$\mathcal{S}$-Store: A Scalable Data Store towards Permissioned Blockchain Sharding

Xiaodong Qi
School of Data Science and Engineering, East China Normal University, Shanghai, China
{xdqi}@stu.ecnu.edu.cn

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796800

***Abstract*—Sharding technique, which divides the whole network into multiple disjoint groups or committees, has been recognized as a revolutionary solution to enhance the scalability of blockchains. For account-based model, state data are partitioned over all committees and organized as *Merkle trees* to ensure data consistency and immutability. However, existing techniques on Merkle tree-based state storage fail to scale out due to a large amount of network and compute overheads incurred by data migration and Merkle tree reconstruction, respectively. In this paper, we propose $\mathcal{S}$-Store, a scalable data storage technique towards permissioned blockchain sharding based on *Aggregate Merkle B+ tree* (AMB-tree). $\mathcal{S}$-Store utilizes consistent hashing to reduce data migration among committees and uses split and merge on AMB-tree to decrease Merkle tree reconstruction overheads. $\mathcal{S}$-Store also employs a novel committee addition protocol that guarantees the system service availability during data migration. Extensive experiments show that $\mathcal{S}$-Sotre outperforms existing techniques by one order of magnitude in terms of transaction execution, data transmission, and committee addition.**

***Index Terms*—Blockchain, Merkle tree, Sharding**

I. Introduction

Blockchain systems offer data transparency, integrity and immutability in a decentralized and potentially hostile environment at cost of lower scalability [9]. Many efforts have been taken to address the scalability problem of blockchains. Some solutions focus on improving the performance of key components of blockchains, including consensus protocol [10], [25] or smart contract execution [4], [19]. However, the benefits of these improvements are limited. Instead, sharding, a well-studied and proven technique to scale out databases, has been adapted to blockchain to increase the parallelism [13], [15], [26]. Generally, sharding divides the whole network into small committees (also called *shards*), each of which preserves a part of the global state data of system. Then every committee deploys a consensus protocol to generate blocks and executes transactions independently. Commonly, the overall throughput of a sharded blockchain can be improved by a factor of m when m committees exist.

The sharded blockchain systems can be divided into two categories based on the data model used: **UTXO** model and **account-based** model. Examples of sharded blockchains utilizing UXTO model include Elastico [15], OmniLedger [13] and Monoxide [23]. However, these systems are limited to cryptocurrency applications in an open setting. Instead, some systems [8], [22] start to support more complicated applications in permissioned setting based on a more generalized account-based model. Specifically, states manipulated by smart contracts are termed as a collection of $\langle k, v\rangle$ pairs, e.g. account address and the corresponding value in Ethereum [6]. Those transactions that invoke smart contracts are routed to the appropriate committees for execution. Usually each committee organizes and maintains state data in a Merkle tree fashion, i.e. *Merkle Patricia tree* (MPT) [24], *Merkle Bucket tree* (MBT) [2] and *Merkle AVL tree* [3]. Within a committee, the usage of Merkle encompasses three purposes: (1) index of states; (2) digest calculation of states; (3) proof called the *witness* for each read.

Different from public blockchains, in a permissioned setting, we can support large-scale applications of sharded blockchain system by leveraging more committees, such as the Blokchain-as-a-Service(BaaS) platform [1]. For example, when the transactions from clients cannot be handled in time, each organization of a permissioned blockchain can devote more physical nodes to form new committees to tackle them. Although Merkle tree-based solutions can easily manage data in static sharded blockchains (i.e., no addition or removal of committee), they cannot scale well as the number of committees changes. As committees join or exit, all state data need to be redistributed across committees, incurring massive migration of data and reconstruction of Merkle trees. To transmit state data to others, each committee needs to deploy a number of random reads on underlying key-value store (KVS) implemented on LSM-tree [20], e.g. LevelDB. However, since LSM-tree optimizes write operations by sacrificing read performance, the latency for data migration is further increased. When this process is completed, each committee also needs to calculate a lot of hashes to reconstruct a completely new Merkle tree. In this study, we focus on a scalable storage technique to solve the following issues.

Data migration issue. The states are partitioned over committees evenly, while fewer data are migrated when network changes. A common method [15], [26] is to divide the key space into m disjoint *zones* by mapping every state $\langle k, v\rangle$ to the committee numbered with $H(k) \bmod m$, where $H(\cdot)$ is a hash function. However, this approach incurs massive network transmissions for the addition of committee due to the redistribution of states, because the equation $H(k) \bmod m = H(k) \bmod (m+1)$ is not held for most situations.

Merkle tree reconstruction issue. With the addition of a committee, all states need to be redistributed, thus each committee scans the local Merkle tree to transmit states to others. Then, each committee rebuilds a fresh Merkle tree based on states received from others. However, this process is inefficient for two reasons. First, the cost of scanning is significant due to read amplification of LSM-tree. Second, the structure of Merkle tree in each committee is destroyed completely and a large number of computational resources are devoted to reconstruct a new valid Merkle tree.

As a result of above two issues, the service of each committee will be disrupted for committee addition, since there is no complete Merkle tree locally. Sequentially, no transaction can be processed by the system and all data become inaccessible. To address above issues, we propose $\mathcal{S}$-Store, a scalable distributed data storage technique, to manage state data in sharded blockchain systems. The major contributions of this paper are summarized below.

- The proposed $\mathcal{S}$-Store applies consistent hashing to reduce the amount of state data to be transferred and achieves fast reconstruction of Merkle tree based on an novel structure *AMB-tree*.
- We design a committee addition (or removal) protocol to guarantee fluent services of $\mathcal{S}$-Store. Therefore, committees can keep dealing with transactions and provide authenticated queries on states within this period.
- The experimental results show that our proposed $\mathcal{S}$-Store technology can reduce latency for data migration by 70% and enable great availablility in a sharded blockchain system with 4 committees and 16 nodes.

The rest of this paper is organized as follows. Section II reviews related work and background on sharding and blockchain. Section IV describes our design of AMB-tree, and Section V describes the operations on AMB-tree. Section VI proposes a committee addition protocol based on AMB-tree. The experimental evaluations are reported in Section VII. Section VIII concludes this paper.

II. Related Work and Background

A. Sharding in blockchain

The poor scalability constrains the development and application of blockchain. Recently, sharding is applied in blockchain to divide the network into multiple committees to process transactions in parallel. The early shared blockchain merely supports network and transaction sharding based on the UTXO model, such as Elastico [15], OmniLedger [13] and Monoxide [23]. In Elastico, every block needs to be broadcast to all nodes for storing (i.e. cannot support the state sharding). The later systems start to consider state sharding in blockchain, such as Chainspace [22] and AHL [8], which generalizes the sharding technique to smart contract application beyond cryptocurrency. Particularly, [8] proposes a *distributed transaction protocol* towards smart contracts based on two-phase commit (2PC) and two-phase locking (2PL) protocols to tackle cross-committee transactions, which manipulate states in multiple committees. Inside each committee, Byzantine failure tolerance consensus protocol is deployed to achieve agreement for blocks among all nodes, e.g., PoW [18] or PBFT [7].

PBFT protocol. Practical Byzantine Fault Tolerance (PBFT) [7], the most well-known BFT protocol, can tolerate f faulty nodes out of n nodes in three phases, such that $n \geq 3f+1$. PBFT is deterministic without invoking any fork, thus existing permissioned blockchain commonly adopts PBFT for agreement of blocks [2], [3].

B. State organization

Various blockchains employ different data models to deal with state data. Bitcoin adopts UTXO model, while typical Blockchain 2.0 systems which support smart contract, such as Ethereum and Hyperledger, utilize the account-based model where the state of each account may be modified in the future. In these systems, the states are organized as a Merkle tree or its variants. These structures are divided into two categories: i) just for digest calculation like MBT [2] , which is not an index structure for states; ii) for state retrieval like MPT [24], which supports witness for any state.

Merkle Bucket Tree (MBT). MBT consists of two parts: a hash table and a Merkle tree. The hash table consists of a series of buckets, each of which contains a number of sorted states. The hash of a bucket is a leaf of the upper Merkle tree, and the root hash of Merkle tree is the digest of all states. Since the MBT is a fixed-size tree, its height keeps constant and recalculation of dirty nodes in the traversal from root to the dirty bucket can give a new digest of states. However, MBT cannot provide a witness for a separate state.

Merkle Patricia Tree (MPT). MPT is a mixture of Patricia trie [17] and Merkle tree. Apart from digest calculation, it is an index of states and provides witness for state read. Differently, each pointer in a node is replaced by a hash pointer, which is the cryptographic digest of child's content. Besides, MPT stores all versions for a state by building a snapshot of global state at every block. However, MPT is not a balanced structure, which affects the performance [27]. To avoid this, Tendermint [3] persists states via a balanced *AVL tree* to remove the structural skew of Merkle tree.

The aforementioned structures are all built based on a KVS with significant read amplification. [14] proposes a disk-oriented *Merkle B+ tree* (MB-tree), which reduces the amplification of LSM-tree. However, MB-tree cannot support multi-version of data.

III. Overview of $\mathcal{S}$-Store

In this study, we focus on the state store towards sharded blockchains upon account-based model, where each account (state) has a global unique address as the key and its content is the value. We overview the proposed $\mathcal{S}$-Store from three aspects: system architecture, adversary model, and interfaces.

System architecture. $\mathcal{S}$-Store is a distributed store engine towards sharding, as depicted in Figure 1. There are two different types of nodes:(1) validators, which participate in consensus, store states and execute transactions;(2) light clients, which store no state locally and propose transactions to update the states of blockchains. Each committee consists of n validators and runs PBFT protocol to achieve agreement for blocks in parallel. All states are partitioned and distributed across m committees, inside which the partial states are managed by an AMB-tree as elaborated in Section IV. The clients call smart contracts to update states by sending transactions to blockchain. As the business grows in size, new

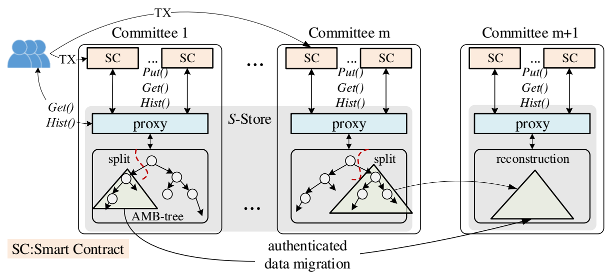


Fig. 1. Architecture of $\mathcal{S}$-Store.

committees need to be added to the system. Sequentially, states are redistributed to rebalance workloads across committees. Since $\mathcal{S}$-Store is designed against permissioned blockchains, the addition of new committees must be granted by the trusted administrator like the CA in Hyperledger Fabric [2]. Each client needs to cache the committee-related information to determine to which committee a certain state belongs to, such as public key, ID, address space, etc.

Adversary model. Same as the assumption of PBFT, within a committee, at most f out of n validators can be malicious, such that $n \geq 3f + 1$. These f faulty nodes can misbehave in arbitrary ways. As stated above, since clients do not store any data locally, they need to request the corresponding states from validators of different committees when necessary. In $\mathcal{S}$-Store, there is no trust between the client and a single validator, so the client needs to verify that the data received from the validator is correct and complete. For this purpose, the validator needs to include a witness about the state when sending it to the client. In $\mathcal{S}$-Store uses the Merkle proof [16] for authentication between a validator and client.

APIs of $\mathcal{S}$-Store. $\mathcal{S}$-Store affords application (smart contract) or clients three core APIs, including: $Put(k, v, blkNo)$, $Get(k, [blkNo])$ and $Hist(k, s, t)$. The $Put(\cdot)$ is used to write data v with key k into $\mathcal{S}$-Store with tag block number $blkNo$ as version, while $Get(\cdot)$ returns the value of state with k at block $blkNo$. By default, the latest block is used. It is important to note that the numbers of blocks are strictly increased within each committee, and no two committees will produce blocks with the same block number, which is detailed in Section IV-C. $Hist(\cdot)$ returns a list of tuples $\langle v, blkNo\rangle$ for k, which enables smart contract or client to trace historical versions of a state between block number s and t.

IV. Design of $\mathcal{S}$-Store

As a distributed store for states towards blockchain sharding, $\mathcal{S}$-Store partitions the state data over all committees via consistent hashing. Within a committee, $\mathcal{S}$-Store organizes states as an *Aggregate Merkle B+ tree* (AMB-tree) for digest calculation and witness generation. To support data provenance, AMB-tree uses a skip list to store multiple versions of the same state.

A. Consistent space division

To dispatch all states to all committees evenly, existing systems adopt a modulo manner, where state $\langle k, v\rangle$ is mapped to committee with sequence number $(H(k) \bmod m)$. However, a great number of states are remapped to other committees when m varies for the addition or removal of committees. Furthermore, each committee has to reorganize local states to construct a valid Merkle tree.

$\mathcal{S}$-Store applies the consistent hashing [12] to divide address space. Consistent hashing maps each state to a point on a ring, and the system maps each available committee to a point on the same ring. To determine which committee a state should be placed in, $\mathcal{S}$-Store finds the location of that state's hash key $hk = H(k)$ on the ring; then walks clockwise around the ring until falling into the first point mapped by a committee. Theoretically, each committee owns a range of the hashring, and any state coming in all of this range will be stored by the same committee. Therefore, the address space of this ring is divided into multiple disjoint ranges. If one of these committees is removed, the range of next committee widens and all states coming in this range goes to it. By this way, the rest of the hashring still remains unaffected. Oppositely, if a new committee is hashed to this ring, it cuts part of range from adjacent committee. Thanks to consistent hashing, only a small portion of all states are affected by given re-division of the ring. As advised by [12], $\mathcal{S}$-Store maps a committee to c different virtual points on a ring, (e.g. 32 or larger) to lower the skew of workload on each committee. Each range owned by a committee is called a *zone* in this study. Once a new committee joins, it decides its own zones, and then synchronizes state data in these zones from others, which is detailed in Section VI.

B. Structure of AMB-tree

Within a committee, each validator manages states of c different zones. Apart from digest calculation and witness support, another key task of Merkle tree is to index states for read/write operation. However, existing structures like MPT cannot gain a satisfied performance on this task, since they integrate KVS implemented on LSM-tree [20], a write-oriented structure (e.g., LevelDB and RocksDB). Particularly, each index node of MPT is stored in LevelDB, where the key is the hash of the node's content and the value is its content. Sequentially, to read a state $\langle k, v\rangle$ in MPT, we need to involves multiple *get* operations on LevelDB to obtain all nodes in the traversal from root to the leaf, which is a second read amplification beyond LSM-tree. Therefore, the read complexity of MPT for state with N different states is $O(\log^3 N)$, since $O(\log N)$ nodes of MPT are visited and the cost of read a node from LevelDB is $O(\log^2 N)$ [11]. Furthermore, the compaction of LSM-tree is a both CPU and I/O heavy operation, which leads to a cliff-like decline on performance [27].

We propose an *Aggregate Merkle B+ tree* (AMB-tree), as depicted in Figure 2, to manage states of each committee. The *Merkle B+ tree* (MB-tree) is used to manage states inside a zone and then c different MB-trees are aggregated by a binary Merkle tree to calculate digest of all states in a committee. Meanwhile, we enable MB-tree support multi-version provenance and witness, which are required in blockchain

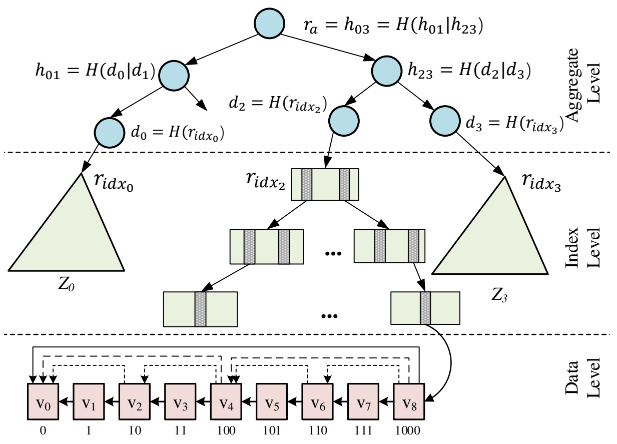


Fig. 2. An AMB-tree with 4 zones.

system [24]. AMB-tree consists of three levels: *aggregate level* constructed by a binary Merkle tree, *index level* formed by an MB-tree and *data level* comprised by an authenticated list. $\mathcal{S}$-Store stores all versions of a state together as a list to optimize $Hist(\cdot)$, instead of snapshotting AMB-tree at every block like MPT. $\mathcal{S}$-Store uses hash key hk of state in AMB-tree for read/write operations.

Aggregate level. It is a binary Merkle tree, where each leaf node contains a hash pointer referring to the root r_{idx} of an MB-tree in index level and each inner node has one or two hash pointer(s) referring to its child(ren). All leaf nodes are sorted according to the ranges of their corresponding zones in clockwise manner, and the range covers '0' on hashring is the smallest one. The root r_a of this Merkle tree, the digest of all states in a committee, is included in block header to check the consistency among all validators. When the underlying MB-tree for a zone changes, the nodes in the traversal from that leaf node to root r_a are recalculated to obtain a new digest of all states. For example, if there is a update on zone Z_0 in Figure 2, the nodes d_0, h_{01} and r_a are recalculated. In addition, the data of aggregate level are maintained in memory entirely to accelerate data access.

Index level. An MB-tree [14] works like a B+ tree, a disk-oriented structure, and consists of ordinary B+ tree nodes that are extended with one hash pointer associated with every pointer item. MB-tree offers excellent I/O performance by integrating the Merkle directly, avoiding second amplification. Specially, each version list for a state in the data level is considered as the leaf of MB-tree. When a state is updated, a new version is appended to the version list, and all nodes in traversal from the root r_a to this list are rehashed to obtain a new digest. As a variant of Merkle tree, MB-tree promises the data integrity and immutability of data via a small proof (witness). Besides, MB-tree enables fast scan operation for data migration due to the bulk reading of B+ tree.

Data level. The multiple versions of a state make up a list in data level to support rapid trace on historical data. Each node in the list contains a hash pointer linked to previous node for data authentication. To response a $Hist(\cdot)$ query, $\mathcal{S}$-Store travels along this list and loads all relevant versions in a batch. $\mathcal{S}$-Store makes the latest version referred by a leaf node of MB-tree, because it is visited more frequently [21]. Different from MPT, $\mathcal{S}$-Store does not make a snapshot of entire Merkle tree at each block for space efficiency. Specifically, within the execution of transactions in block, if a node (both index node and leaf node) of MPT is updated, MPT makes a new copy of it and update aside. Therefore, we can obtain any version of a state from the root hash included in corresponding block header. However, this mechanism will wast huge space to save all snapshot. Instead, $\mathcal{S}$-Store merely appends the new version to the list of a state without copying nodes.

C. Authenticated append-only skip list

To enable fast version tracing queries in data level, $\mathcal{S}$-Store implements an index, namely *authenticated append-only skip list* (AASL), on the version list, which is exampled in Figure 2 with 8 versions. Different from a normal skip list, AASL provides witness as tamper evidence for any version of a state, and is append-only since the key of the appended version is the block number that increases strictly in our case. Specifically, the insertion of a new version does not modify any historical versions.

Let $V_{hk} = \langle v_1, v_2, \cdots \rangle$ be the sequence of block numbers ($blkNo$) for state with hash key $hk = (k)$, in which $v_i < v_j$ for all $i < j$. In the absence of ambiguity, we also use the version number v_i to refer to the corresponding version node in AASL. There is a left sentinel node v_0 and other nodes maintain some forward links to previous nodes. The links between nodes are organized level by level. The 0-$level$ contains all versions, the l-$level$ contains every 2^l-th version of 0-$level$, and so on. A node maintains a link containing a hash pointer, a physical pointer, and an index key, to predecessor in each level, e.g., v_4 points to v_3, v_2 and v_0 in 0-$level$, 1-$level$ and 2-$level$ respectively. It can be proved that the number of forward link(s) maintained by a node v_i equals to number of successive '0' at tail of i's binary representation plus 1. The preceding version set PV_i, a set of node(s) linked by node v_i, is calculated as follows.

$$PV_i = \{v_{i-2^j} | 0 \leq j \leq lowbit(i)\} \tag{1}$$

In Equation (1), $lowbit(i)$ returns number of successive 0 at the tail of i. For instance, preceding version set PV_8 for v_8 is $\{v_{8-2^0} = v_7, v_{8-2^1} = v_6, v_{8-2^2} = v_4, v_{8-2^3} = v_0\}$, since its binary representation is "0x1000" tailed with three '0'. To access a historical version from the latest version, the query complexity is a logarithm of list's length. Furthermore, the insertion of the new version just appends a node to AASL without any modification on current available data.

The AASL for a state requires the block number is monotonically increasing to enable fast retrieval of historical versions, which is met within a committee. Yet, if a committee is added and a state (AASL) moves to it, the block number restarts from 1, which results in duplicated block number. Furthermore, to insert a new version, it may alter multiple nodes to make all nodes sorted. To address this issue, $\mathcal{S}$-Store uses the concatenation of committee ID and block height

$blkNo = \langle cid|bh \rangle$ as the block number, where committee ID cid is the identifier of a committee and each block produced by a committee is tagged with a block height bh numbered from 1. Specifically, a committee's ID increases with the addition or removal of a committee. If the maximal ID of existing committees is m_cid, the ID of new committee C' is $m\_cid + 1$. By this method, the $blkNo = \langle m\_cid + 1|bh \rangle$ of blocks produced by C' is always greater than any block of existing committees. Therefore, when the AASL of a state moves to C', the block number of versions inserted to it still keeps monotonic. Similarly, when a committee is deleted from blockchain system, its states are merged with other committees. If a committee needs to synchronize states (AASLs) from the committee to be removed, its ID is set to $m\_cid + 1$ again.

V. Operations on AMB-tree

A. Read with witness

Since there is no trust between validators and clients, for each verified read, the client obtains a proof (witness) from the committee of $\mathcal{S}$-Store. The client sends the read request $Get(k, [blkNo])$ to a validator in the committee, which stores the state with hash key $hk = H(k)$. To answer a read $Get(k, [blkNo])$, $\mathcal{S}$-Store seeks it by hash key $hk = h(k)$ and builds a witness $\mathcal{O}(hk)$ based on AMB-tree. The witness $\mathcal{O}(hk)$ consists of three parts: $\mathcal{O}_a(hk)$, $\mathcal{O}_{idx}(hk)$ and $\mathcal{O}_d(hk)$. $\mathcal{S}$-Store builds $\mathcal{O}_a(hk)$ by including in $\mathcal{O}_a(hk)$ the hashes of siblings of nodes in traversal from root r_a to the leaf node referring to the zone Z_i, which covers hash key hk, in aggregate level. In index level, $\mathcal{S}$-Store initiates a top-down traversal like B+ tree to find the leaf (the AASL V_{hk}). The hash pointers contained in each node visited by traversal are included in $\mathcal{O}_{idx}(hk)$.

In data level, the proof $\mathcal{O}_d(hk)$ for $Get(k, [blkNo])$ just contains the latest version node v_l in default. If a block number $blkNo$ is specified, $\mathcal{S}$-Store returns the greatest version v_t such that $v_t \leq blkNo$. To seek version v_t from latest version node v_l, $\mathcal{S}$-Store selects the smallest version v_{p_1} in v_l's preceding version set PV_l such that $v_{p_1} \geq blkNo$. If $v_{p_1} = blkNo$, then the whole search is over. Otherwise, if $v_{p_1} > blkNo$, $\mathcal{S}$-Store searches v_t from v_{p_1} recursively until encountering a version v_{p_j}, such that $v_{p_j} \leq blkNo$ and $v_{p_j+1} > blkNo$. The visited version nodes $\langle v_l = v_{p_0}, v_{p_1}, \cdots, v_{p_j} = v_t \rangle$ form a hash linked list from v_l to v_t, which is the proof $\mathcal{O}_d(hk)$ for state with hash key hk at $blkNo$. For $Hist(k, s, e)$, it involves a read $Get(hk, e)$ to get version v_e at first, and then searches list in 0-$level$ forward to find a node v_s such that $v_s \leq s$. The nodes from v_s to v_e are the results for $Hist(\cdot)$. Sequentially, $\mathcal{S}$-Store constructs the witness $\mathcal{O}(hk) = \{\mathcal{O}_a(hk), \mathcal{O}_{idx}(hk), \mathcal{O}_d(hk)\}$ for a read.

In order to received state $\langle k, v \rangle$ based on witness, a client validates each of the three parts of the witness, $\{\mathcal{O}_a(hk), \mathcal{O}_{idx}(hk), \mathcal{O}_d(hk)\}$ separately. The client verifies the correctness of state v_t based on $\mathcal{O}_d(hk) = \langle v_l = v_{p_0}, v_{p_1}, \cdots, v_{p_j} = v_t \rangle$. For each version node v_{p_i} $(0 \leq i \leq j)$, the client computes hash of v_{p_i}, and then checks whether computed hash is identical to hash pointer contained in version node $v_{p_{i+1}}$. Second, client uses $\mathcal{O}_{idx}(hk)$ and root r_{idx} of MB-tree to check v_l. The client iteratively computes all the hashes of the sub-tree containing AASL V_{hk} from bottom to top in index level based on $\mathcal{O}_{idx}(hk)$. Last, the verification for root r_{idx} is a standard Merkle verification based on $\mathcal{O}_a(hk)$ and root r_a of AMB-tree.

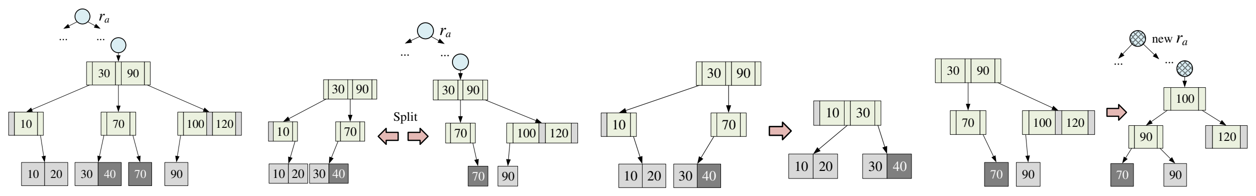
Algorithm 1: Split Operation on AMB-tree

```
Input: $T$: AMB-tree, $HK_s$: split hash key
Output: $T^*$: adjusted AMB-tree, $T_c$: MB-tree
1  $T_B \leftarrow$ MB-tree of zone that overlaps split hash key $HK_s$
2  $T_l, T_r \leftarrow T_B, T_B$
3  $p \leftarrow GetRoot(T_B)$
4  while $p \neq null$ do
5      $p_s \leftarrow$ the split child for $HK_s$ of $p$
6      $T_l \leftarrow RemoveRightChild(T_l, p, p_s)$
7      $T_r \leftarrow RemoveLeftChild(T_r, p, p_s)$
8      $p \leftarrow p_s$
9  $T_l \leftarrow AdjustLeftTree(T_l)$
10 $T_r \leftarrow AdjustRightTree(T_r)$
11 $T^* \leftarrow ReplaceZone(T_B, T_r)$, $T_c \leftarrow T_l$
12 return $T^*, T_c$
13 procedure AdjustLeftTree($T_l$) do
14     /*$T_l$: the left partial tree*/
15     $t \leftarrow GetRoot(T_l)$
16     while $t \neq null$ do
17         $T_l \leftarrow RemoveRightKey(T_l, t)$
18         $t_{rm} \leftarrow$ the rightmost child of $t$
19         if $Valid(t)$ is false then
20             $t_{ls} \leftarrow$ the left sibling of $t$
21             if $CanMerge(t, t_{ls})$ then
22                 $T_l \leftarrow Merge(T_l, t, t_{ls})$
23             else
24                 $T_l \leftarrow ShiftLeftKey(T_l, t_{ls}, t)$
25         $t \leftarrow t_{rm}$
26     $T_l \leftarrow HashResolve(T_l)$
27     return $T_l$
```

B. Split and merge

With the addition of a committee, some states are remapped into other committees according to consistent hashing, and some zones of different committees are divided into several disjoint sub-zones. Then the new committee aggregates half of these sub-zones to form its own AMB-tree. To enable each committee to quickly rebuild AMB-tree, we designed an efficient split and merge operation on AMB-tree. The division of a zone is a split operation on a MB-tree in AMB-tree. Given a MB-tree, we can split it in half at the middle of the root, each forming a separate MB-tree. However, it is not suitable for our scheme, since the split key is a random hash in $\mathcal{S}$-Store.

Algorithm 1 presents the split operation on AMB-tree T at a specified hash key HK_s, which returns an adjusted AMB-tree T^* and a partial MB-tree T_c cut from T which will move to the new committee. First, T_B is set to the MB-tree of zone overlapping HK_s, and then the left and right partial tree T_l and T_r are initialized to T_B (lines 1-2). Second, T_B is split by



(a) MB-tree of AMB-tree (b) Split into two partial trees (c) Adjust left partial tree (d) Adjust right partial tree

Fig. 3. Split on AMB-tree at hash key 64.

a traversal from root to leaf node (lines 3-8). For each node p in this traversal, the split child p_s of p is the root of a sub-tree containing hash key HK_s (line 5). Then, it removes all right siblings of p_s from T_l by function $RemoveRightChild()$ (line 6). Particularly, the function $RemoveRightChild(T_l, p, p_s)$ removes a child p_c of node p from T_l by removing a sub-tree rooted with p_c. Similarly, all left siblings of p_s are removed from T_r by function $RemoveLeftChild()$ (line 7). Third, T_l and T_r are adjusted to two valid MB-trees via function $AdjustLeftTree()$ and $AdjustRightTree()$ respectively (lines 9-10). Last, after adjusting all invalid nodes, T_B is replaced with right partial tree T_r to build a new AMB-tree by function $ReplaceZone()$ and T_c is assigned to T_l, since states in T_l have smaller hash keys than that in T_r and these states in T_l are remapped to new committee according to consistent hashing.

Due to the similarity of adjustments for both trees, we only describe the process for the left partial tree (lines 13-26). The adjustment for left tree is operated on the rightmost path from the root to leaf node of MB-tree. For each node t in this path, the keys without right child are removed by function $RemoveRightKey()$ (line 17), and t_{rm} is the rightmost child of t after removal. Then, if the sub-tree rooted with t is not a valid node in MB-tree (line 19), i.e. the number of index items in t is less than half of limit predefined by B+ tree, it tries to merge t and its left sibling t_{ls} by function $Merge()$ (lines 20-22). Therefore, the parent of t may be update for the merge of t and t_{ls}. If fail to merge, $\mathcal{S}$-Store shifts some keys from t_{ls} to t to enable it valid through their parent by function $ShiftLeftKey()$ (lines 23-24). After adjusting all invalid nodes, the hash pointers in nodes are recalculated by the function $HashResolve()$ to rebuild the property of Merkle tree (line 26). We just have to recalculate the hashes of nodes whose contents are updated.

To migrate states to the new committee, existing committees simply need to split MB-trees in AMB-trees and the new committee just has to aggregate MB-trees to rebuild a fresh AMB-tree. Therefore, the overheads for structure reconstruction are reduced significantly for all committees. The merge operation on AMB-tree is a reverse process of split, which tries to merge two contiguous MB-trees into a new one and rebuild a new AMB-tree.

VI. Scalability of $\mathcal{S}$-Store

A. Communications among committees

When a new committee joins, it needs to interact with existing committees to get its own state data. Each committee remains correct on a macro level, but the honesty of each individual validator is uncertain. It means a validator cannot simply trust the data sent by the validator of another committee. Therefore, we define three fundamental *communication functions* used by two committees C_s and C_r to exchange information as follows.

- $\boldsymbol{Req(R_s, C_r, args)}$: A validator R_s in committee C_s requests some data from committee C_r, where $args$ is the arguments. It has to promise that the returned data is agreed by a quorum ($n - f$ in PBFT) of committee C_r.
- $\boldsymbol{Sync(C_s, C_r, args)}$: A committee C_s synchronizes data from another committee C_r via function $Sync(\cdot)$, which guarantees a majority of C_s get the same data.
- $\boldsymbol{Opr(C_s, C_r, args)}$: A committee C_s deploys a concrete operation on committee C_r by function $Opr(\cdot)$, which ensures that the call of $Opr(\cdot)$ is the embodiment of a quorum of C_s. Meanwhile, $Opr(\cdot)$ may return some values to committee C_s, and promises that a majority of C_s receive returned data. Due to the space limit, we cannot detail the implementation of the aforementioned functions.

B. Committee addition protocol

Serial addition protocol. We first assume the addition and removal of committees are serial and then relax this assumption. Algorithm 2 presents the protocol for serial addition of committee C'. First, committee C' gains the information of zones I_{zone} and committee ID cid from existing committees $\{C_1, \cdots, C_m\}$ through $Sync(\cdot)$ function (lines 2-4). If the committee ID is greater than m_cid, m_cid is set to cid (lines 5-6). Second, committee C' selects c sub-zones based on I_{zone} in turn (lines 5-10). For each selection, C' generates a split hash key HK_s randomly by function $SplitKey()$ (line 8) and determines the zone Z_i that covers HK_s in committee C_{s_i} (line 9). Then C' deploys a split operation on Z_i by $Opr(\cdot)$ to produce two new sub-zones and get the root r_{idx_i} of MB-tree (new sub-zone) (line 10). In practice, to shorten the latency for $Opr(\cdot)$, transaction TX_{opr} is packaged on a priority basis in all committees. After that, the committee starts to synchronize the data of sub-zone asynchronously (line 13), which is detailed later. Third, committee C' aggregates MB-trees of c zones into an AMB-tree. Each validator does not wait for the accomplishment of state migration , instead it just uses the roots of MB-trees to build the upper Merkle tree (line 15). Last, the new committee C' starts to process transactions and provides read service (line 16).

State read and lazy migration. As described above, in order to avoid the data migration blocks the transactions

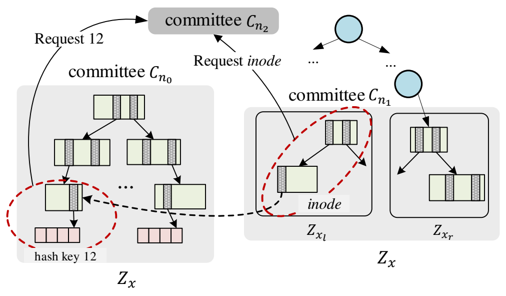


Fig. 4. Remote state request for concurrent addition.

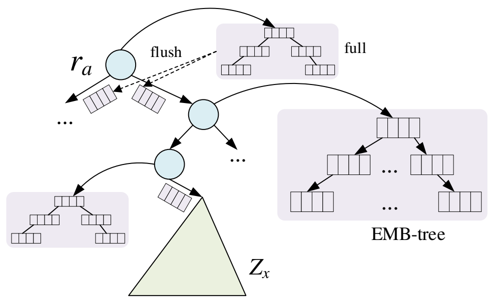


Fig. 5. AMB*-tree with buffer EMB-tree in aggregate level.

processing, $\mathcal{S}$-Store allows new committee C' to provide services without access to complete state data. However, committee C' may not have enough states in local AMB-tree to execute transaction. To this end, $\mathcal{S}$-Store exploits a lazy migration mechanism to enable the execution of transactions. When execute transactions agreed by C', each validator reads states from other committees by function $Req(\cdot)$ if necessary. Meanwhile, a witness of each requested state is transmitted for verification. Then, each validator inserts newly synchronized data into the local AMB-tree.

Before writing data to AMB-tree, each validator still needs to synchronize the index nodes and relevant version nodes from the corresponding committee. The write to AMB-tree may trigger the split of nodes in MB-tree (i.e., breaking the limit of index items in a node), so the validator needs to request some extra nodes in MB-tree to adjust the partial structure. When a validator is idle, it switches to synchronize the left states from other committees by function $Sync(\cdot)$. After state data migration, the corresponding committee can remove the MB-tree to free up the storage space. Although the lazy migration manner increases the latency for transaction execution in the new committee, it allows the committee to start service quickly.

Concurrent addition protocol. The situation becomes complex when multiple committees add concurrently. Specifically, a committee may split a zone in another new committee, who does not accomplish state synchronization for it. For example, a new committee C'' picks up zone Z_x that belongs to another new committee C', and splits it into two sub-zones Z_{x_l} and Z_{x_r}. However, C'' cannot synchronize data of the selected zone Z_{x_l} from C' because C' has not synchronized all states in zone Z_x from another committee C yet.

A naive method is to lock a zone during synchronization and other split operations are blocked if the zone is locked, which increases the latency of addition. Instead, an efficient and realistic way is to allow C'' to request Z_{x_l}'s data from C directly. The details of this processing are as follows. (1) New Committee C'' deploy split operation at hash key HK_s on zone Z_x in committee C' by function $Opr(\cdot)$. (2) Upon receiving split operation from C'', C' splits zone Z_x into Z_{x_l} and Z_{x_r} accordingly, and sends back the root of Z_{x_l}. To split Z_x, committee C' may request some necessary nodes from committee C, which surround the path from root r_a of Z_x's MB-tree to leaf containing split hash key HK_s, as depicted in Figure 3, . (3) Then, C' rebuilds its AMB-tree based on zone Z_{x_r} and stops synchronizing data for Z_{x_l}. When committee C'' receives root of Z_{x_l}, it starts to synchronize data of Z_{x_l} from committee C and C'.

Algorithm 2: Serial committee addition protocol

```
Input: $C'$: new committee
  $\mathcal{C} = \{C_1, \cdots, C_m\}$: existing committee set
  $c$: number of zones managed by a committee
1  $I_{zone} \leftarrow \emptyset$, $T_a \leftarrow \emptyset$, $m\_cid \leftarrow 0$
2  for $C_i$ in $\mathcal{C}$ do
3      $I, cid \leftarrow Sync(C', C_i, ZONE\_INFO)$
4      $I_{zone} \leftarrow Append(I_{zone}, I)$
5      if $cid > m\_cid$ then
6          $m\_cid \leftarrow cid$
7  for $i$ from 1 to $c$ do
8      $HK_s \leftarrow SplitKey(i)$
9      $Z_i, C_{s_i} \leftarrow SelectZone(I_{zone}, HK_s)$
10     $r_{idx_i} \leftarrow Opr(C', C_{s_i}, SPLIT\_ZONE, Z_i, HK_s)$
11     $T_a \leftarrow Append(T_a, r_{idx_i})$
12     /*work in background concurrently*/
13     $Sync(C', C_{s_i}, ZONE\_DATA, r_{idx_i})$
14 $SetCommitteeID(C', m\_cid + 1)$
15 $T \leftarrow AggregateZone(T_a)$
16 $StartService(C', T)$
```

When read a state with hash key hk in Z_{x_l}, committee C'' first requests it from C'. If C' has not synchronized the state from C, it returns the path from root to the deepest index node $inode$ in MB-tree of Z_{x_l} to C''. Then, C'' turns to request the state from C based on $inode$. Figure 4 illustrates an example of remote data request for hash key 12. Once finishing the synchronization of Z_{x_l}, C'' informs C' its accomplishment, and C' removes all data of zone Z_{x_l}. Only when C' obtains all data of Z_{x_r}, it can tell C to remove data of zone Z_x. For concurrent addition of committees, the lazy migration may involve multiple committees, but the process is similar. Due to the space limit, we do not present the details for committee removal, which can be implemented based on the merge operation of AMB-tree.

C. Optimization for AMB-tree

Both $Get(\cdot)$ and $Put(\cdot)$ may incur remote data requests. It is inevitable for a validator to request states for $Get(\cdot)$ caused by transaction execution from other committees. We can improve the performance of $Put(\cdot)$ by reducing remote data request. Inspired from *Fractal-tree* [11], we propose an enhanced structure called AMB*-tree, as depicted in Figure 5, to cache states. Specifically, each node in aggregate level

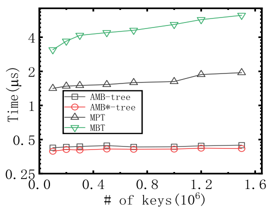


Fig. 6. Latency for $Get(\cdot)$.

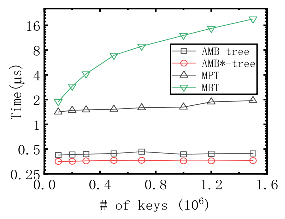


Fig. 7. Latency for $Put(\cdot)$.

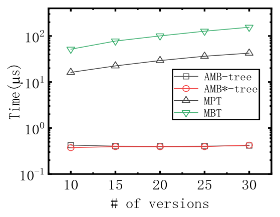


Fig. 8. Latency for $Hist(\cdot)$.

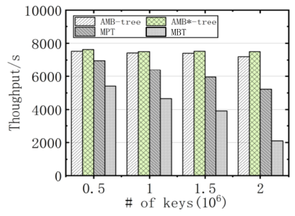
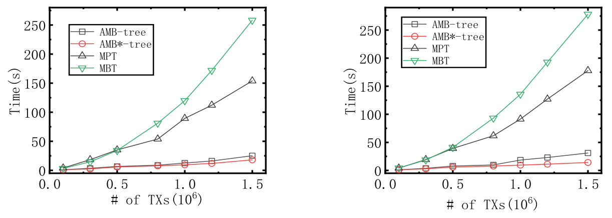


Fig. 9. Throughput based on variable structures.



(a) r:w=1:1

(b) r:w=1:3

Fig. 10. Execution time against number of transactions.

is associated with a fixed-size buffer and states in a buffer is organized as an *Embedded Merkle B+ tree* (EMB-tree). When a state is written to $\mathcal{S}$-Store, it is inserted into the EMB-tree of root r_a. When a buffer is full, the states in EMB-tree are flushed downwards a level. Eventually, every state arrives at a leaf in the aggregate level and is inserted into MB-tree in the index level. All EMB-trees are maintained in the main memory, thereby buffers of all nodes can be seen as a cache for write operations. The benefits of this approach includes: (1) most inserted states are cached in the aggregate level without remote data request; (2) The performance of $Get(\cdot)$ is improved since the latest updates are cached in memory.

To read a state with hash key hk in AMB*-tree, $\mathcal{S}$-Store needs to check the buffers of nodes in the traversal from root r_a to leaf (AASL) of state. If state resides in EMB-tree, $\mathcal{S}$-Store needs not to search it in index level. Meanwhile, the EMB-tree is still able to provide integrity proof for any state. Otherwise, $\mathcal{S}$-Store is forced to search index and data level as described in Section V-A. As new committee joins, some zone Z_x of AMB*-tree may be split. There may be states belonging to Z_x are buffered in aggregate level. Therefore, $\mathcal{S}$-Store flushes all states falling in Z_x to its MB-tree without affecting other states, and then splits Z_x's MB-tree.

VII. Evaluation

A. Implementation and setup

As a popular benchmark for OLTP workload, SmallBank is also widely adopted for blockchain systems [9]. The codes of smart contracts are written in Go language. The hash function is Keccak-256 and the threshold signature is implemented based on the BLS signature scheme [5]. The AMB-tree and AMB*-tree are implemented in about 3000 lines of C++ codes. Each node in aggregate level carries a buffer (EMB-tree) with a size of 2MB for AMB*-tree. We use the MPT[1] and MBT[2] as comparison. We integrate the consensus module of Fabric 0.6 [3] as our PBFT implementation, which can process 13,000+ transactions per second.

All experiments are conducted upon a cluster of 16 machines divided into 4 committees evenly, each of which is equipped with 16 CPU cores with 2.8GHz, 96GB RAM, 512GB SSD disk space and 1Gbps network bandwidth. Each committee runs as an instance of a PBFT cluster and the 2PC-based method [8] is used to tackle cross-committee transactions. To this end, a committee is assigned as the coordinator, which dispatches transactions to related committees and collects votes from them.

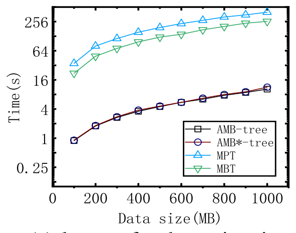
B. Experimental results

We evaluate $\mathcal{S}$-Store based on AMB-tree and AMB*-tree compared with MPT and MBT from two aspects: basic performance of read/write and scalability.

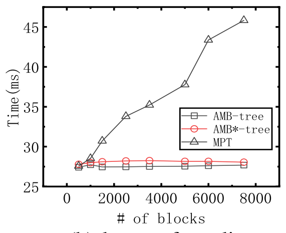
1) Results on the performance of AMB-tree: Figure 6-8 reports the latency for data access of three storage interfaces for 4 structures: AMB-tree, AMB*-tree, MPT, and MBT. Figure 6 and 7 present the latency of interface $Get(\cdot)$ and $Put(\cdot)$ as the number of keys varies from 0.1M to 1.5M. It turns out that AMB-tree and AMB*-tree outperform MPT and MBT significantly. For instance, with 0.8M keys, the latency of $Get(\cdot)$ and $Put(\cdot)$ for MBT exceeds 4 and 8 μs respectively, one order of magnitude slower than that of AMB-tree. Besides, the AMB*-tree performs better than AMB-tree with the enhancement of EMB-tree. Figure 8 compares the latency of $Hist(\cdot)$ for 4 structures against the number of accessed versions upon a data set with 0.1M keys. To add multi-version accesses to MBT, the version number of each state is appended to its key. We track multiple versions in MPT through seeking each version upon all related snapshots. Similarly, AMB-tree and AMB*-tree obtain lower latency for $Hist(\cdot)$ query, because they can load consecutive versions in a batch. Instead, the latency of MPT and MBT for $Hist(\cdot)$ is proportional to the number of versions.

Figure 9 presents the influence on throughput of system against the number of keys. For MPT and MBT, the execution cost of transactions becomes the bottleneck gradually with more keys, i.e. the throughput of blockchain based on MBT is about 2100, less than one third of AMB-tree and AMB*-tree. As expected, AMB*-tree achieves higher throughput than AMB-tree.

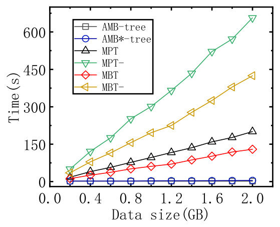
[1]https://github.com/ethereum/go-ethereum/tree/master/trie
[2]https://github.com/hyperledger-archives/fabric/tree/core/ledger
[3]https://github.com/rleonardco/fabric-0.6



(a) latency for data migration

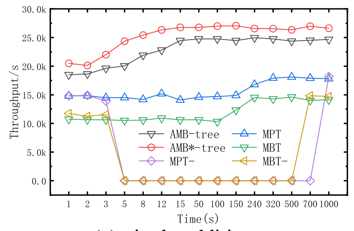


(b) latency for split

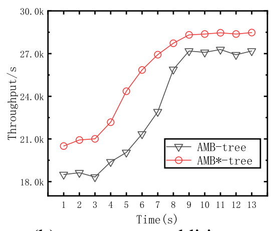


(c) latency for addition

Fig. 11. Performance of committee addition protocol.



(a) single addition



(b) concurrent addition

Fig. 12. Throughput of system during addition.

Figure 10 reports the total commit latency of all structures, excluding consensus overhead, against the number of transactions with two read/write ratios, 1:1 and 1:3. Every transaction may write new states to blockchain, thereby the number of keys increases with execution of transactions. As more transactions are executed, the latency of AMB-tree and AMB*-tree increases linearly with a small factor, while MBT gains the worst result. As shown in Figure 10(a), MBT spends more than 250s in executing 1.5M transactions. Besides, with a higher ratio of write operation, as presented in Figure 10(b), the execution time increases as well, since writing is a more time-consuming operation. It takes 20 more seconds for blockchain to execute 1.5M transactions based on MBT with a read/write ratio 1:3 instead of 1:1.

2) Results of committee addition: Figure 11 and 12 evaulate the performance of committee addition. The new committee uses function $Sync(\cdot)$ to synchronize MB-tree of each zone from others. The function $Sync(\cdot)$ involves multiple rounds of $Req(\cdot)$, each of which requests a sub-tree serialized into a 2M network package. In default, we apply consistent hashing to divide space of hash key for MPT and MBT, and aggregate multiple MPTs or MBTs via a Merkle tree like the aggregate level in AMB-tree.

Figure 11(a) reports the latency for data migration for different structures against the data size. AMB-tree and AMB*-tree outperform MPT and MBT again, since MPT or MBT involves massive random read operations on KVS. When the data size is 1,000MB, the overheads of AMB-tree and AMB*-tree for native read on states account for 10% of all time cost, while they increase to 30% and 50% for MPT and MBT respectively.

Figure 11(b) reports the latency for split implemented based on function $Opr(\cdot)$ of different structures against the number of blocks executed by committee. To split an MPT which snapshots global state at each block, the only way is to split all snapshots, and all states are reinserted into two new MBTs for split of MBT. The latency of AMB-tree and AMB*-tree remains about 27ms, while the latency of MPT grows as the number of blocks increases since it has to split all snapshots. The latency of MBT is more than 30s with 1.5M different keys, which is not depicted because the value is too large.

Figure 11(c) compares the time overhead for the addition of a committee. The lines MPT- and MBT- represent the results for MPT and MBT under a modulo partition manner. AMB-tree and AMB*-tree gain smaller latency since the protocol is not blocked under lazy migration mechanism. Instead, MPT and MBT consume much more time for blockage caused by state synchronization. The latency of MPT- or MBT- grows rapidly with greater data size, since great data need to be migrated before.

Figure 12(a) presents the throughput of blockchain system during the addition of a committee when data size is 2GB. For each test, the new committee joins system at the 3rd second. With AMB-tree and AMB*-tree, the new committee starts to process transactions quickly, while the service of entire system is blocked for more than 500s with MPT- or MBT- due to state redistribution. With MPT and MBT, only the new committee is blocked for 150s and 300s respectively. As time goes by, the throughput of AMB-tree and AMB*-tree increases correspondingly. Moreover, AMB*-tree outperforms AMB-tree due to the use of EMB-tree. Figure 12(b) shows the result of concurrent addition of two committee when 2 zones are split concurrently. The addition of multiple committees entails no significant side effect on the throughput, and two committees start to work upon addition protocol is finished.

VIII. Conclusions

In this paper, we propose $\mathcal{S}$-Store to manage state data for sharded blockchain from three aspects. First, we apply the consistent hashing to partition global state for load balance among committees. Second, we design novel data structures AMB-tree and AMB*-tree to manage part of global state data in each committee, which enhances storage scalability of system for its excellent read/write performance. Third, we introduce a committee addition protocol based on AMB-tree and AMB*-tree to new committees without breaking the services of system.

Acknowledgments

This work was supported by National Key Research and Development Program of China (2021YFB2700100), and National Science Foundation of China (61972152, U1911203 and U1811264).

REFERENCES

[1] Ant Group BaaS products. https://antchain.net/products/myChain, 2021.

[2] Hyperledger. https://www.hyperledger.org, 2021.

[3] Tendermint document. https://tendermint.com/docs/, 2021.

[4] M. J. Amiri, D. Agrawal, and A. El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *IEEE ICDCS*, pages 1337–1347. IEEE, 2019.

[5] D. Boneh, C. Gentry, B. Lynn, H. Shacham, et al. A survey of two signature aggregation techniques. *RSA cryptobytes*, 6(2):1–10, 2003.

[6] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3:37, 2014.

[7] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[8] H. Dang et al. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 SIGMOD*, pages 123–140. ACM, 2019.

[9] T. T. A. Dinh, J. Wang, et al. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM SIGMOD*, pages 1085–1100. ACM, 2017.

[10] T. Hanke, M. Movahedi, and D. Williams. Dfinity technology overview series, consensus system. *arXiv:1805.04548*, 2018.

[11] W. Jannen, J. Yuan, Y. Zhan, et al. Betrfs: A right-optimized write-optimized file system. In *13th $USENIX$ FAST*, pages 301–315, 2015.

[12] D. Karger, E. Lehman, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, volume 97, pages 654–663, 1997.

[13] E. Kokoris-Kogias, P. Jovanovic, et al. Omniledger: A secure, scale-out, decentralized ledger. *IACR Cryptology ePrint Archive*, page 406, 2017.

[14] F. Li et al. Authenticated index structures for outsourced database systems. Technical report, Boston University Computer Science Department, 2006.

[15] L. Luu et al. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM CCS*, pages 17–30. ACM, 2016.

[16] R. C. Merkle. Method of providing digital signatures, Jan. 5 1982. US Patent 4,309,569.

[17] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM JACM*, 15(4):514–534, 1968.

[18] S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[19] S. Nathan, C. Govindarajan, et al. Blockchain meets database: Design and implementation of a blockchain relational database. *Proceedings of the VLDB Endowment*, 12(11).

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[21] S. Ponnapalli, A. Shah, et al. Scalable and efficient data authentication for decentralized systems. *arXiv preprint arXiv:1909.11590*, 2019.

[22] A. Sonnino. Chainspace: A sharded smart contract platform. 2017.

[23] J. Wang et al. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th $USENIX$ NSDI*, pages 95–112, 2019.

[24] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[25] M. Yin, D. Malkhi, et al. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM PODC*, pages 347–356. ACM, 2019.

[26] M. Zamani et al. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM CCS*, pages 931–948. ACM, 2018.

[27] H. Zhang, C. Jin, and H. Cui. A method to predict the performance and storage of executing contract for ethereum consortium-blockchain. In *International Conference on Blockchain*, pages 63–74. Springer, 2018.