and to be committed in the block where it is stored. The detailed concurrency control algorithms are shown in Algorithms 5 and 6.

Example. *In the previous example shown in Fig. 6, the consensus node needs to check whether there is any conflict between candidate transactions (i.e., $\{tx_3, tx_4, tx_5\}$) and the transactions already committed in the last two blocks. It is easy to see that tx_3 satisfies the serializability requirement. For tx_4, since it reads a value at key 00 during off-chain execution (i.e., block$_{100}$), which is later written by tx_2 committed in block$_{101}$, it will be aborted when OCC is used. However, it can still be committed when SSI is used. In this case, tx_4 is considered to be executed before tx_2 although it is committed in a*

Algorithm 6: Check Read/Write Conflict (SSI)

Input: read set $\{r\}_{tx}$, write set $\{w\}_{tx}$, block height j which corresponds to H_{old}.

$flag_1 \leftarrow false, flag_2 \leftarrow false$;
for $w \in \{w\}_{tx}$ **do**
  if $\exists i$ *s.t.* $i \in M_{w \mapsto i}[w] \wedge i \geq j$ **then return** *failed*;
  if $\exists i$ *s.t.* $i \in M_{r \mapsto i}[w] \wedge i \geq j$ **then** $flag_1 \leftarrow true$;
for $r \in \{r\}_{tx}$ **do**
  if $\exists i$ *s.t.* $i \in M_{w \mapsto i}[r] \wedge i \geq j$ **then** $flag_2 \leftarrow true$;
if $flag_1 \wedge flag_2$ **then return** *failed* **else return** *success*;

later block. The transaction tx_5, on the other hand, is not serializable under either of the concurrency control methods.

4.3 Node Synchronization

After the consensus nodes create a new block, it needs to be synchronized with the rest of the network. There are three kinds of nodes to be considered, namely block observers, storage nodes, and newly joined nodes.

4.3.1 Block Observers. As discussed in Section 3.2, only a subset of the consensus nodes, known as block proposers, are responsible for creating the new block. On the other hand, the block observers merely validate and log the blocks created by the block proposers. Here, they can simply run the same algorithms introduced in Section 4.2 to validate the transactions associated with the new block. If any of the transactions fails the validation, the proposed block is considered to be invalid and is discarded. Depending on the underlying consensus protocol, a view change procedure may also be triggered when an invalid block is observed from the block proposers.

Since the network transmission plays a big role in blockchain bottlenecks [13], it is important to reduce the amount of data sent to the block observers. One idea is to reduce the size of the Merkle proof π_{write}. Instead of using the proof computed by the storage nodes at the time of transaction execution, the block proposers can compress it with three means. First, through comparison with the original proof that is computed based on the Merkle trie corresponding to H_{old}, the compressed one is built based on the latest block, i.e., the one immediately preceding the current block. This also has the benefit of reducing the block observers' computation overheads during updating $\mathcal{T}_w$. Second, the compressed proof does not need to start from the root of the Merkle trie. All tree nodes that are shared with the previous partial trie $\mathcal{T}_w$ can be omitted and be replaced with a common prefix of the search keys to save bandwidth. Third, instead of returning one proof per transaction in the block, the block proposers can bundle them together into one Merkle multiproof. The algorithm to compute this compressed proof for a single transaction is similar to Algorithm 3. The only difference is that instead of finding the proper places to insert the missing subtrees (Lines 8 to 9 in Algorithm 3), it extracts these subtrees along with their prefix of the search key to generate the compressed proof. To produce the compressed proof for multiple transactions in a block, the block proposer can invoke the above procedure to create a compressed proof for each individual transaction. Then these proofs can be merged by combining subtrees that share the same prefix to create the final compressed proof. When the block observers receive the compressed proof, it is no longer verified against the old state root H_{old} but against the most recent partial trie $\mathcal{T}_w$. Apart from this, the rest of the algorithm is identical to Algorithm 2.

Example. *In the example shown in Fig. 6, the compressed proof is illustrated in the bottom right corner. This proof uses the common prefix* 1 *to represent its location in the partial Merkle trie $\mathcal{T}_w$. It is computed by first comparing the differences between π_{write}'s of tx_3 and tx_4 with the partial trie $\mathcal{T}_w$ in the previous block then combining these differences into one proof. When received by the block observers, the Merkle root hash of the subtree in the proof is computed and compared with the hash value stored locally in the corresponding node of $\mathcal{T}_w$ (highlighted by the red rectangle) to ensure the integrity of the compressed proof.*

4.3.2 Storage Nodes. The storage nodes follow similar behaviors to the consensus nodes. That is, they execute the same procedure described in Section 4.2 to commit the transactions ordered by the consensus protocol. At the same time, they also store the relevant data instead of just their digests. This includes both the transaction data and the state data. For the transaction data, the storage nodes record the transaction input tx_{input}, the read/write set $\langle \{r\}_{tx}, \{w\}_{tx} \rangle$, the state root H_{old} that the transaction is executed upon, and the signature π_{TEE} generated by TEE for each transaction. There is no need to store the Merkle proof for the write set π_{write}, because it can be recomputed on-the-fly from the full Merkle trie. As for the state data, the storage nodes maintain the full Merkle trie locally. This is achieved by updating the trie directly during the transaction commitment instead of using the temporary partial trie $\mathcal{T}_w$.

4.3.3 Newly Joined Nodes. When a fresh node joins the network, it has to synchronize and examine all the blocks as well as their transactions, starting from the genesis block. This procedure is quite similar to that of existing blockchain systems, where a newly joined node invokes Algorithm 2 to validate all the blocks downloaded from the blockchain network. Yet, there are two notable differences. First, the newly joined node will retrieve the existing blocks from the storage nodes instead of the consensus nodes. This is due to the fact that only the storage nodes keep a whole copy of the transaction data in our SlimChain design. Second, while it is still required to maintain the temporary states comprising the last k blocks for the read/write set maps in order to check the serializability of the transactions, we can optimize the usage of the partial trie $\mathcal{T}_w$. Instead of invoking the update (Algorithm 3) and tidy (Algorithm 4) operations on $\mathcal{T}_w$ for each block, we can resort to a batching strategy. Specifically, the storage nodes can compute a mega Merkle multiproof to the addresses which will be updated in the next m blocks and send it to the newly joined node. After that, the newly joined node can compute the root state for the next m blocks by applying the relevant write sets to the aforementioned mega multiproof. The parameter m can be dynamically chosen based on network conditions and the memory capacity of the nodes. Clearly, this has the advantage of saving bandwidth by avoiding transmitting duplicate tree nodes.

4.4 Sharding

Sharding is a common technique employed to improve scalability and boost performance for distributed systems. The core idea is to partition the database or states horizontally among different

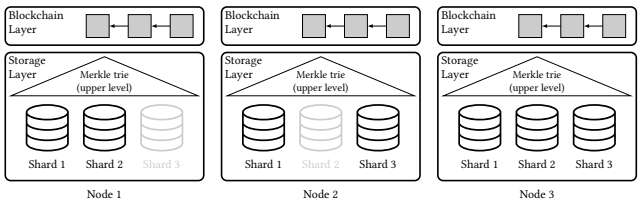

Figure 7: Example of Sharding among Storage Nodes

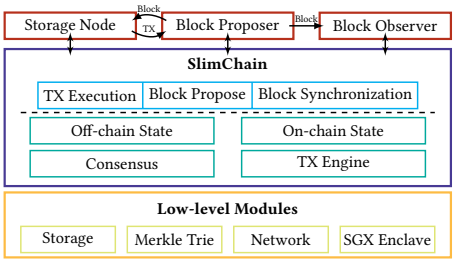

Figure 8: System Architecture of SlimChain

peers. Here, each individual partition is referred to as a shard and is maintained by a separate database instance. Sharding not only has the advantage of reducing the storage cost among individual nodes in the network, but also benefits the transaction execution by spreading the load. To address the issues of existing sharding techniques mentioned in Section 2.4, we propose a brand new sharding method. Instead of creating a fixed number of sub-chains, in SlimChain, sharding only happens among the off-chain storage nodes. Owing to the stateless design, the on-chain consensus nodes are lightweight and thus can forgo the sharding. For the off-chain storage nodes, the sharding is highly flexible and completely dynamic. The storage nodes can choose to store partial or full states, on the basis of their storage capacities. An example of sharding is shown in Fig. 7: Nodes 1 and 2 store only two shards, whereas Node 3 stores the entire replica of the states. Furthermore, nodes can expand or shrink their local storage dynamically based on the needs of the application. For the sake of system efficiency, each node may choose to always store the upper layer of the Merkle trie, which is updated frequently.

During the off-chain transaction execution, there is no special treatment needed for cross-shard transactions. When a transaction involves multiple shards, the client can choose to send it to a storage node that has all the necessary shards; alternatively, multiple storage nodes can work together to process the transaction. In the latter case, one of the storage nodes can be chosen to execute the transaction inside TEE, where the necessary state data is retrieved either from local storage or from the remote nodes that own other shards. The rest of the algorithm is identical to the one described in Section 4.1. Finally, since there is no sharding in the on-chain layer, sharding is completely transparent to the on-chain transaction commitment procedure. As such, not only SlimChain can maintain the same level of security in the consensus layer with or without sharding, but also ensures cross-shard transactions to be committed on-chain in an atomic fashion without any extra latency.

5 IMPLEMENTATION

We implement an end-to-end prototype of SlimChain in the Rust programming language, which consists of around 26,000 lines of codes. The source codes are available at https://github.com/hkbudb/slimchain. Figure 8 shows the system architecture, which consists of three layers.

The bottom layer consists of multiple low-level modules. This includes: (i) *Storage*. We use RocksDB as the data storage;[5] (ii) *Merkle trie*. It offers the functionalities of manipulating the Merkle Patricia Trie. We choose BLAKE2b as the cryptographic hash function used throughout SlimChain; (iii) *Network*. Two different network engines are developed. HTTP protocol is used for communication in the permissioned settings, whereas the libp2p library is used in the permissionless settings. The latter offers the functionalities of node discovery, gossip broadcast, and P2P RPC messaging;[6] (iv) *SGX enclave*. The Rust SGX SDK is used to implement the secure enclave.[7]

The middle layer implements the essential functions of the blockchain system. The off-chain state module manages the states stored in the storage nodes, including the sharding support. Furthermore, the on-chain state module manages the blockchain's temporary states and associated transaction commitment algorithms as discussed in Section 4.2. For the consensus, since our proposed algorithms in SlimChain are not involved in the consensus process, we use a pioneering consensus protocol for permissioned and permissionless settings, respectively. Specifically, the Raft consensus protocol is used in the permissioned setting.[8] On the other hand, Proof-of-Work (PoW) is used for the permissionless system whose mining difficulty is set such that a new block is generated for every 5 ~ 10 seconds. To execute smart contracts, the TX Engine is built based on a Rust Ethereum Virtual Machine, which interprets smart contracts written in Solidity.[9] The TX Engine runs inside an SGX enclave and signs the execution results using an ephemeral key generated inside the same enclave. This ephemeral key is in turn signed by the Intel SGX attestation services.[10] This hierarchical PKI design can significantly improve the performance of SGX remote attestation.

The top layer handles the entire transaction and block processing cycle. Specifically, this includes: (i) off-chain transaction execution, (ii) block proposing, and (iii) block synchronization.

6 PERFORMANCE EVALUATION

In this section, we evaluate the performance of our proposed SlimChain system against three baselines:

- Classic: It follows the design of traditional blockchains like Ethereum [6]. There are only on-chain consensus nodes in the system. Transactions are executed by block proposers during block generation, then by block observers during block validation. All these executions are done sequentially. Since all nodes are stateful, all transaction and state data are replicated by every node.
- Stateful: It is a stateful counterpart of SlimChain, which contains all the components of SlimChain except the partial Merkle trie $\mathcal{T}_w$. The transaction processing is similar to that of SlimChain. However, the entire transaction and state data are directly stored by all nodes during transaction commitment.

[5] https://rocksdb.org
[6] https://github.com/libp2p/rust-libp2p
[7] https://github.com/apache/incubator-teaclave-sgx-sdk
[8] https://github.com/async-raft/async-raft
[9] https://github.com/rust-blockchain/evm
[10] https://api.portal.trustedservices.intel.com

Table 2: System Parameters

Parameters	Permissioned	Permissionless
consensus protocol	Raft	PoW
concurrency control method	OCC, **SSI**	
proof compression optimization (Sec 4.3.1)	Disable, **Enable**	
# of consensus nodes	4, 8, **16**	
# of storage nodes	1, 2, 3, **4**	
maximum TX per block	1,024	4,096
on-chain temporary state length k	16	

- Fabric#: In the permissioned setting, we also compare our system with FabricSharp [29]. The transactions are first executed by the endorsement nodes, who play a similar role as the storage nodes in SlimChain. For fair comparison, the endorsement policy is set to only require the transaction being endorsed by a single endorsement node. Then, transactions are ordered and committed by the consensus nodes. All nodes are stateful and thus store all ledger information.

Noted that both Classic and Stateful are implemented using the same program language and libraries as discussed in Section 5 for fairness.

6.1 Experiment Setup

We deploy the blockchain network in the Azure cloud in the US-East region. The consensus nodes are run on the Standard_D2_v2 machines, whereas the storage nodes or endorsement nodes use the Standard_DC4s_v2 machines.[11] The default network topology consists of 16 consensus nodes and 4 storage nodes or endorsement nodes. The network bandwidth among nodes is 1500 Mbps. Moreover, we enable multithread computation in the storage nodes, where three parallel threads are used to drive the TX engine. The consensus nodes on the other hand run in a single thread. Table 2 lists all the system parameters used in the experiment, where the default values are highlighted in boldface.

Blockbench [38] is used to evaluate the performance. It offers both micro benchmarks consisting of DoNothing (denoted as DN), CPUHeavy (denoted as CPU), IOHeavy (denoted as IO), and macro benchmarks consisting of KVStore (denoted as KV) and SmallBank (denoted as SB). We measure the following metrics to evaluate the proposed system: (i) success rate, the ratio between the number of the committed transactions and the sent transaction requests; (ii) peak throughput, the maximum number of committed transactions per second. It is measured by performing a binary search among varying transaction sending rates; (iii) latency, the average time from sending a transaction request to the transaction commitment in the local consensus node; and (iv) storage size, the average per transaction storage size. Except for Fabric#, we further break the latency down to multiple components, including (i) storage node execution time (denoted as exec); (ii) time of transactions staying idle in the queue of block proposers before being processed (denoted as wait-prop); (iii) block proposing time excluding the consensus process (denoted as propose); (iv) time related to consensus (denoted as raft/mining); (v) block validation time by block observers (denoted as validate); and (vi) network communication time (denoted as net). For each experiment, 300,000 transaction requests are randomly generated using 100,000 sender accounts.

In the following, we first compare the overall performance between SlimChain and baselines when processing different smart contracts under the default system parameters. Then, we present the performance evaluation with varying system parameters. Finally, we investigate the system performance when the proposed sharding technique is used. Due to the space limitation, additional experimental results are presented in Appendix C.

[11] https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-general

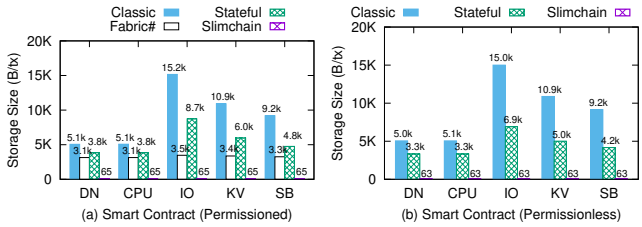

Figure 9: Consensus Node Storage Size (B/tx) vs. Smart Contract

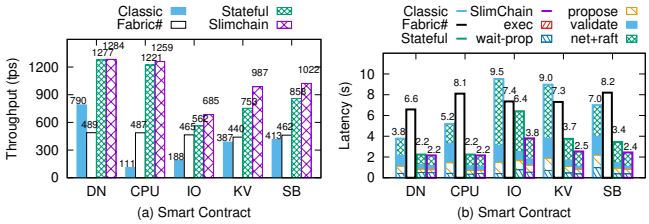

Figure 10: Throughput/Latency vs. Smart Contract (Permissioned)

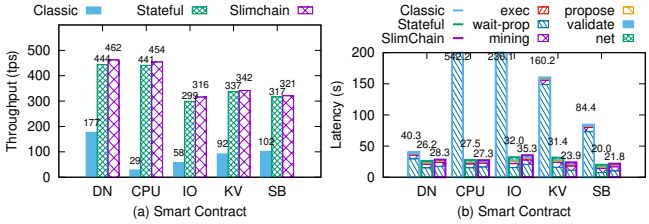

Figure 11: Throughput/Latency vs. Smart Contract (Permissionless)

6.2 Experimental Results

6.2.1 Overall Performance. Figure 9 shows the average per transaction storage size for the consensus nodes in SlimChain and baselines. We can see that SlimChain reduces the on-chain storage requirements for the consensus nodes by 97% ~ 99%. This is because our *stateless* design shifts most of the storage burden to the off-chain storage nodes. We also observe that the on-chain storage size of SlimChain remains constant regardless of the smart contracts.

Figures 10 and 11 report the the peak throughput and respective transaction commit latency for different smart contracts under the default system parameters. As shown in Figs. 10a and 11a, SlimChain increases the throughput by 1.6X ~ 11.3X and 2.6X ~ 15.6X against Classic for all smart contracts. In particular, SlimChain achieves the best performance improvement for CPUHeavy thanks to the outsourcing computation and the concurrent execution of smart contracts. Compared with Fabric#, SlimChain increases the throughput by 1.4X ~ 2.6X. SlimChain shares similar throughput with Stateful, although some notable improvement can be observed in KV and SB in the permissioned setting.

For latency, Figs. 10b and 11b show that SlimChain and Stateful achieve the lowest latency in comparison with Classic and Fabric# across almost all smart contracts. Note that the storage node execu-

Table 3: Success Rate vs. Concurrency Control (Permissioned)

Smart Contract	KV	SB
OCC Success Rate	90.49%	99.62%
SSI Success Rate	95.34%	99.75%

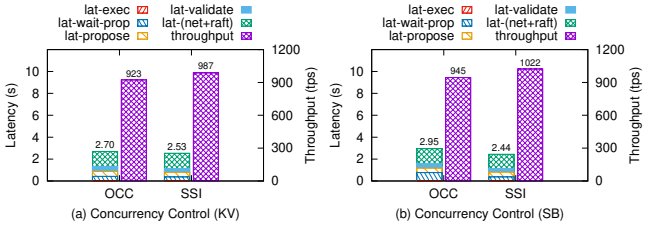

Figure 12: Performance vs. Concurrency Control (Permissioned)

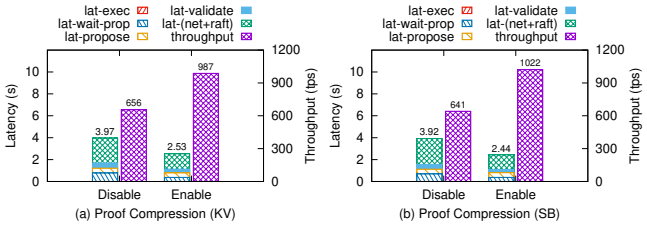

Figure 13: Performance vs. Proof Compression (Permissioned)

tion time in SlimChain and Stateful is too small to be visible in the figure. Moreover, it can be observed from the latency breakdown that the biggest contributor to the latency is network communication and Raft consensus (net+raft) in the permissioned setting. In comparison, the transactions spend most of time in the queue (wait-prop) in the permissionless setting.

Overall, we can observe that SlimChain achieves the lowest storage requirement with no sacrifice in terms of performance.

6.2.2 Impact of System Parameters. In this section, we evaluate the performance of SlimChain under different system parameters to test their impact. We only test the two macro benchmarks in the permissioned setting here, i.e., KVStore (KV) and SmallBank (SB).

First, we evaluate the influence of two concurrency control methods, namely OCC and SSI. Their commit success rates are presented in Table 3, in which SSI yields a higher success rate than OCC for both smart contracts. The results are aligned with the analysis in Section 4.2. As shown in Fig. 12, these higher success rates contribute to an increase in throughput by 7% ~ 8% and a decrease in latency by 6% ~ 21%.

Next, we evaluate how the proposed proof compression optimization discussed in Section 4.3.1 impacts the system performance. As shown in Fig. 13, SlimChain yields a better performance in throughput by 1.5X ~ 1.6X and a reduction in latency by 36% ~ 38% when the proof compression optimization is used. This is expected as the network often contributes to the bottleneck of distributed system and the proof compression helps reduce transmission overheads during block synchronization.

6.2.3 Sharding Performance. Finally, we investigate the performance of SlimChain with sharding. We evenly partition the smart contract world states based on smart contract addresses. Figure 14a shows KVStore's peak throughput and latency with storage nodes being partitioned to different numbers of shards. Here, the number of shards being equal to 1 means that sharding is disabled. Since our

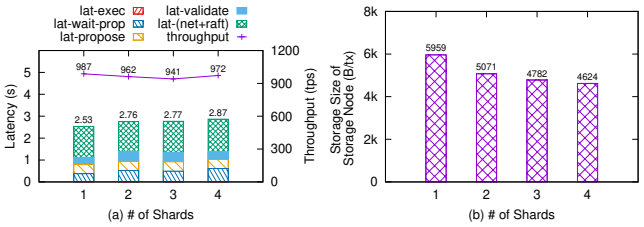

Figure 14: Performance vs. # of Shards (KV, Permissioned)

proposed sharding technique is completely transparent to the consensus nodes, we observe only very small changes in transaction commit latency and peak throughput. On the other hand, Fig. 14b shows that with more shards, more storage spaces are saved among the storage nodes. Because all of the storage nodes need to store the upper-level Merkle trie, this space saving is however not linear in terms of the number of shards.

7 CONCLUSION

In this paper, we have designed a novel *stateless* blockchain system, SlimChain, that scales transactions through off-chain storage and parallel processing. Specifically, the ledger states and transaction executions are moved to off-chain storage nodes to improve system scalability. To support *stateless* transaction commitment, we designed new off-chain transaction execution, on-chain transaction validation, and node synchronization schemes, along with a novel partial Merkle trie structure. To further improve system performance, we proposed optimizations to reduce network transmissions and a new sharding technique. Extensive experiments show that the proposed SlimChain system reduces the on-chain storage requirements by 97% ~ 99% and improves the peak throughput by 1.4X ~ 15.6X over the existing systems.

There are many interesting research problems that deserve further investigation for *stateless* blockchain, e.g., how to further reduce on-chain states by utilizing more advanced data structures; how to decrease the operating costs for storage nodes; and how to support data provenance under the new stateless design.

ACKNOWLEDGMENTS

This work was supported by Research Grants Council of Hong Kong (Project Nos. 12201520, 12200819) and NSERC Discovery Grants. Jianliang Xu is the corresponding author.

REFERENCES

[1] C. Xu, C. Zhang, and J. Xu. 2019. vChain: enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management*, 141–158.

[2] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi. 2021. Authenticated keyword search in scalable hybrid-storage blockchains. In *Proceedings of the 37th IEEE International Conference on Data Engineering*, 996–1007.

[3] S. Nakamoto. 2008. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[4] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman. 2016. Medrec: Using blockchain for medical data access and permission management. In *2nd International Conference on Open and Big Data*, 25–30.

[5] S. A. Abeyratne and R. P. Monfared. 2016. Blockchain ready manufacturing supply chain using distributed ledger. *International Journal of Research in Engineering and Technology*, 5, 9, 1–10.

[6] G. Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

[7] M. Zamani, M. Movahedi, and M. Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 931–948.

[8] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment*, 12, 11, 1597–1609.

[9] A. Chepurnoy, C. Papamanthou, and Y. Zhang. 2018. EDRAX: A cryptocurrency with stateless transaction validation. Cryptology ePrint Archive, Report 2018/968. (2018).

[10] D. Boneh, B. Bünz, and B. Fisch. 2019. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, 561–586.

[11] A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich. 2020. Aggregatable subvector commitments for stateless cryptocurrencies. Cryptology ePrint Archive, Report 2020/527. (2020).

[12] E. Androulaki et al. 2018. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*.

[13] C. Fan, S. Ghaemi, H. Khazaei, and P. Musilek. 2020. Performance evaluation of blockchain systems: a systematic survey. *IEEE Access*, 8, 126927–126950.

[14] R. C. Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, 218–238.

[15] B. Parno, J. Howell, C. Gentry, and M. Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, 238–252.

[16] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *23rd USENIX Security Symposium*, 781–796.

[17] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. 2015. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, 253–270.

[18] J.-E. Ekberg, K. Kostiainen, and N. Asokan. 2013. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, 1497–1498.

[19] V. Costan and S. Devadas. 2016. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086. (2016).

[20] O. P. Project. 2020. The oasis blockchain platform. https://oasisprotocol.org/papers.

[21] J. Teutsch and C. Reitwießner. 2019. A scalable verification solution for blockchains. *arXiv*.

[22] F. Saleh. 2020. Blockchain without waste: Proof-of-stake. *The Review of Financial Studies*, 34, 3, 1156–1190.

[23] M. Castro and B. Liskov. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20, 4, 398–461.

[24] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. 2016. Corda: An introduction. https://docs.corda.net/en/pdf/corda-introductory-whitepaper.pdf.

[25] J. P. M. Chase. [n. d.] Quorum: A permissioned implementation of ethereum. https://github.com/jpmorganchase/quorum.

[26] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, 105–122.

[27] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, 123–140.

[28] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran. 2019. Blockchain meets database: Design and implementation of a blockchain relational database. *Proceedings of the VLDB Endowment*, 12, 11, 1539–1552.

[29] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi. 2020. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 543–557.

[30] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. 2019. FastFabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency*, 455–463.

[31] C. Gorenflo, L. Golab, and S. Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency*, 1–9.

[32] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. 2017. Chainspace: A sharded smart contracts platform. *arXiv*.

[33] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. 2020. ResilientDB: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13, 6, 868–883.

[34] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang. 2020. Pointproofs: Aggregating proofs for multiple vector commitments. Cryptology ePrint Archive, Report 2020/419. (2020).

[35] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1–10.

[36] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*, 67–78.

[37] M. J. Cahill, U. Röhm, and A. D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems*, 20–42.

Algorithm 7: Update Partial Merkle Trie $\mathcal{T}_w$ (Full)

Input: write set Merkle proof π_{write}, write set $\{w\}_{tx}$.

1 $Q \leftarrow$ `init_queue()`;
2 Q.`enqueue`($\langle \pi_{write}$.`root()`, $\mathcal{T}_w$.`root()`$\rangle$);
3 **while** ***not*** Q.`is_empty()` **do**
4 $\langle n_\pi, n_\mathcal{T} \rangle \leftarrow Q$.`dequeue()`;
5 **switch** $\langle n_\pi.type, n_\mathcal{T}.type \rangle$ **do**
6 **case** $\langle$*Extension Node*, *Extension Node*$\rangle$ **do**
7 **if** $n_\pi.key = n_\mathcal{T}.key$ **then**
8 **if** $n_\mathcal{T}.child$ *is suppressed* **then**
9 Update $n_\pi.child$ using values from $\{w\}_{tx}$;
10 Replace $n_\mathcal{T}.child$ with $n_\pi.child$;
11 **else** Q.`enqueue`($\langle n_\pi.child, n_\mathcal{T}.child \rangle$);
12 **else if** $n_\mathcal{T}.key \subsetneq n_\pi.key$ **then**
13 Split n_π at $|n_\mathcal{T}.key|$;
14 Q.`enqueue`($\langle n_\pi.child, n_\mathcal{T}.child \rangle$);
15 **else** Update $n_\mathcal{T}$ using values from $\{w\}_{tx}$;
16 **case** $\langle$*Leaf Node*, *Extension Node*$\rangle$ **do**
17 **if** $n_\mathcal{T}.key \subsetneq n_\pi.key$ **then**
18 Split n_π at $|n_\mathcal{T}.key|$;
19 Q.`enqueue`($\langle n_\pi.child, n_\mathcal{T}.child \rangle$);
20 **else** Update $n_\mathcal{T}$ using values from $\{w\}_{tx}$;
21 **case** $\langle$*Leaf Node or Extension Node*, *Branch Node*$\rangle$ **do**
22 $idx \leftarrow n_\pi.key[0]$; Split n_π at 1;
23 Q.`enqueue`($\langle n_\pi.child, n_\mathcal{T}.child[idx] \rangle$);
24 **case** $\langle$*Branch Node*, *Branch Node*$\rangle$ **do**
25 **for** $\langle n_1, i \rangle \in n_\pi$.`child_nodes_with_index()` **do**
26 $n_2 \leftarrow n_\mathcal{T}.child[i]$;
27 **if** n_2 *is suppressed* **then**
28 Update n_1 using values from $\{w\}_{tx}$;
29 Replace n_2 with n_1;
30 **else** Q.`enqueue`($\langle n_1, n_2 \rangle$);
31 **else** Update $n_\mathcal{T}$ using values from $\{w\}_{tx}$;

[38] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, 1085–1100.

A PARTIAL MERKLE TRIE UPDATE ALGORITHM

Algorithm 7 shows the full algorithm to update the partial Merkle trie taking account of different node types in the Merkle Patricia Trie. The overall algorithm works in a similar fashion to Algorithm 3 by finding the proper places through tree traversal to insert the missing subtrees and apply the written values. However, due to the nature of the concurrent transaction execution and asynchronous network, the write set Merkle proof π_{write} will be likely outdated and thus will have a different tree structure compared with the partial Merkle trie $\mathcal{T}_w$. Therefore, care is needed to handle newly writes that are only present in the partial trie. Two of the special treatments are worth highlighting. In certain cases, the extension node or the leaf node in $\mathcal{T}_w$ may be split into a branch node and the remaining tree node due to the writes from other concurrent transactions. Such a situation entails us performing the same node splitting for the corresponding nodes in π_{write} (Lines 13, 18 and 22). On the other hand, the tree node in π_{write} and that in $\mathcal{T}_w$ may become drastically different. This happens when the current values corresponding to the write addresses are zero, so they are

not present in the Merkle Patricia Trie. At the same time, the old Merkle paths in π_{write} have since been overwritten to other values resulting to drastically different tree structure. Since the conflict check (Algorithms 5 and 6) ensures that there is no write-write conflict, we can update $\mathcal{T}_w$ directly without using the outdated subtrees from the write set proof π_{write} in such a situation (Lines 15, 20 and 31). It is also worth noting that the same algorithm can be used to compute the compressed proof for the block observers, as discussed in Section 4.3.

B PRACTICAL ISSUES

In this section, we briefly discuss certain practical issues that arise when applying SlimChain to real-life blockchain applications.

Transaction Fees. In permissionless blockchains, both the block proposers and storage nodes may want to collect a certain amount of transaction fees. To facilitate this, a block proposer can always store the state of its account balance locally, which incurs only a slight storage overhead. With this state, the block proposer can generate a necessary Merkle proof to update the balance to collect the fee when creating the new block. For the storage nodes, the transaction fees are simply processed as part of normal transaction executions. However, to boost system efficiency, we should avoid unnecessary read-write dependencies between different transaction executions. This means that instead of collecting the transaction fees into the same account, the storage nodes can use multiple accounts for this purpose. In this way, different transactions can be handled without causing conflicts.

Handling Forks. For some consensus protocols relying on eventual consistency, such as Proof-of-Work, there could occur instances of forks. In such a scenario, the blockchain nodes are required to rewind the current chain states to an old-time point and apply the updates from the current longest chain. For SlimChain, this entails rewinding the temporary states discussed in Section 4.2. This can be done in two ways. If the rewind goes back far enough in time, the consensus nodes can simply discard the current temporary states and retrieve the old temporary states from the storage nodes. Alternatively, if the rewind concerns only a few recent blocks, a better strategy would be keeping a copy of these temporary states and using the corresponding old copy directly. This would not increase storage overheads too much as only old copies for recent blocks are stored. Moreover, one can use techniques such as Copy-on-Write and persistent data structures to further reduce the storage size.

C ADDITIONAL EXPERIMENTS

C.1 Storage Size and Success Rate

In the section, we present more details in terms of the storage size and success rate for SlimChain and baselines under the default system parameters.

Table 4 shows the average per transaction storage size for both consensus and storage nodes. It shows that SlimChain drastically reduces the on-chain storage requirement up to 99%. On the other hand, the off-chain storage requirement for the storage nodes is similar to those of the consensus nodes in the baseline systems. This is expected as we shift most of on-chain storage to dedicated storage nodes in SlimChain.

Due to concurrent executions in SlimChain and Stateful, transactions may be aborted due to serialization conflicts. Table 5 shows

Table 4: Storage Size (B/tx) vs. Smart Contract

Smart Contract	DN	CPU	IO	KV	SB
Permissioned					
Classic Consensus Node	5,051	5,068	15,153	10,912	9,184
Stateful Consensus Node	3,844	3,838	8,727	5,955	4,761
Stateful Storage Node	3,844	3,838	8,727	5,955	4,761
Fabric# Consensus Node	3,137	3,136	3,472	3,365	3,253
Fabric# Endorsement Node	3,137	3,136	3,472	3,365	3,253
SlimChain Consensus Node	65	65	65	65	65
SlimChain Storage Node	3,844	3,851	8,718	5,959	4,750
Permissionless					
Classic Consensus Node	5,046	5,057	14,999	10,877	9,155
Stateful Consensus Node	3,329	3,325	6,904	5,012	4,166
Stateful Storage Node	3,329	3,325	6,904	5,012	4,166
SlimChain Consensus Node	63	63	63	63	63
SlimChain Storage Node	3,324	3,344	7,057	5,074	4,137

Table 5: Success Rate vs. Smart Contract

Smart Contract	DN	CPU	IO	KV	SB
Permissioned					
Fabric#	100%	100%	99.91%	94.48%	99.42%
Stateful	100%	100%	99.98%	95.54%	99.79%
SlimChain	100%	100%	99.98%	95.34%	99.75%
Permissionless					
Stateful	100%	100%	99.91%	91.92%	99.29%
SlimChain	100%	100%	99.89%	92.59%	99.22%

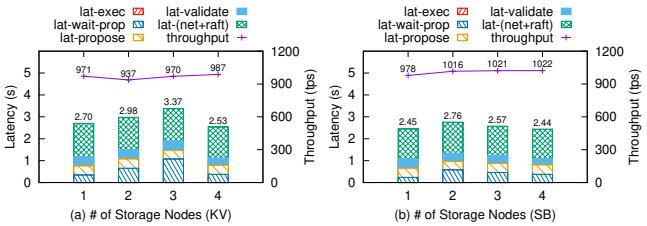

Figure 15: Performance vs. # of Storage Nodes (Permissioned)

the transaction commit success rate among different smart contracts when the SSI concurrency control method is used. There is no conflict for the DoNothing and CPUHeavy workloads as they only involve minimal IO operations for updating the nonce of the transaction sender accounts. Overall, we observed high success rates across all smart contracts.

C.2 Impact of Network Topology

In this section, we evaluate the performance of SlimChain under different network topologies. Similar to Section 6.2.2, we only test the two macro benchmarks in the permissioned setting here.

Figure 15 shows how the system performs with varying numbers of storage nodes. Although more storage nodes mean more computation capacities, the increasing number of storage nodes also contributes to a higher cost in the network communication. As such, we observed that with the increasing storage node number, the throughput and latency do not change evidently. Nevertheless, it is still worth noting that multiple storage nodes are essential in real-life decentralized applications to ensure blockchain liveness.

In addition to the varying number of storage nodes, we also investigate the system performance under different blockchain network sizes. As shown in Fig. 16, the number of consensus nodes

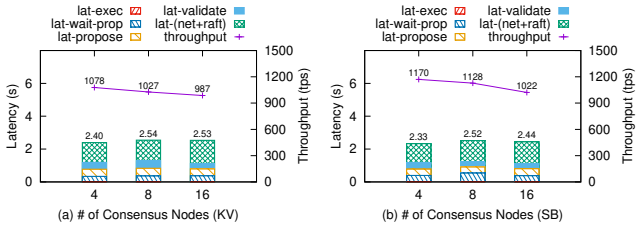

Figure 16: Performance vs. # of Consensus Nodes (Permissioned)

plays an important role in both the peak throughput and latency. This is expected as the Raft consensus protocol introduces a linear complexity in network communication with respect to the network size. As consensus protocol can be one of the bottlenecks in a blockchain system [13], we observed that more consensus nodes lead to a reduction in overall system performance.