B+AVL trees: towards data structures for robust and efficient blockchain state synchronization

Michele Cattaneo
Università della Svizzera italiana
Lugano, Switzerland
michele.cattaneo@usi.ch

Eliã Batista
Università della Svizzera italiana
Lugano, Switzerland
delime@usi.ch

Fernando Pedone
Università della Svizzera italiana
Lugano, Switzerland
fernando.pedone@usi.ch

***Abstract*—This paper addresses state transfer in blockchain systems, crucial for new or recovering peers. Current systems use periodic snapshots and cryptographic structures like Merkle trees for efficient validation in Byzantine fault tolerant scenarios. Recent improvements use chunk-based data structures (e.g., AVL* trees, Merkle B+trees) to reduce overhead and enable efficient chunk-level validation. This paper introduces B+AVL trees, a novel chunk-based structure that significantly advances the state transfer process. B+AVL trees combine the balancing mechanism of AVL trees with Merkle hashing for validation, enabling efficient and secure state transfer. They are more space-efficient and simpler to manage than AVL* trees, and they use more compact validation proofs than Merkle B+trees. B+AVL trees represent a major improvement by balancing binary trees and managing chunks in a way that optimizes performance and data validation, ensuring robust state synchronization in blockchain systems. The paper thoroughly assesses B+AVL trees and compares them to competing approaches.**

***Index Terms*—Blockchain, Data structures, State synchronization, State machine replication, Byzantine fault tolerance**

I. Introduction

This paper addresses state transfer in blockchain systems, which occurs when a peer recovers from a failure or joins the network. To catch up with the rest of the system, the peer must update its state to match the operational peers. Modern blockchain systems (e.g., [1], [2]) use periodic snapshots of the system state to improve state transfer performance. A recovering peer first downloads a recent snapshot and subsequent blockchain data (i.e., full blocks or block headers containing summaries). The peer installs the snapshot and then replays the transactions that occurred after it to fully reconstruct the current system state.

The problem of state transfer in blockchain systems is also relevant to state machine replication subject to malicious (i.e., Byzantine) replicas. A distinctive aspect is that blockchain peers store state in a special data structure, such as a Merkle tree [3] or Merkle-Patricia-tree [2]. In Tendermint's AVL+ trees [4], every leaf node stores a cryptographic hash of its data, and every inner node stores a hash calculated from the hash of its children. A recovering peer can validate a snapshot by computing the root hash of the Merkle tree and comparing it with the root hash stored in the trusted block header.

To decrease the time it takes for a new peer to join the blockchain, a snapshot can be divided into *chunks*, each containing multiple tree nodes, and a recovering peer can download chunks from many operational peers concurrently. Variations of this technique have been implemented by both blockchain (e.g., [5]) and Byzantine-fault tolerant state machine replication systems (e.g., [6]–[8]). Some approaches have been proposed to improve the performance and robustness of this scheme [9], [10]. Building chunks from the state tree to serve a recovering peer and rebuilding tree nodes from chunks can be an expensive operation involving tree traversal, serialization, and deserialization of tree nodes. This cost can be substantially reduced by storing the system state in "chunk-based" data structures, as in AVL* trees [9] and Merkle B+trees [10]. While AVL* trees embed state subtrees into chunks, Merkle B+trees store the leaves of a cell in a chunk. In both cases, a peer serving the state to a recovering peer must only traverse and serialize the list of existing chunks.

The recovering peer receives large data units, saving resources similarly to batching [11], and avoids allocating individual tree nodes by relying on chunks. Moreover, chunks can be validated independently using Merkle proofs, allowing the peer to detect invalid data without rebuilding the entire tree [5]. Such validation is crucial in blockchain systems to protect against malicious peers that could tamper with data to delay synchronization and waste resources.

In this paper, we introduce B+AVL trees, a novel chunk-based data structure inspired by AVL* and Merkle B+trees, building on their advantages while avoiding their shortcomings. B+AVL trees are balanced binary trees that use rotations, like AVL trees, and integrate Merkle hashes to enable individual validation of nodes and subtrees, as in AVL* trees. However, in AVL* trees, maintaining balance through rotations can affect chunk roots, often requiring chunk splits before or after rotations and adding significant complexity. B+AVL trees, inspired by B+trees, avoid these complications by filling chunks sequentially and splitting them when full, without requiring complex cross-chunk rotations. They are more space-efficient than AVL* trees, allowing chunks to store more data, and they provide more compact validation proofs than Merkle B+trees.

Table I compares the main properties of blockchain data structures discussed in the paper. It highlights the complexity of managing each structure, the size of cryptographic proofs (proof size), and whether it supports self-verifiable chunks. The proof size denotes the total number of elements (hashes)

required to verify the inclusion of a specific entry in the tree. The comparison considers AVL+ trees, AVL* trees, Merkle B+trees, and B+AVL trees, where n denotes the total number of entries in a tree, and b the number of entries in a B-tree cell.

Data structure	Complexity	Proof size	Self-verifiable chunks
AVL+ tree	low	$\mathcal{O}(\log_2 n)$	no
AVL* tree	high	$\mathcal{O}(\log_2 n)$	yes
Merkle B+tree	low	$\mathcal{O}(\log_b n \cdot b)$	yes
B+AVL tree	medium	$\mathcal{O}(\log_2 n)$	yes

TABLE I: Blockchain data structures (n: number of entries, b: entries in a B-tree cell).

We have thoroughly evaluated B+AVL trees and compared them to competing strategies. We show that B+AVL trees are competitive with other techniques in insertion operations, and offer up to $3.17\times$ improvement in search performance, thanks to their optimized data layout. They provide significantly smaller Merkle proof sizes, reducing them to less than 1% of the size in conventional B+trees, and offer more stable Merkle proof sizes compared to AVL* trees. They also enhance space efficiency through balanced, contiguous chunk organization, achieving up to 10% more space efficiency with smaller chunk sizes, despite some variability as the tree expands. During state synchronization, B+AVL trees outperform existing techniques both in the absence and presence of attacks, delivering up to $4.7\times$ better performance in the latter scenario. They achieve faster synchronization through streamlined serialization and rebuild processes. Under Byzantine attacks, B+AVL's self-verifiable chunks enable rapid detection and recovery, maintaining robustness even as the number of Byzantine peers increases, resulting in a performance that is $23.5\times$ better when facing four Byzantine peers.

The rest of the paper is structured as follows. Section II presents the system model and assumptions. Section III motivates and recalls AVL* trees and Merkle B+trees. Section IV introduces B+AVL trees. Section V experimentally evaluates B+AVL and compares it to competing techniques. Section VI reports on related works, and Section VII concludes the paper.

II. Background

This section introduces the system model and basic assumptions, explains how application state is managed in blockchain systems, and introduces the state synchronization problem.

A. System model and assumptions

We consider a blockchain system implemented by distributed peers that communicate by exchanging messages. Peers are *honest* if they follow the protocol specification, or *malicious* (i.e., Byzantine) if nothing can be assumed about their behavior. Communication between honest peers is reliable. There is an unbounded number of clients and a bounded number of peers. Clients submit transactions to the blockchain, and peers order and execute client transactions. The blockchain behaves correctly if a fraction of the peers, typically more than two-thirds, is honest [12]. The system is *partially synchronous* [13]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [13], and it is unknown to the peers.

B. Blockchain and state management

A blockchain is a distributed ledger, an append-only log of transactions. The append-only log is structured as a linked-list of blocks, each block divided into a header and a body. The header contains, among other information, a cryptographic link to the previous block and a hash of the state. The body contains a list of transactions, each transaction cryptographically signed by the client that submitted it. We consider blockchain systems that ensure a total order on linked blocks (e.g., [14]). Therefore, peers agree on the next block to be appended by means of a Byzantine fault-tolerant consensus protocol (e.g., [15], [16]).

Peers rely on efficient data structures to maintain application state, that is, state created and modified as a result of the execution of client transactions. Merkle trees [17] or similar balanced trees (e.g., Merkle-Patricia trees [2], [9], [10]) are data structures typically used. In addition to providing efficient data access, these structures empower peers with the ability to prove the integrity of the state to the clients. In a Merkle-tree, each leaf holds a piece of the state and its hash, and each inner node holds the hash of its children. The hash of the tree's root (i.e., Merkle-root) is stored in the block header to ensure its integrity. Merkle trees provide succinct proofs of data integrity of any node of the tree. One can prove in logarithmic time and space that a leaf v belongs to the tree by providing the hash of the siblings of the tree nodes in the path from v to the Merkle-root.

Some blockchain systems provide an API for applications to handle state management. The API includes basic operations:

- **insert(key, value)** adds a new key-value pair to the state; if the key already exists, it updates the value.
- **search(key)**: searches for a key in the tree and returns its associated value, or *null* if the key is not found.
- **delete(key)**: removes the key-value pair associated with the given key.

Blockchain systems typically use multi-version Merkle trees, where a new version is logically created with each block. To boost performance, copy-on-write is used: the current Merkle root is copied to form the new version, and only modified nodes are duplicated. In some inter-blockchain protocols, relayer peers traverse old tree versions to verify state proofs from other blockchains. [18].

C. State synchronization

As in state machine replication [15], a new, slow, or recovering blockchain peer needs to catch up with operational peers. A peer can catch up by retrieving all ordered blocks and executing all transactions in these blocks. Merkle-trees enable more efficient techniques: peers can download a recent

snapshot of the application state from operational peers and execute only transactions that succeed the downloaded snapshot. The recovering peer can check the integrity of the downloaded snapshot by computing the Merkle-root hash and comparing it to the hash stored in the blockchain block that corresponds to the downloaded snapshot. Variations of this approach have been used in popular blockchain clients [19].

The data structure used to encode the application state and Merkle-tree has an important impact on the performance of state synchronization [9]. The main aspects involve how data is allocated in the data structure (e.g., chunked data versus individual data items) and how peers can verify the integrity of a snapshot. In the next section, we delve into the details of data structures relevant to our study.

III. Blockchain data structures

In this section, we motivate and recall the notions of AVL* trees and Merkle B+trees, data structures designed to optimize the performance of blockchain data management and state synchronization. We conclude with a discussion comparing AVL* trees and Merkle B+trees.

A. Motivation

Some blockchain systems batch entries of a tree data structure and serialize and deserialize batches instead of individual tree entries to improve the performance of state synchronization. Peers in CometBFT, for example, use AVL+ trees (see Section III-B) for state management [1]. A peer serving a snapshot traverses the tree, groups tree nodes into chunks, and serializes the whole chunk. Serializing a chunk that contains multiple tree nodes is significantly more efficient than serializing each node individually. This efficiency arises from grouping all the nodes within a chunk together in contiguous memory, which streamlines the serialization process.

When nodes are stored contiguously, the entire chunk can be serialized in a single operation, eliminating the need for looping through individual nodes. In contrast, if a binary tree is used where nodes are scattered across memory, each node must be serialized separately, requiring a loop to iterate through all nodes. Figure 1 compares the performance of serializing and deserializing a tree using the two techniques, chunked data and individual data items. Both serialization and deserialization benefit significantly from chunked nodes, and the advantage increases with chunk size. The chunked approach outperforms the serialization and deserialization of individual nodes by a factor of 26× when chunks contain 4096 data items, and by a factor of nearly 6× when chunks contain 1024 data items.

Although chunking speeds up serialization and deserialization, it introduces the overhead of building chunks during state transfer. A more aggressive alternative is to use data structures naturally organized around larger data units, such as AVL* trees [9] and Merkle B+trees [10], where a peer serving a snapshot can avoid on-the-fly tree traversal and chunk construction. Recovering peers also benefit by not needing to allocate individual tree nodes.

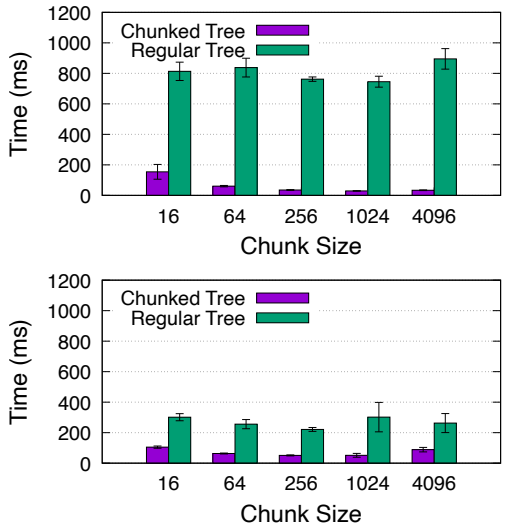


Fig. 1: Time to fully serialize (top) and deserialize (bottom) a chunked and a regular tree with 1 million elements, and various chunk sizes (each element is 512 bytes).

In addition to (de)serialization efficiency, chunked structures offer another crucial advantage in Byzantine settings. When state is organized into independently verifiable chunks, corrupted chunks can be detected immediately upon receipt, without requiring the download of the entire tree or structure. This enables early detection and rejection of invalid data, significantly speeding up recovery. In contrast, non-chunked structures must often download and process the complete structure before validation can occur, exposing systems to much higher delays and inefficiencies when facing faulty or malicious peers.

B. AVL* trees

AVL* trees extend AVL and AVL+ trees. An AVL tree is a self-balancing binary tree where the heights of a node's children differ by at most one. When this balance is violated (after insertions or deletions), rotations restore it, ensuring logarithmic-cost search, insertion, and deletion operations. Insertions may require up to two rotations; deletions can require rotations proportional to the tree's height. An AVL+ tree is a variant where only leaves store values; inner nodes exist only in memory and can be recomputed from the leaves, reducing persistent storage needs.

AVL* trees [9] are AVL+ trees where leaves are batched into larger data structures (i.e., chunks) to improve state management and synchronization (see Figure 2 (a)). To provide a proof of integrity of any element of an AVL* tree (i.e., leaf), tree nodes are labeled with cryptographic hashes, leading to what is known as a hash tree or Merkle tree. In a Merkle AVL tree, each leaf has the hash of its value, and each inner node has the hash of its children. In Figure 2 (a), the highlighted hashes h_8, h_c, h_d, and h_f form the proof required to verify the inclusion of leaf 7 in the tree. These hashes are kept up-to-date as the tree evolves, and to retrieve them, a peer traverses

the path from the leaf to the root, collecting the sibling hashes along the way. During validation, a peer uses these hashes to reconstruct the path up to the root and compares the resulting root hash to the trusted root hash stored in the corresponding block header. If the computed and stored root hashes match, leaf 7 is confirmed as a valid entry. A proof in a Merkle AVL tree has size $O(\log_2 n)$, where n is the number of tree leaves.

C. Merkle B+trees

B-trees generalize binary trees by allowing each inner node to have between $\lceil m/2 \rceil$ and m children, where m is the tree order [20]. All leaves are at the same level, and searches, insertions, and deletions have logarithmic time complexity. B-trees are designed for storage and communication efficiency by configuring cell sizes via the tree order. In a B+tree, internal nodes store copies of the keys, while leaves store keys and values; leaves may also include pointers to their successors for faster sequential access.

Differently from AVL* trees, which require sophisticated algorithms to maintain parts of the tree in chunks without violating tree invariants, Merkle B+trees are a straightforward variation of B+trees, extended to provide proofs of integrity [10]. Each entry in a leaf cell has the hash of its value, and each entry in an inner cell has a hash computed based on the hash of its children. In Figure 2 (b), the integrity of leaf 7 can be checked with hashes h_5, h_6, h_8, h_a and h_b. A proof in a Merkle B+tree has size $O(b \times log_b\ n)$, where n is the number of leaves in the tree and b the number of elements in a cell.

D. Discussion

Both AVL* trees and Merkle B+trees improve blockchain data management and state synchronization by embedding chunking into the tree structure. Chunk-based designs enable efficient data serialization and deserialization, reducing communication overhead (see Section III-A). Another advantage is the use of *self-verifiable* chunks: a recovering peer can immediately validate a chunk upon receipt by checking an embedded subtree root proof [9], without rebuilding the full tree. This significantly improves recovery performance in the presence of malicious peers, who might otherwise delay synchronization by serving invalid data (see Section V).

IV. The B+AVL tree

The B+AVL tree is an advanced data structure designed for efficient storage, search, and proof generation over large collections of key-value pairs. It combines features of self-balancing binary search trees (like AVL trees) and chunked leaf storage (as in B+ trees), allowing both efficient updates and compact proofs. The B+AVL tree maintains balance across two levels: the global tree structure and the internal organization of chunks within leaves.

A. B+AVL tree structure

A B+AVL is a tree at two levels. Firstly, it is a self-balancing binary search tree with each inner node containing the hashes of its two child nodes and a copy of the smallest key in their right subtree (see Figure 3). Differently from an ALV* tree, however, leaf nodes do not only host space for a single key-value pair. Instead, leaf nodes are implemented like B+Tree leaf nodes, effectively representing a chunk of data. This allows to have chunks in contiguous memory for efficient serialization. When a chunk is full, it is split and the two halves are hosted in two new leaf nodes. A new inner node is then added to link the leaves. To avoid linear terms in the size of a proof for individual elements, which would be introduced by the flat chunks, each chunk is augmented with an additional, *in-chunk*, hash tree. This is not a search tree and its sole purpose is to provide logarithmic-sized proofs to elements of a chunk. We refer to the root of this tree as the *chunk root*. A chunk root provides the first elements of the hash proof for a whole chunk. When a chunk proof and the proof of an element within a chunk are concatenated, a logarithmic-sized proof for element within the overall tree is obtained.

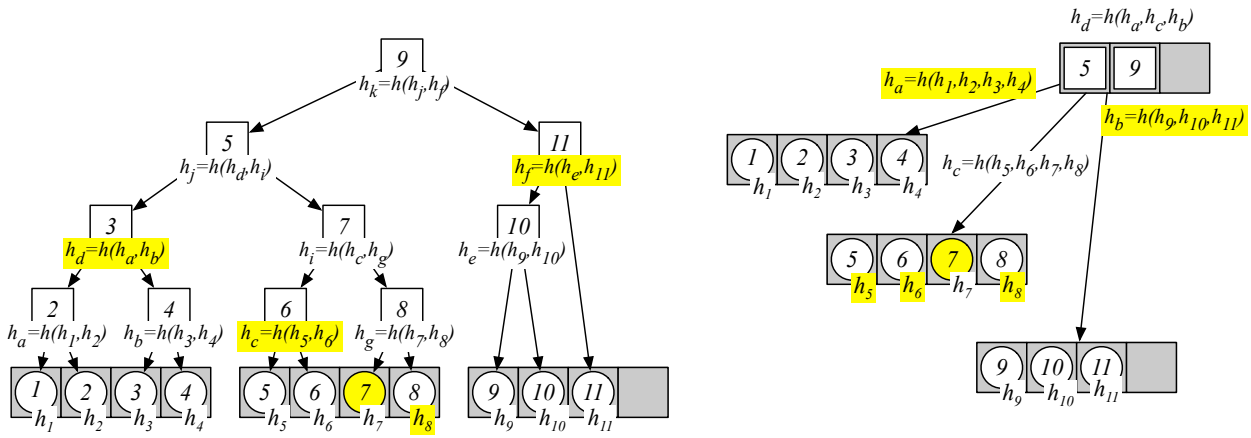
A B+AVL tree rotates inner nodes by treating the leaves containing a chunk, no matter the size, as a unit. Like an AVL tree, the height of two child nodes can differ by 1 at most. However, because a chunk vary from being half-full to full, the height difference between child nodes in the in-chunk tree can also differ by up to 1. Consequently, after expanding chunks in the logical tree view, the height difference between the children of any node in a B+AVL tree can be up to 2, compared to the maximum difference of 1 in an AVL tree. This, however, avoids the complex rotations across chunks that a AVL* tree needs to undergo to stay balanced.

B. Chunks

A B+AVL tree represents a key-value store with fixed-sized keys and variable sizes values. Each chunk contains three arrays for keys, values and hashes respectively. The keys array has known size and contains sorted keys. Each key is augmented with the size and starting position of the corresponding value in the values array. This permits new values to simply be appended to the values array and limit data movement during insertions. We pre-allocate an estimated size for the value array and if the overall size is surpassed, the array is re-allocated. For a chunk with M keys, the hashes array contains $2M - 1$ entries to represent the nodes of in-chunk hash tree in flat memory. The first half of this array contains the hashes of inner nodes of the tree embedded in the chunk, while the second half contains direct hashes of key-value pairs. Thanks to the splitting mechanism, it is easy to link consecutive chunks to simplify range queries.

C. Searches and proofs

To find a value, the inner nodes are traversed until a leaf node is reached. During this traversal, a chunk proof can be obtained by keeping track of the hash values of the sibling's hash for each node encountered. Within the chunk, the value is retrieved with a binary search in the keys array. Because the in-chunk tree is not a search tree, the proof for the specific element can not be obtained while searching. Instead, one must



(a) AVL* tree (b) Merkle B+tree

Fig. 2: Different data structures used to store state in a blockchain peer: (a) AVL* tree and (b) Merkle B+tree. (Circles depict keys and values, white squares contain inner keys, used to navigate the data structure, gray area represents data chunks.) The proof that 7 is a valid element in the AVL* tree includes hashes h_8, h_c, h_d and h_f, while in the Merkle B+tree the proof includes hashes h_5, h_6, h_8, h_a and h_b (highlighted in yellow).

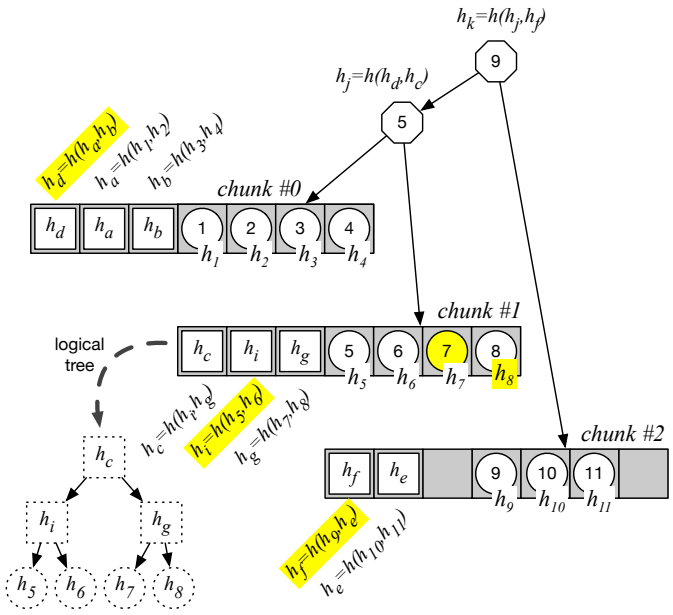


Fig. 3: The B+AVL tree. Gray area represents data chunks, circles depict keys, octagons contain inner keys, white squares are hashes that build up a logical tree. The proof that 7 is a valid element contains hashes h_8, h_i, h_d and h_f.

traverse the in-chunk hash tree upwards knowing the parent relationship[1] of each node. A proof of an individual element is obtained by concatenating the chunk proof and the element proof within the chunk. For instance, in Figure 3, the integrity of leaf 7 can be checked with hashes h_8, h_i, h_d, h_f, and the root hash in the block header.

[1]For a 0-indexed array, parent$(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$.

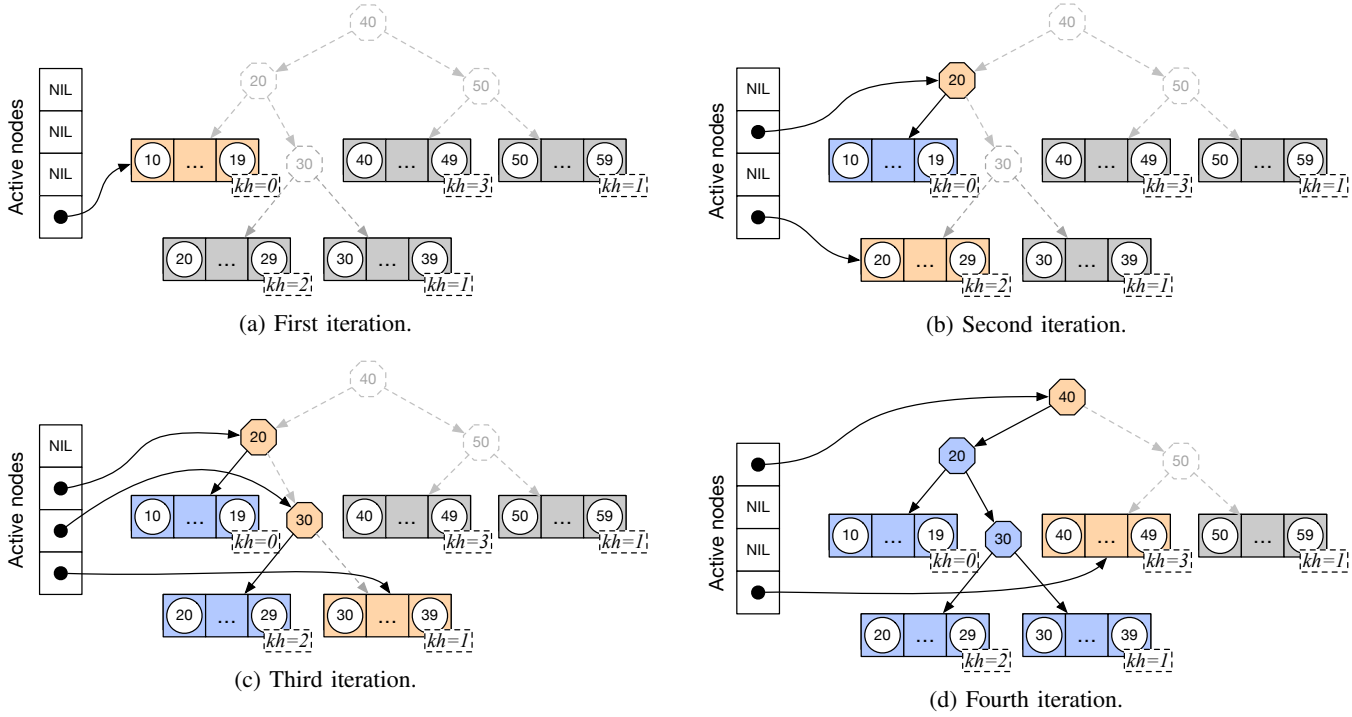
D. B+AVL tree rebuild

To rebuild the tree, a peer must request all chunks, in any order, and validate them individually. To do that, the key and value arrays are deserialized, and the in-chunk tree is recomputed by the recovering peer. Then, using the computed chunk root, the chunk proof provided by a correct peer will allow the recovering peer to ensure its validity. For instance, in Figure 3, chunk $\#1$, with keys $5..8$, can be proved with hashes h_d and h_f. Once all chunks are deserialized, they are sorted, and the inner tree structure can be rebuilt.

Since the inner structure contains only metadata, specifically, hashes and key copies, and no actual data values, and given that all data chunks have been previously verified, the integrity of the overall rebuilt tree is guaranteed. Consequently, there is no possibility that the reconstructed tree will be invalid. To ensure the same tree structure is obtained, we must note that every inner node of the B+AVL tree contains a copy of the smallest key in the left-most chunk of its right subtree. The inner node is found at a certain height in the tree structure and keeping track of this metadata, which we will call *key-height*, in all chunks is enough to fully encode and hence rebuild the structure of the inner tree. The same approach is used in the AVL+ and AVL* trees.

In more detail, during the rebuild process, we maintain a set of nodes referred to as *active nodes*, which includes both inner and chunk nodes that are not yet part of a fully reconstructed subtree. These nodes are stored in an array indexed by their key-height values. The rebuilding process begins with the chunk containing the smallest key, which is then *activated*, and proceeds in order of increasing keys.

For each chunk i, we utilize its key-height metadata, denoted as kh_i, to create the corresponding inner node n_i. This node is *activated* by inserting it into the active nodes array



(a) First iteration.

(b) Second iteration.

(c) Third iteration.

(d) Fourth iteration.

Fig. 4: Four steps of the rebuilding procedure of a B+AVL tree. Active nodes are depicted in orange, while nodes that are part of a fully rebuilt subtree are shown in blue. The chunks (gray squares) correspond to the same chunk data structure as in Figure 3; however, for simplicity, only the key range and the corresponding key-height (kh) are shown here. For example, $kh = 3$ for the chunk whose smallest key is 40, since the inner node with key 40 resides at height 3 in the tree.

at index kh_i. All existing active nodes with $kh_j < kh_i$ are then connected to form the left subtree of n_i. Once these nodes are connected, they are considered *deactivated* and are removed from the array, as their corresponding subtrees are now complete. This guarantee holds because all chunks with smaller keys have already been processed. Before advancing to the next chunk, the current chunk is inserted as a leaf-level node (with height 0) into the active nodes array.

Figure 4 illustrates four stages of the rebuild procedure, showing the state of the tree after each iteration. In the first iteration (Figure 4a), the initial chunk does not have a corresponding key in the inner structure. Consequently, this iteration concludes by inserting the current chunk as a leaf-level node (height 0) into the active nodes array.

In the second iteration (Figure 4b), the chunk has a key-height value of 2. Accordingly, an inner node is created and placed at height 2 in the active nodes array. The only active node with a height less than 2 is the chunk from the previous iteration; this node is therefore connected as the left subtree of the newly created inner node. As in all iterations, before proceeding to the next step, the current chunk is inserted into the active nodes array as a leaf-level node at height 0.

In the third iteration (Figure 4c), the chunk has a key-height value of 1. An inner node is therefore created and placed at height 1 in the active nodes array. The only active node with a height less than 1 is the chunk from the second iteration, which is connected as the left subtree of the newly created inner node. Again, the current chunk is then inserted into the active nodes array as a leaf-level node at height 0. At this stage, the active nodes array contains three entries.

In the fourth iteration (Figure 4d), the chunk has a key-height value of 3. A new inner node is therefore created and placed at height 3 in the active nodes array. All active nodes with a height less than 3 are connected to form the left subtree of this newly created inner node. Like before, the current chunk is then inserted into the active nodes array at height 0. This procedure is repeated for each subsequent chunk, and after processing the final chunk, all remaining active nodes constitute the path from the root to the rightmost chunk. These nodes are then linked together, completing the tree-rebuilding algorithm.

V. Evaluation

In this section, we provide technical details about the data structures implementation and the experimental setup. We then compare B+AVL trees to competing approaches considering performance, proof size, space efficiency, and state synchronization. We conclude with a summary of our findings.

A. Implementation and setup

Our Go implementations are based on Tendermint's AVL+ tree [4], retaining only essential features. We use Amino (a Proto3 subset) for serialization. We evaluate four structures: B+AVL, AVL*, B+tree, and a Baseline (AVL+), where chunks

are filled with serialized leaves until the target size, possibly spanning subtrees and lacking proof validation.

We evaluate the data structures in both standalone and distributed scenarios. The standalone tests focus on insertion/search performance, proof sizes, and space use. In the distributed setting, we assess state transfer costs and resilience to Byzantine attacks. A recovering peer uses a scheme like the one used in AVL* [9] to track chunk indices and detect when the full tree is retrieved. We simulate an attack where a malicious peer corrupts random chunks. Our benchmarks compare B+AVL to the Baseline tree, which must refetch the entire state upon attack, highlighting B+AVL's efficiency under such conditions.

The standalone experimental setup is composed of a machine equipped with two AMD EPYC 7282 16-Core processors, 32 GB of RAM, running Linux Ubuntu 18.04.6, and Go version 1.20.2. The distributed experiments were conducted in CloudLab [21], where the setup was configured with 3 to 12 server machines (active peers) and 1 client machine (recovering/new peer), all interconnected via a 1-Gbps switched network. We emulated a WAN environment by incorporating latencies observed in AWS regions across Europe. The machines are equipped with eight 64-bit ARMv8 (Atlas/A57) processors, 64 GB of RAM, running Linux Ubuntu 20.04, and Go version 1.13.8.

B. Performance

Our initial experiments assess the performance of constructing various trees through incremental insertions and searches within AVL* trees, Merkle B+ trees, B+AVL trees and the Baseline tree. We compare these techniques using chunks of 256 elements while varying the overall tree size. In all experiments, each element is 512 bytes in size, with 4-byte keys.

Figure 5 (top) shows the time required by the four techniques to construct trees of five different sizes, using the same sequence of elements from a shared dataset. For smaller trees (1K and 10K elements), the B+AVL tree performs slightly worse than the others, followed by the B+tree, due to B+AVL's need to recompute all hashes in a chunk after each insertion. While optimizing hash computation could improve this, it would increase complexity. As tree size grows (100K and 1M elements), performance differences between techniques diminish.

Figure 5 (bottom) shows the search performance of the four techniques when querying 10% of the elements in trees of varying sizes. All techniques use the same key sequence. For small trees, the Baseline tree performs best, while for larger trees, the B+AVL tree outperforms the others. This is due to its contiguous memory layout within chunks, which improves cache utilization by minimizing cache misses during search operations. Specifically, using the Linux `perf` tool to collect hardware performance counters, we observed that the B+AVL tree achieved the lowest cache miss rate of 26.7%, followed by the B+tree that exhibited a cache miss rate of 29.0%. The

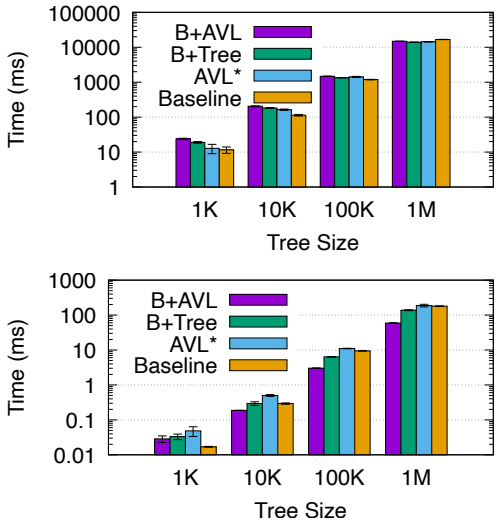


Fig. 5: Performance of insert (top) and search (bottom) operations for all four data structures, using trees with 512-byte elements and 256-element chunks, across varying tree sizes. The y-axis (log scale) shows the average time to insert all elements or search for 10% of them. Whiskers indicate standard deviation.

AVL* tree showed a rate of 30.6%, and the Baseline tree had the highest rate, at 33.1%.

C. Proof size

Figure 6 presents the distribution of proof sizes for trees with 1 million elements while varying the chunk sizes. The proof size for each element was measured after all elements were inserted. The Merkle B+tree (bottom left) exhibits a linear increase in proof size (see also Figure 2 (b)). This increase occurs because proving a value requires the hashes of all siblings for each node. Consequently, as the chunk size grows, the number of required hashes also increases, a significant drawback for B+trees.

In contrast, B+AVL, AVL*, and the Baseline tree exhibit much smaller proof sizes that remain unaffected by chunk size. Additionally, the B+AVL tree demonstrates a desirable feature of reduced variance in proof size as the chunk size increases. AVL trees maintain balance by ensuring the height difference between children of a node is at most one, while the length of a proof is directly proportional to the depth of the leaf nodes, which presents some variance. In contrast, B+AVL trees build subtrees within chunks to have a maximum depth difference of only one. This design results in a smaller variance in proof length when considering individual chunks. As the chunk size increases, this balanced structure exerts a greater influence on the overall proof length, providing the B+AVL tree with a distinct advantage. This advantage is also perceived in the cumulative distribution functions shown in the bottom right figure, where the B+AVL tree demonstrates more consistent and shorter proof lengths compared to the AVL* and Baseline trees.

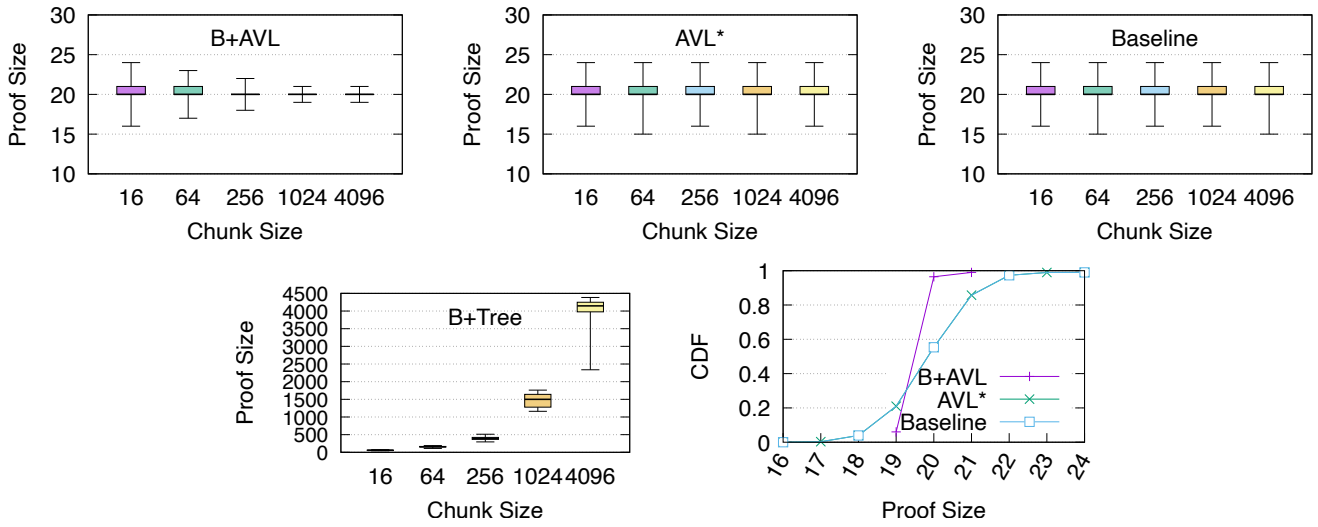


Fig. 6: Proof sizes (boxplots) were measured on a tree with 1 million elements after all insertions, varying the chunk sizes. Whiskers represent the minimum and maximum values. The CDF illustrates the distribution of proof sizes for the B+AVL, AVL*, and Baseline trees with a chunk size of 1024, revealing that B+AVL proofs have lower variance.

D. Space efficiency

In the next set of experiments, we evaluate the space efficiency of all techniques. We define space efficiency as the ratio of the actual number of chunks for the tree to the minimal number required in an ideal scenario. For example, a space efficiency of 2 indicates that there are twice as many chunks as would be needed in the optimal case. The worst-case scenario for the B+tree and B+AVL trees is when all chunks are half full [22], leading to space efficiency of 2. The Baseline tree (not depicted in the plots) offers the best space efficiency, equals to 1. Therefore, we will focus our discussion on the remaining three trees.

In Figure 7, the two plots at the top illustrate the space efficiency after each element is inserted. The x-axis represents the number of elements in the tree, while the y-axis depicts the space efficiency. On the left, the chunk size is set to 16, and on the right, it is set to 4096. We observe that for small chunk sizes, all three trees exhibit stable space efficiency as the tree grows, with the B+AVL and B+tree showing the best results. However, with larger chunk sizes, all trees display more unstable space efficiency as the tree grows, especially the B+AVL and B+tree.

While the average space efficiency is similar across all trees with chunk size 4096 (see Table II), the AVL* maintains a more consistent space efficiency compared to the B+AVL and B+tree, which exhibit greater fluctuations as elements are inserted. This behavior occurs because B+trees tend to fill all chunks before increasing their heights, given that elements are uniformly distributed for insertion. When chunks are nearly full (i.e., space efficiency close to one), subsequent inserts tend to cause splits, reducing space efficiency. As most chunks split (i.e., space efficiency approaching two), they become half full and start to fill up again. This issue does not arise with the AVL*, since the tree height is not as closely linked to the number of chunks.

This phenomenon is also observable in the two cumulative distribution functions at the bottom of Figure 7, where we measure the chunk occupancy distribution at three different points during element insertion with chunks of size 4096. The AVL* (bottom left) displays a more uniform chunk occupancy distribution, ranging from half full to full at all three points. In contrast, the B+AVL (bottom right) has varying distributions: with 300K elements, as the space efficiency approaches 2 (see the plot on top), the distribution indicates that most chunks are nearly half full. However, with 250K and 500K elements, where space efficiency approaches the optimal, most chunks are nearly full.

E. State synchronization

In the distributed state synchronization experiments, we compare B+AVL only to the Baseline tree, as it reflects the widely used approach in practice (e.g., Tendermint's AVL+) despite lacking self-verifiable chunks and requiring full re-transfer after attacks. We exclude AVL* and B+tree because they are either standalone structures not used for state synchronization (B+tree) or employ similar chunk verification to B+AVL (AVL*), thus offering comparable resilience but without B+AVL's space and proof efficiency.

Figure 8 compares the B+AVL with the Baseline tree, which builds chunks by simply filling them as it traverses the tree, resulting in non-self-verifiable chunks. The plots illustrate results from experiments conducted in an emulated wide-area network with a tree containing 100K elements. The plots on the left show state synchronization with only honest peers, while those on the right include one Byzantine. The top graphs display the time (in milliseconds) required to transfer the entire

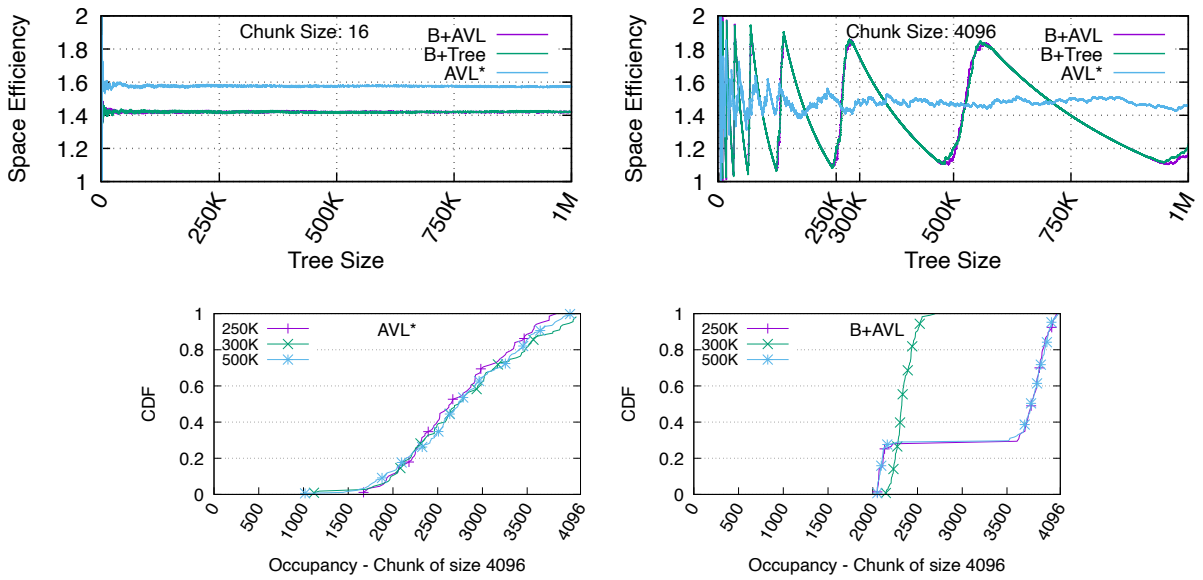


Fig. 7: Space efficiency after inserting each element, with 16 (top left) and 4096 (top right) chunks. The x-axis shows the number of elements in the tree, and the y-axis represents space efficiency. Cumulative distribution functions for chunk occupancy distribution at three different points during element insertion with chunks of size 4096, for AVL* (bottom left) and B+AVL (bottom right).

	Chunk size				
Data structure	16	64	256	1024	4096
AVL* tree	1.58 ($\pm$0.01)	1.56 ($\pm$0.01)	1.50 ($\pm$0.02)	1.51 ($\pm$0.02)	1.48 ($\pm$0.04)
Merkle B+tree	1.42 ($\pm$0.00)	1.43 ($\pm$0.00)	1.43 ($\pm$0.06)	1.41 ($\pm$0.17)	1.40 ($\pm$0.23)
B+AVL tree	1.42 ($\pm$0.00)	1.44 ($\pm$0.01)	1.44 ($\pm$0.03)	1.43 ($\pm$0.13)	1.41 ($\pm$0.24)

TABLE II: Average (and standard deviation) space efficiency of the three data structures with varying chunk sizes.

tree with varying chunk sizes and 9 peers serving chunks. The middle plots illustrate the impact of varying the number of peers serving chunks with a chunk size of 256. The bottom graphs depict the number of chunks sent by each technique as the chunk size varies with 9 peers serving chunks.

We observe that in the absence of attacks, the B+AVL performs slightly better than the Baseline, despite sending more chunks. This advantage is attributed to the fact that the serialization, communication, and rebuild times of the Baseline are higher than in B+AVL (Figure 9 left). Serialization in the Baseline tree needs traversing the entire tree to fill up the chunks. In the B+AVL tree, chunks are directly accessible, allowing for more efficient serialization, as the memory in each B+AVL chunk is contiguous. Consequently, serialization can be performed with a single call per chunk, compared to the Baseline, which requires a call for each leaf. Additionally, contiguous memory in the B+AVL chunks improves cache performance, particularly for keys, as multiple keys can fit into the same cache line. This efficient memory usage further speeds up the serialization process (see also Section III-A). Rebuilding is also faster in the B+AVL tree due to its streamlined structure. In the B+AVL, there is only one inner node per chunk, whereas in the Baseline tree, there is one inner node for each leaf node. This means the Baseline has significantly more inner nodes to rebuild, slowing down the process. Deserialization instead takes longer in the B+AVL tree since the hash values of the logic tree within the chunk have to be computed, in order to validate the individual chunks. In contrast, the Baseline tree only requires the deserialization of individual leaves, making this process more straightforward and less resource-intensive.

Under Byzantine attacks, B+AVL trees enable peers to verify data integrity with minimal computational overhead. Their small proofs allow rapid verification and the advantage of self-verifiable chunks becomes clear. Honest peers using B+AVL trees can quickly detect invalid data, blacklist the Byzantine peer, and refetch corrupted chunks from another source. In contrast, the Baseline tree can only detect corruption after downloading the entire tree, leading to much longer synchronization times and increased data transfer. Figure 8 right shows that the Baseline tree can take up to $4\times$ longer and require twice as many chunks as B+AVL.

The performance gap worsens for the Baseline tree as the number of potential Byzantine peers increases, as shown in Figure 9 right, which illustrates the impact of varying the number of Byzantine peers up to four. With more Byzantine peers, the Baseline tree becomes increasingly inefficient, lead-

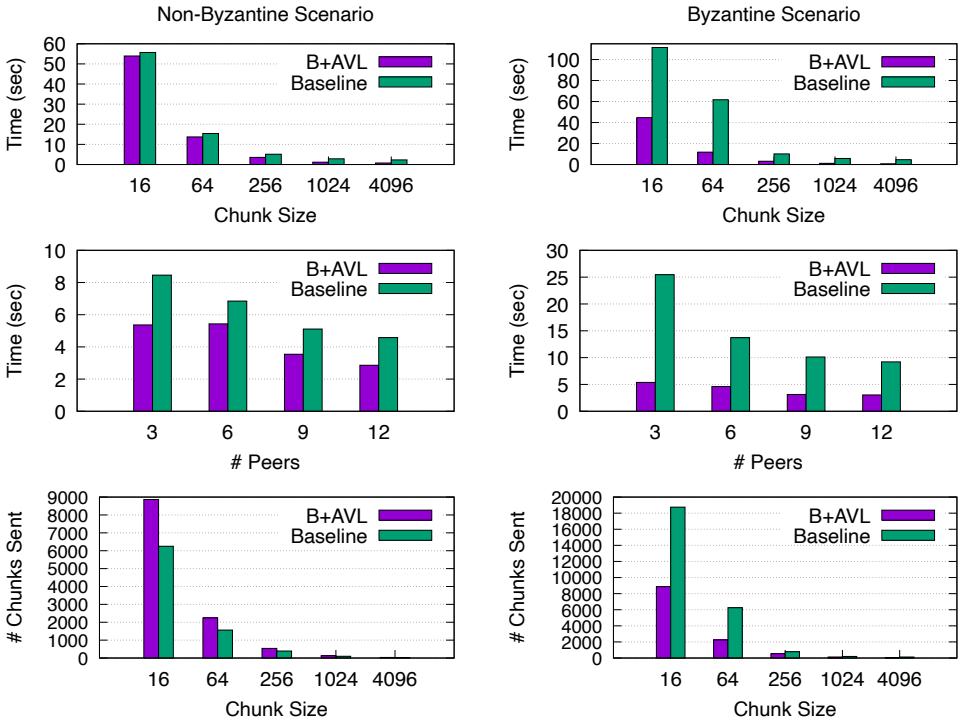


Fig. 8: State synchronization time of B+AVL and Baseline trees without attacks (left column) and with 1 Byzantine peer (right column) for a tree of size 100K: performance versus chunk size with 9 peers (top); performance versus number of peers with 256 chunks (middle); and the number of chunks sent during synchronization as we vary chunk size with 9 peers (bottom).

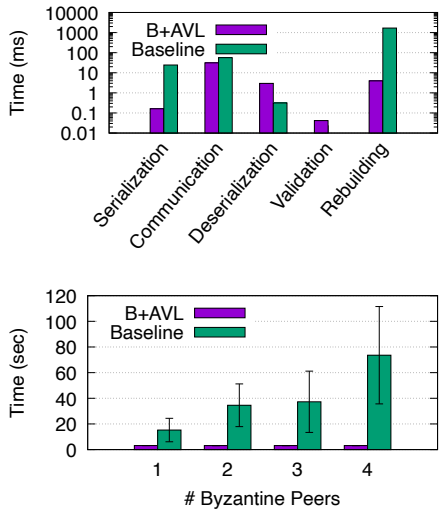


Fig. 9: Top: performance breakdown showing time (ms, log scale) for each phase of state synchronization with a 100K-element tree and 256-element chunks. Bottom: B+AVL and Baseline tree performance under Byzantine attack, with 100K elements and 256-element chunks, varying the number of Byzantine peers.

ing to longer synchronization times. In contrast, the B+AVL tree maintains strong performance, demonstrating its superior resilience and efficiency in managing Byzantine attacks. Interestingly, in some cases, the B+AVL tree completes state synchronization faster in Byzantine scenarios than in non-Byzantine ones (e.g., Figure 8 top with a chunk size of 16). This can happen if the Byzantine peer is in a distant region on the network. Once blacklisted, the peer fetches chunks only from closer peers, speeding up synchronization. This underscores the B+AVL tree's ability to effectively manage Byzantine peers and optimize peer selection, further improving synchronization times in certain conditions.

F. Summary of results

Table III presents a summary of our findings. It highlights the key advantages of B+AVL trees and provides a concise summary of the results across various metrics, comparing the performance of all techniques normalized to the Baseline tree, with smaller values representing better performance.

VI. Related work

Improving the speed of state synchronization without the need to process the entire transaction log has been a central focus for many Byzantine fault-tolerant state machine replication (BFT SMR) systems. Most systems employ a highly efficient method that revolves around utilizing state snapshots, also known as checkpoints. These snapshots can swiftly be acquired by new or recovering peers from already operational peers, facilitating the rapid synchronization of their state to the

		Baseline tree	AVL* tree	B+tree	B+AVL tree
Search	Small tree	**1**	2.86	1.95	1.68
	Large tree	1	1.04	0.76	**0.32**
Insert	Small tree	**1**	1.10	1.64	2.05
	Large tree	1	0.86	**0.83**	0.89
Proof size	Small chunk	1	1	2.85	**0.99**
	Large chunk	1	0.99	192.20	**0.98**
Space efficiency	Small chunk	**1**	1.58	1.42	1.42
	Large chunk	**1**	1.48	1.40	1.41
No Byzantine State Synchronization	Small chunk	1	—	—	**0.97**
	Large chunk	1	—	—	**0.32**
	Few peers	1	—	—	**0.63**
	Many peers	1	—	—	**0.62**
One Byzantine State Synchronization	Small chunk	1	—	—	**0.40**
	Large chunk	1	—	—	**0.15**
	Few peers	1	—	—	**0.21**
	Many peers	1	—	—	**0.33**
Many Byzantine State Synchronization	1 byzantine peer	1	—	—	**0.20**
	4 byzantine peers	1	—	—	**0.04**

TABLE III: Summary of the results for various metrics evaluated, comparing the performance of all techniques normalized to the Baseline tree. The values represent the average performance observed across multiple experiments. Smaller values indicate better performance. Best results shown in boldface.

current state of the system. For instance, PBFT [8] proposed using a Merkle tree for its checkpoints, while Upright [23] and BFT-SMaRt [6] find Merkle trees and copy-on-write semantics too invasive for application developers. Upright suggests three simple state transfer methods, whereas BFT-SMaRt offers a detailed Collaborative State Transfer protocol [7], where the state is downloaded from a single peer and verified against hashes from other peers.

Today, as state sizes in blockchain systems continue to expand, synchronizing becomes more costly. Consequently, Merkle trees have garnered considerable attention. Unlike traditional SMR systems, which often deem Merkle trees too expensive or only utilize them periodically for synchronization, many blockchain systems already leverage Merkle-based data structures for state storage. These trees serve a vital role in enabling light clients, empowering them to query specific leaves efficiently and verify their integrity without the need to download the entire state or transaction history.

In Geth [24], Ethereum's Go implementation, state synchronization involves requesting individual nodes of the tree. Peers are unable to anticipate the duration of this synchronization process as they lack knowledge of the total number of nodes [25]. Since the Merkle tree is a component of the consensus rules, with Merkle roots stored in block headers, peers can authenticate the accuracy of received nodes. However, the performance of both requesting and providing peers is adversely affected by the small size of nodes (less than 1 KB), their randomized distribution in the database, and the substantial state size (tens of GBs), when nodes are requested individually.

Batching nodes of a tree could be a potential solution, yet it poses challenges in ensuring verifiability while safeguarding honest peers against malicious attacks. For instance, in OpenEthereum [26], snapshots are periodically created by serializing the entire state and segmenting it into sizeable chunks, each associated with hashes published in a manifest file. However, since the manifest is not integrated into the consensus mechanism, verifying the correctness of a chunk before downloading all chunks is not feasible. Thus, the successful completion of state synchronization hinges on obtaining a correct manifest, necessitating strong assumptions like trusting a specific peer or presuming a majority of connected peers to be honest. In Tendermint IAVL+ snapshots, tree nodes are serialized and grouped into fixed-size chunks [4]. However, because Tendermint's block header lacks snapshot hashes, peers cannot verify chunks incrementally, underscoring the need for a tree structure with built-in chunking support.

Blockchain scalability improvements can also be classified into two main categories: on-chain and off-chain. On-chain solutions enhance the blockchain infrastructure itself, such as efficient consensus algorithms [27] [28] and sharding [29] [30]. Off-chain solutions, or "Layer-2" (L2) techniques, offload computation and storage to reduce the load on the main blockchain (L1). Examples include State Channels [31], and rollups [32]. zkSync Lite [33], a zero-knowledge rollup, achieves a performance nearly 6 times faster than Ethereum.

In the context of optimizing rollup mechanisms, Sparse Merkle trees have been adopted. In [34] the authors study the sparse Merkle tree algorithms presented in zkSync Lite, and propose an efficient batch update algorithm to calculate a new root hash given a list of account (leaf) operations. Using the construction in zkSync Lite as a benchmark, their algorithm improves the account update time from $O(log_n)$ to $O(1)$ and reduces the batch update cost by half using a one-pass traversal. Advances in cryptography have also introduced generalizations of Merkle trees called accumulators, enabling $O(1)$ proofs of set-membership and state-less blockchain clients [35].

VII. Conclusion

The paper addresses the challenge of state transfer in blockchain systems, focusing on the process of recovering peers catching up with operational ones to maintain a consistent state. While downloading all transactions to rebuild state is inefficient, blockchain systems like Cosmos and Ethereum use periodic snapshots for recovery, supplemented by Merkle tree structures for validation. To expedite this process, chunks of the state tree can be downloaded concurrently from multiple peers. Existing solutions like AVL* trees and Merkle B+trees improve efficiency by embedding state subtrees or leaves into chunks, but they have their limitations.

This paper introduces B+AVL trees, a new chunk-based data structure that combines the balance and validation benefits of AVL trees and Merkle hashes, while being more space-efficient and simpler than AVL* trees, in which rotations can move leaf nodes across chunks, requiring complex algorithms to maintain those chunks. Additionally, B+AVL trees offer more compact validation proofs than Merkle B+trees. The paper thoroughly evaluates B+AVL trees and shows experimentally their advantages when compared to AVL* trees and Merkle B+trees.

VIII. Acknowledgements

This work was partially supported by the Hasler Foundation (project number 2024-08-07-133).

References

[1] "Cosmos network," https://cosmos.network/.

[2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[3] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.

[4] "Adr 053: State sync prototype," https://github.com/tendermint/-tendermint/blob/master/docs/architecture/adr-053-state-sync-prototype.md.

[5] "Cosmos sdk," https://github.com/cosmos/cosmos-sdk.

[6] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[7] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 169–180.

[8] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[9] E. Fynn, E. Buchman, Z. Milosevic, R. Soulé, and F. Pedone, "Robust and fast blockchain state synchronization," in *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, ser. LIPIcs, E. Hillel, R. Palmieri, and E. Rivière, Eds., vol. 253. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 8:1–8:22.

[10] E. Fynn, "Scaling blockchains," Ph.D. dissertation, Università della Svizzera Italiana (USI), 2021.

[11] R. Friedman and R. van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocols," in *Proceedings of the 6th International Symposium on High Performance Distributed Computing, HPDC '97, Portland, OR, USA, August 5-8, 1997*. IEEE Computer Society, 1997, pp. 233–242.

[12] I. Eyal and E. G. Sirer, "Majority is not enough: bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[13] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," vol. 35, no. 2, pp. 288–323, 1988.

[14] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018. [Online]. Available: http://arxiv.org/abs/1807.04938

[15] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. [Online]. Available: https://doi.org/10.1145/571637.571640

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 45–58. [Online]. Available: https://doi.org/10.1145/1294261.1294267

[17] M. Szydlo, "Merkle tree traversal in log space and time," in *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004, pp. 541–554.

[18] "The interblockchain communication protocol," https://github.com/cosmos/ics/blob/master/spec.pdf.

[19] "Official go implementation of the ethereum protocol," https://github.com/ethereum/go-ethereum.

[20] D. Knuth, *The Art Of Computer Programming, vol. 3: Sorting And Searching*. Addison-Wesley, 1973.

[21] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[22] D. Comer, "Ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, p. 121–137, jun 1979. [Online]. Available: https://doi.org/10.1145/356770.356776

[23] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 277–290.

[24] "Geth v1.9.0: Six months distilled," https://blog.ethereum.org/2019/07/10/geth-v1-9-0/.

[25] "Go ethereum faq," https://geth.ethereum.org/docs/faq.

[26] "Openethereum warpsync," https://openethereum.github.io/wiki/WarpSync.

[27] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds. Cham: Springer International Publishing, 2017, pp. 357–388.

[28] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless bft consensus through metastability," 2020. [Online]. Available: https://arxiv.org/abs/1906.08936

[29] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 17–30. [Online]. Available: https://doi.org/10.1145/2976749.2978389

[30] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 931–948. [Online]. Available: https://doi.org/10.1145/3243734.3243853

[31] L. D. Negka and G. P. Spathoulas, "Blockchain state channels: A state of the art," *IEEE Access*, vol. 9, pp. 160 277–160 298, 2021.

[32] V. Buterin, "An incomplete guide to rollups," https://vitalik.eth.limo/general/2021/01/05/rollup.html.

[33] M. Labs, "Zksync: scaling and privacy engine for ethereum," https://github.com/matter-labs/zksync.

[34] B. Ma, V. N. Pathak, L. Liu, and S. Ruj, "One-phase batch update on sparse merkle trees for rollups," 2023. [Online]. Available: https://arxiv.org/abs/2310.13328

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