IMPROVED AUTHENTICATED DATA STRUCTURES FOR BLOCKCHAIN SYNCHRONIZATION

BY

XIAOYAO QIAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Andrew Miller

ABSTRACT

One of the most important components in a public blockchain like Bitcoin and Ethereum is the authenticated data structure that keeps track of all block data, transactions, and the world state (account balance, smart contract states, etc.) Thanks to authenticated data structures, lightweight nodes only need to store authentication information and can delegate queries to those nodes with a full replica of data and the authenticated data structure. The lightweight nodes can trust the query results after verifying against the authentication information. It is also critical to have enough nodes in the network that are equipped with the authenticated data structure to ensure scalability and availability, which is especially important for public blockchains. Therefore, every public blockchain highly encourages users to download the authenticated data structure as the first step.

Fetching all elements from the entire authenticated data structure is a novel query type that has not gathered attention in the past. We describe this new emerging query type in the three-party authenticated data structure (ADS). We improve the design and implementation of the authenticated data structure so that the new query type is well-supported. We specifically apply the improvements to the Ethereum blockchain network. With our proposed ADS system in Ethereum, we improve Ethereum state synchronization performance by 216 times.

To Yuanyuan, for her patience and companionship.
To my parents, for their love and support.
To Tom, hey man, I've made it.

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Andrew Miller, who provides invaluable support, inspirations, and guidance for my research during my study. I am also very grateful to him for introducing me into the fascinating blockchain world.

Many thanks to the Consensys INFURA folks for providing insights and suggestions.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Blockchain projects such as Bitcoin[1] and Ethereum[2] seek to provide decentralized, cryptographically-secure value transfer networks. Since the broad adoption of Ethereum in value transfers, token sales[3], and even crypto-gaming, its network size and data volume have been increasing vastly. Last year, Ethereum daily transaction volume ranges from 250,000 to 1,250,000, and there are 60,000 to 120,000 new accounts created every day. One side-effect of the rising data volume is that it now takes weeks to synchronize the blockchain from scratch [4, 5, 6].

When a fresh new node that has never synchronized with the Ethereum network wants to join, it would have to download all the blocks, validate the blocks (proof-of-work, total difficulty, transactions, etc.), and replay all transactions in the history to compute the latest global state. When this process is finished, the client is confident about the integrity of the blockchain data because it has performed full validation on blocks, and replay all transactions locally. The client is thus ready to process new queries from the network to update on its state. When new blocks are mined, they are broadcasted to the network. Receivers synchronize with the network by listening for these new blocks, validating them, and applying the incremental changes to their own states. In short, synchronization in blockchains typically involves downloading and fully validating all history blocks from the network, and processing all transactions to compute the latest state where new transactions would base on. However, as the blockchain grows longer, the time and computing resources required for the synchronization grows linearly. The community has proposed different synchronization modes to speed up the process of acquiring the latest global state. However, currently on the Ethereum mainnet, even the fastest synchronization typically takes around a week of time. This has been a long-standing issue that has not seen a good resolution yet. Geth [7], the official Ethereum Golang client, proposed the "Fast Sync" mode in late 2015 in order to improve node synchronization performance by directly downloading the latest state. Parity[8], another popular Ethereum client written in Rust, has proposed "Warp Sync" mode at the end of 2016 aiming to utilize state snapshots to ease sync performance. These two modes worked fine in 2016. Geth Fast Sync was able to finish a synchronization from scratch at the time (with 837869 blocks and 10.2 million state entries) in 31-32 minutes when a normal sync took over 4 hours [9]. Parity Warp Sync finished synchronization around 1 hour when it was released [10]. However, as shown in Figure 1.1, the average block size, the transaction volume, and the number of unique address are growing rapidly since mid 2016.

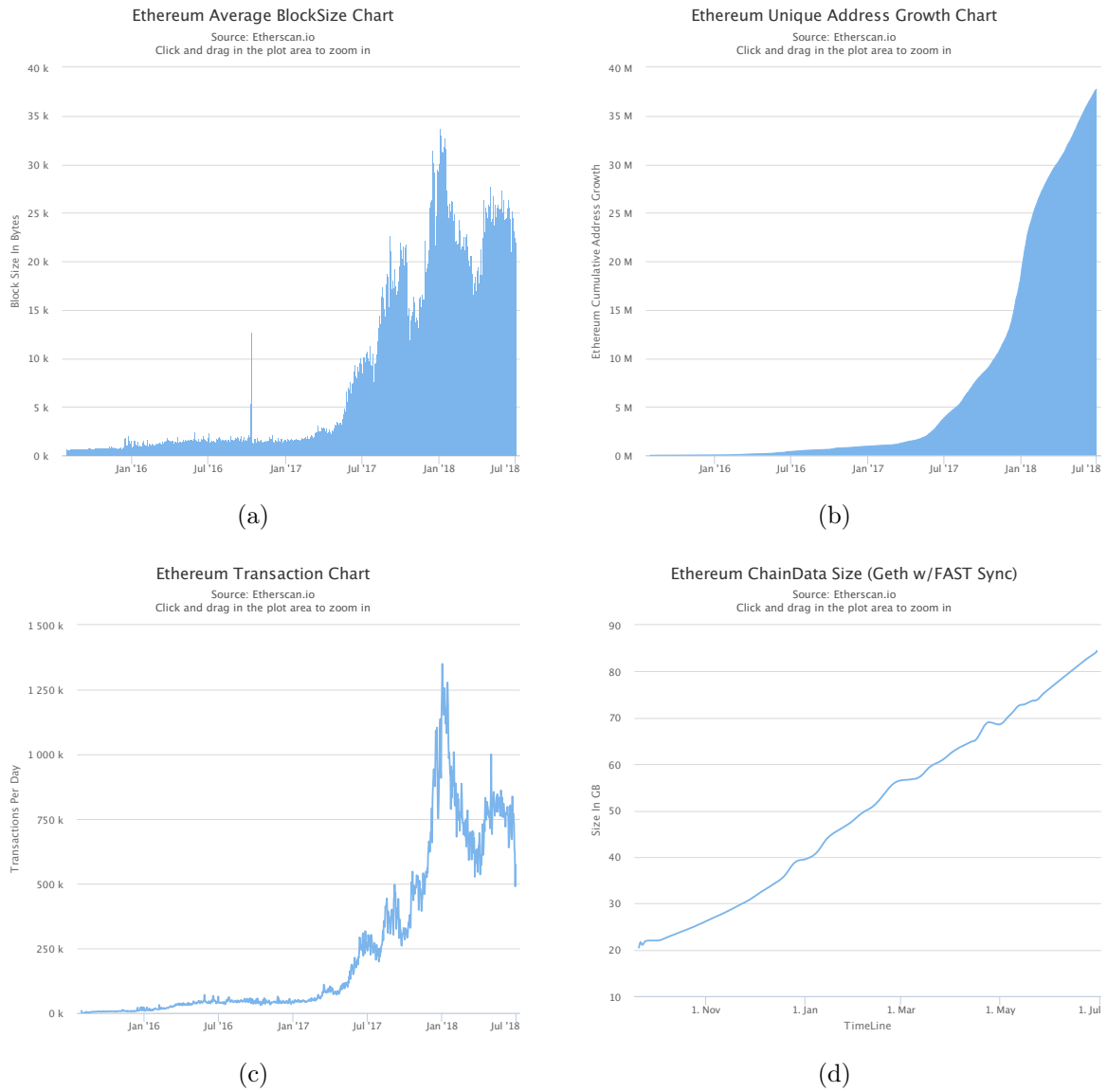


Figure 1.1: **Ethereum Stats Chart.** These 4 figures show from Jan 2016 to Jan 2018 Ethereum's: (a) the growth of average block size; (b) the growth of unique addresses; (c) the transaction volume growth; and, (d) Geth Fast Sync database size.
(Source: etherscan.io)

Correspondingly, block data and state data in Ethereum grow and these two modes are no longer efficient since 2017 [11]. We will further discuss and compare different sync modes in Ethereum in Chapter 3.

Full synchronization is not always necessary, though. A client can also choose to only download the block headers which contain authentication information about transactions and states, and delegate its queries/requests, such as balance checking, to other peers that

do have the latest state, known as the Light Ethereum Subprotocol (LES) [12] or Light Sync mode. LES is made possible by the authenticated data structure (ADS)[13] that is used to store the Ethereum state. An authenticated data structure is a model of computation where untrusted parties answer queries on a data structure on behalf of a trusted source and provide a proof of the validity of the answer to the user [13]. On receiving a delegated query, the remote peer processes the request and returns the results and proof to the caller. The proof is essential here because the peer could be a malicious node and could deceive callers by sending them fake results if they do not provide proofs.

Delegating computations over to an untrusted peer is a well-researched topic in authenticated data structures [14, 15, 16, 17, 18]. Queries used in LES are mostly point query (read-/write) to the data such as finding or modifying the balance of an account. Synchronization can be analogous to a full scan of the data in the ADS. The cryptocurrency communities are aware of the importance of efficiently scanning the entire ADS, but have not yet seen a good solution to achieve this. In this paper, we propose a new authenticated data structure system that is able to handle full scan queries efficiently. In particular, we implement two new synchronization modes (Turbo Sync and Checkpoint Sync) which would maximize sync performance for both normal end users who are only interested in the latest state, as well as analytic users who would like all history states. Turbo Sync exchanges processing power for bandwidth usage. It allows nodes to acquire the latest Ethereum state by downloading a snapshot instead of replaying all history transactions. It leverages a state cache that always store the latest state entries to achieve fast snapshot generation. Checkpoint Sync leverages Turbo Sync to download state snapshots at evenly distributed checkpoints so that multiple concurrent workers can replay transactions in between checkpoint states in parallel.

Our contributions in this work can be summarized as:

1. We identify blockchain synchronization, a critical bottleneck facing cryptocurrencies today, as fetching all elements from an authenticated data structure [13]. We then propose new architectural designs for the authenticated data structure models used in Ethereum, which results in two new sync modes that maximize synchronization performances.

2. Empirical benchmarking with a prototype implementation shows that our proposed sync modes speed up Ethereum state sync by over 216 times.

CHAPTER 2: BACKGROUND AND RELATED WORK

In this chapter, we will introduce some background knowledge about authenticated data structures and Ethereum. We will go over some implementations of ADS, and how Ethereum implements ADS.

2.1 AUTHENTICATED DATA STRUCTURE

Authenticated data structures (ADS) is a data model involving three parties: source (trusted data source), directory (untrusted peer), and user. We show, in Figure 2.1, a schematic view of a three-party authenticated data structure model.

- Source holds the original version of the data. When the data in the source is updated, it produces structure authentication information, which consists of a signed timestamped statement about the current version of the data. The source is trusted.
- Directories are untrusted peers that acquire copies of the data from the source. When the source updates the data, the directories receive updated data along with the updated structural authentication information. Directories also provide query answering service to users. On receiving a query request, directories return the answer to the query, the latest structural authentication information from the source, and an authentication proof that proves the correctness of the answer.
- Users do not hold a copy of the data. They instead issue queries to the data. They delegate the queries to directories. Unlike the source, directories are untrusted peers, therefore, the users have to be able to verify the answers from the directories with the help of the additional authentication information and proofs from the directories.

With authenticated data structure, it is possible to outsource massive amounts of data and computations on those data while remaining confident about data integrity. In most cases, directories are added to enhance system-wide scalability.

Throughout the rest of this section, we denote with m the size of the dataset maintained by the authenticated data structure.

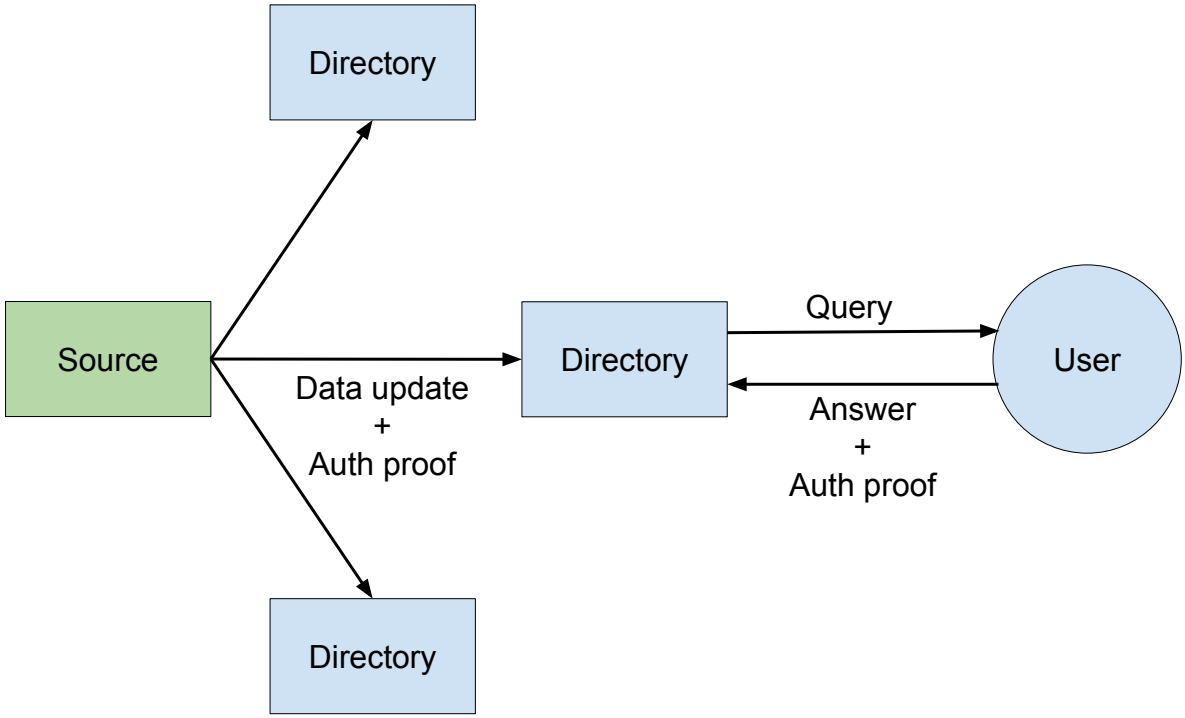


Figure 2.1: **Three-party Authenticated Data Structure Model**

Work has been done on using hash list as an authenticated data structure [19]. The hash over all elements in the list is used as the authentication info when passed around between source and directories. Hash list is inefficient. In case of data update, it has to recompute the hash by scanning through all elements. If we want to preserve data history, every list would have to be archived at the directories. In addition, a directory would have to send over the entire list to users along with the authentication proof; otherwise, the user would not be able to verify the proof. Namely, data update time complexity is $O(m)$, and it takes $O(m^2)$ space.

Merkle tree[20] structure (hash tree scheme) is commonly used to implement authenticated data structure. In a Merkle tree, data are stored at leaf nodes. Cryptographic hashes are computed, stored, and aggregated up on branch nodes, all the way to the root. Each branch node is a hash value over its children nodes. Merkle tree uses linear space and has $O(\log(m))$ proof size, query time and verification time.

Persistent authenticated dictionaries (PAD) [16] are one specific subtype of an authenticated data structure. PAD archives the entire history of updates sent by the source to the directories, and allows users to search data by keys related to any time. Red-black tree and skiplist are commonly used to implement PADs [16].

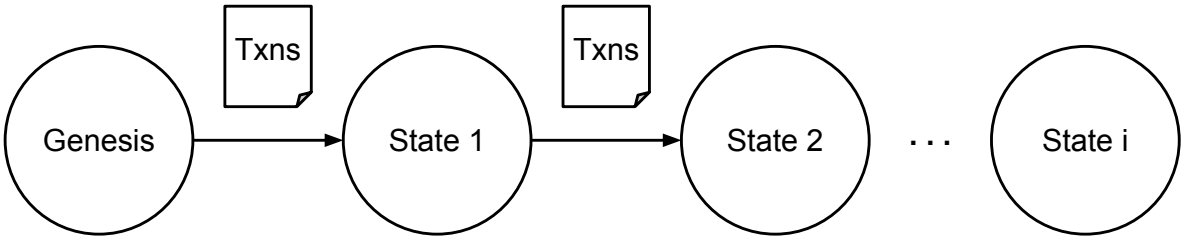


Figure 2.2: **Transaction-based State Machine**

There are also research on converting generic data structures into authenticated data structures [21, 14].

Iris [18] is a real world system that leverages authenticated data structure in a enterprise cloud-oriented file system allowing file system tenants to verify the integrity of data while performing typical file system operations.

2.2 ETHEREUM

Ethereum is a blockchain network that supports peer-to-peer value transfer and decentralized applications knowns as smart contracts. Each node in the Ethereum network can be considered as a transaction-based state machine. The state machine starts with an initial state(the Genesis). As illustrated in Figure 2.2, old states transition to new states when transactions are executed.

Data on Ethereum are logically separated. Transactions are grouped into blocks. Blocks are then propagated across the network. Each new block is chained with a hash pointer pointing to the previous block, thus forming a blockchain. The state is not propagated explicitly. The state is the product of processing transactions in each block so it is implicit and it is not disseminated in the network. Essentially, there is one new state after every block. It is not necessary for all nodes to keep all history states locally. As long as a client has the latest state, it can perform regular tasks.

Ethereum block data include housekeeping metadata such as block number, block hash, previous block hash, proof-of-work, nonce, timestamp, miner address, mining difficulty, gas

usage, and most importantly, three Merkle roots: transaction root, transaction receipt root, and state root. The three Merkle roots correspond to three off-chain data components structured in Merkle Patricia Tree (MPT) [22]: transaction trie, transaction receipt trie, and state trie. Ethereum only puts Merkle roots in the block headers because, in this way, clients can minimize the amount of data sent over the network while at the same time ensure the data integrity. The transaction trie and transaction receipt trie are MPTs composed of transactions and transaction receipts(event logs). These two types of data are immutable and persistent, i.e., transactions and transaction receipts will not change in the later blocks. The state trie is more complicated. The state trie is an MPT composed of account data (balances, nonces, smart contract code and data, etc.) Every transaction causes state change. Moreover, each account has its own storage space. The storage of each account is organized in an MPT too. The storage MPT Merkle root is stored as part of the account data in the state trie. Figure 2.3 illustrates the hierarchy of block headers, state trie Merkle root, and storage trie Merkle roots.

2.2.1 Merkle Patricia Trie

Merkle Patricia Trie (MPT) [22] is the authenticated data structure used in Ethereum that can be used to store any key-value pairs. In an MPT, nodes branch out to children with each pointer representing part of the prefix to the key. The entire path from the root to the particular data node forms a key. Furthermore, pointers from a node to its children are represented as hashes of the children nodes, hence are hash pointers.

MPTs are history independent[23]. Therefore, no matter in what order are the same set of key-value pairs inserted, the resulting tree structures are always the same with the same Merkle root. The same key-value pair is also always down at the same path. This is an important feature because we can rebuild an MPT regardless of their insertion order to validate data integrity. In Chapter 4, we will describe how we leverage this feature and use a sorted data store to generate verifiable data snapshots which significantly improves Ethereum synchronization performance. MPT provides $O(\log(n))$ efficiency for inserts, lookups, and deletes where n is the number of nodes in the trie.

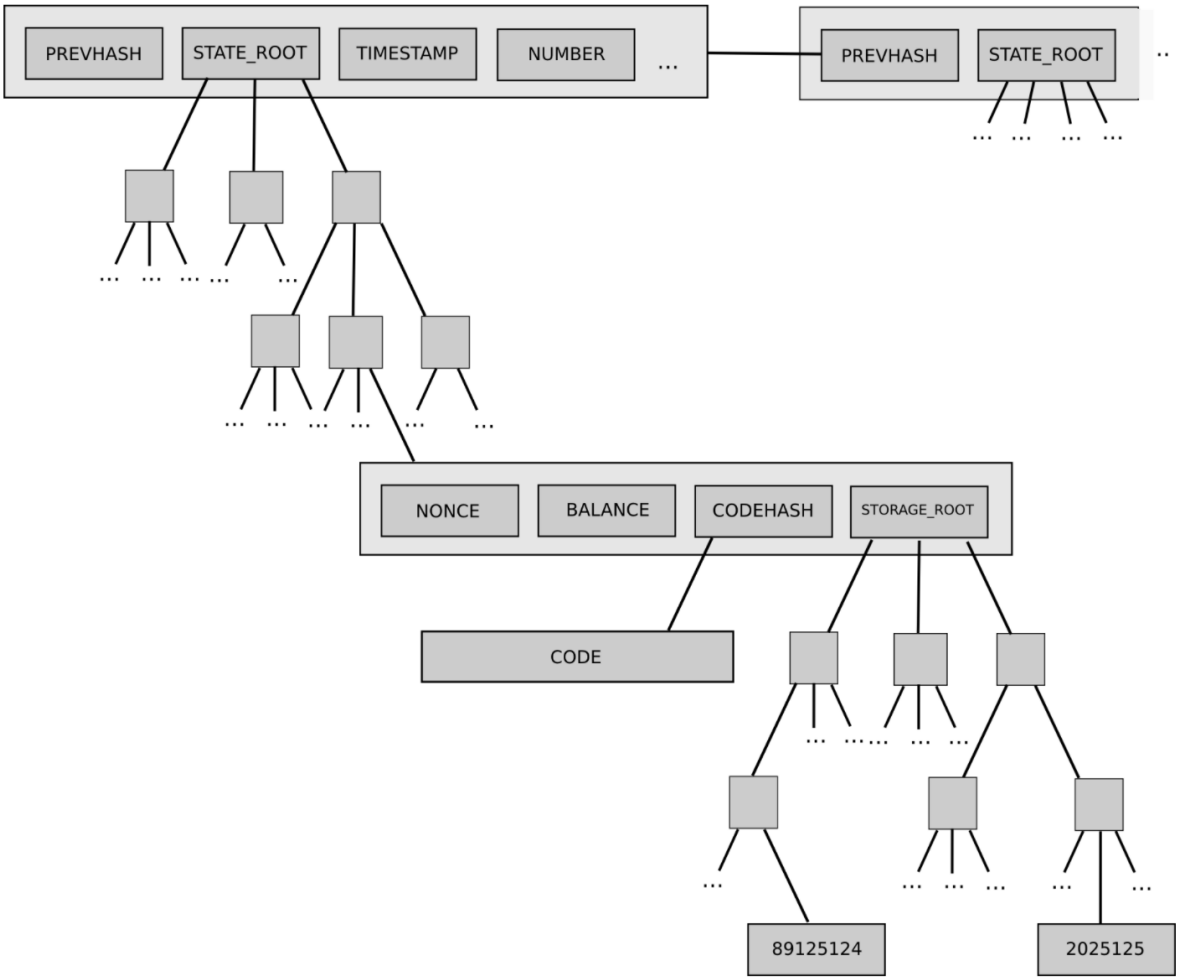


Figure 2.3: **Hierarchy of Ethereum State Trie.** State roots in block headers implicitly point to state MPTs, where a state trie entry contains a storage root that implicitly points to another storage MPT. (source: https://github.com/ethereum/wiki/wiki/Ethereum-Development-Tutorial license: Creative Commons)

2.3 STORAGE

Log Structured Merge tree(LSM)[24] based in-process data storage engines like LevelDB[25] and RocksDB[26] are used among popular Ethereum clients such as Geth and Parity. They are typically key/value store without schema constraints. The LSM-tree family organizes data in multiple layers on disk, and manage the layers in the background. LevelDB, in particular, is also the storage engine for Bitcoin core infrastructure[27]. LSM-based data stores are generally known to be fit for write-intensive scenarios, but relatively bad at read-intensive situations. Asymptotically, the time complexity of a write to the LSM tree is always $O(1)$, while a read takes amortized $O(\log(n))$ [28]. When new records are ready to be written,

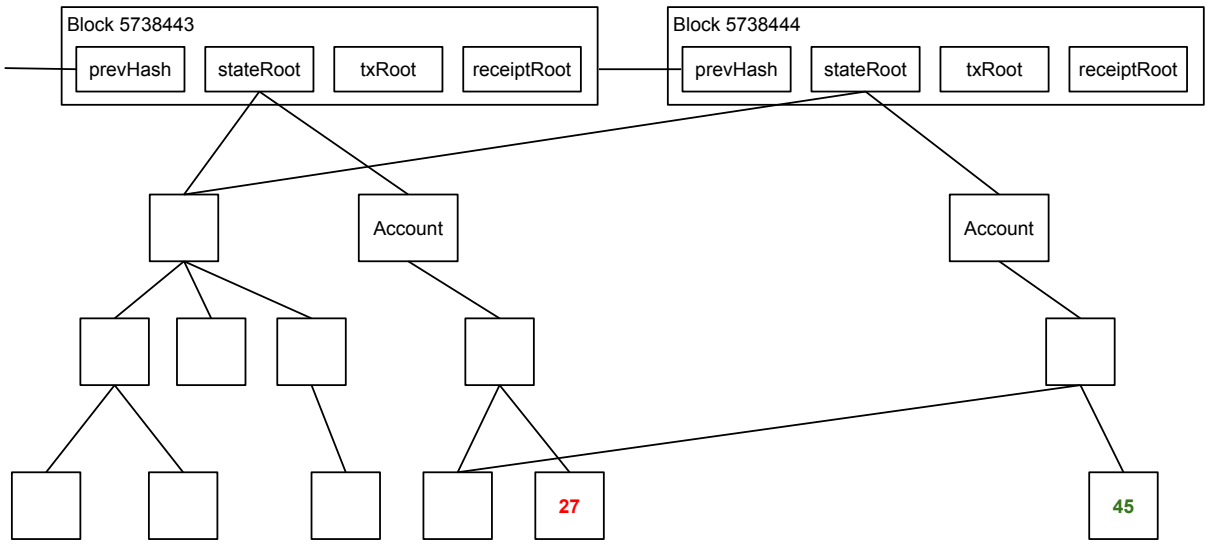


Figure 2.4: **Ethereum State Trie, an example Merkle Patricia Trie**

they are batched in memory, and appended to the disk when the size of the batch reaches a threshold. Data files are organized in layers. When the total amount of data in a layer reach a threshold, they will be compacted and merged to the data files on the next layer. With these two features, LSM trees minimizes the amount of random writes on disk.

There are limitations with these storage engines. There are not too many different data access patterns in the MPT. When one node in the MPT is to be updated or read, there is a high chance that nodes along the Merkle branch will be changed or expanded accordingly. Therefore, it is beneficial for both read and writes if the data store handles data locality well. However, Ethereum uses hashes exclusively for identifying anything. Hashes are uniformly, randomly distributed identifiers. These hashed keys are ordered lexicographically on disk, therefore data locality is lost. Point lookups associated with hashes are very expensive. This is mostly felt for the state trie, where hundreds and thousands of entries are scattered and pointing all over the disk. The way Ethereum adapts the storage of the trie structure is through indirections. Each entry is represented as a key-value pair where the key is the hash of the node, and the value is an RLP[1] [22] encoded value. If the node is a branch node, the value will be an encoded list where each position points to a different key in the database. The side-effect of such indirections is severe data read and write amplification. Reading one single key can result in 64 LevelDB reads in Geth, while writing a single key can lead to a similar number of LevelDB writes [29].

[1]RLP stands for Recursive Length Prefix, an encoding format used in Ethereum.

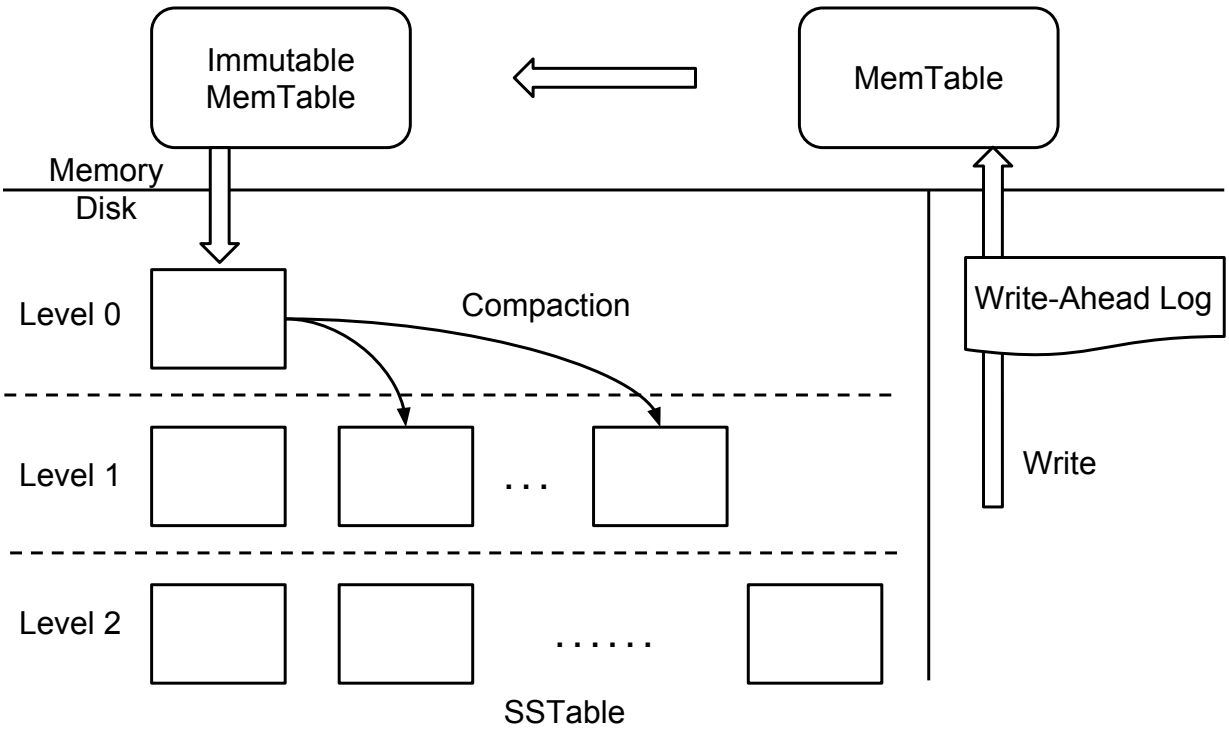


Figure 2.5: **LevelDB Write Flow.** This figure shows the write flow of LevelDB on a high level.

Due to the above defects, there are studies into designing better storage plan for Ethereum. Merkelized LSM (mLSM)[29] tree is dedicated to reduce read and write amplification in LSM-based ADS while still preserving the capability to verify data reads. The main idea behind mLSM is to maintain multiple independent tries. On each level of the LSM tree, instead of storing raw data, mLSM stores multiple immutable Merkle tries. Updates to mLSM are batched in memory and written to Level 0 as a new Merkle trie. Periodically, the tries in the i^{th} level will be compacted into a single trie in the $i + 1^{th}$ level. Turbo-Geth[30] proposes to redesign the key format of the MPT in the underlying key-value data store to capture data locality, which should reduce read amplification and make range scan faster.

CHAPTER 3: PROBLEM DEFINITION

In this chapter, we will define the blockchain-specific authenticated data structure model and the new query type that is emerging thanks to the rising popularity of cryptocurrencies. We will analyze and compare various synchronization modes in Ethereum, and eventually, pinpoint the synchronization performance bottleneck. At the end of this chapter, we will formulate our goal.

3.1 OVERVIEW

In Ethereum, each client making incremental computation requests should know the latest previous state. This is where synchronization comes into play. Synchronization in Ethereum involves downloading all history block data from the network, verifying the data, and retrieve or compute the latest state where new transactions would base on.

As a result, a minimum synchronized node must have:

- Block headers
- Transactions for each block
- The latest state

In a public blockchain network, usually, there is no direct correspondence of a "source" entity. Take Ethereum for example: every node is equally untrusted. Every node in Ethereum plays the role of a directory. Even miners who package and broadcast new blocks to the network are considered directories because they could have deceived other parties by producing blocks with bogus proof-of-work (PoW). A miner cannot be trusted, but the mined block can be trusted as long as the provided proof-of-work is verified. In this sense, in a blockchain network like Ethereum, the "source" is really reflected by the consensus of the network.

Therefore, the authenticated data structure model for public blockchains is a deviation from the ordinary cases but with the same properties. Here we define the public blockchain specific ADS model as follows:

- Source: a *logical* entity that holds the original version of the data. In the public blockchain specific setting, the canonical state (the longest valid chain) itself is the source. Unlike the ordinary three-party authenticated data structure model, the source

does not actively update on the data. Data are updated on queries from users to directories. On data update, the source still produces the structural authentication information. The source is still the trusted source of truth for data.

- Directories: physical entities that hold replicates of the data from the source. Directories interact with the source by listening for updates from the network, performing verifications, and eventually updating the changes to its own local replica. On requested by users, directories still take on the task of processing queries and responding with results and authentication proofs.

- Users: users receive trusted digest of the authenticated data structure from the source, and verify authenticated queries from directories.

Basically, blockchain synchronization is a client asking to become a directory by fetching all elements from the ADS as well as the authentication info. Having many directories helps balance query loads and avoids single point of failure. Therefore, this new query type forms a valid use case, and especially helps ensure the decentralization characteristics of blockchain.

3.2 SYNC MODES

Generally, there are 6 ways a client could synchronize with the network across popular Ethereum clients. All synchronization modes can be broken down into block synchronization and state trie synchronization. Block synchronization involves downloading blocks from the network and performing validations (proof-of-work, total difficulty, etc.) State trie synchronization involves acquiring the state trie and validating state trie root consistency with the block it belongs to. State tries can be acquired either through replaying history transactions or downloading directly from the network.

3.2.1 Full Archival Sync

Full archival sync is the most classical synchronization mode. In a full archival sync mode, the client connects to a few peers and starts to construct the block skeleton – it downloads every 192^{nd} block from one peer. It then downloads the rest of the blocks from multiple peers in parallel. Then it processes (verifies proof-of-work, executes transactions) through all history transactions to generate states block by block. Full archival nodes hence preserves all history states, so they act like persistent authenticated dictionaries [16]. This behavior remains the same after the sync is finished: the client listens for newly mined blocks and

processes transactions to generate new states. In this approach, the client would end up with all history states. However, there is no way to parallelize this process because each state at block $i + 1$ is based on the previous state i. Hence state generation must be sequential.

Running a full archival node requires plentiful computing resources. The current size of the underlying database of a full archival node has exceeded 1TB. As of now, it typically takes weeks and even months for a full archival sync to finish.

3.2.2 Fast Sync

Fast sync differs from full archival sync in the way it synchronizes state data. In fast sync mode, the client connects to a few peers and download all the blocks (first construct skeleton, then fill in the gaps). Then it would grab one of the latest state Merkle roots (latest block - 64 for Geth), dynamically expand the state trie level by level, and request to download state data from peers. For example, the client will first request to download the data for the state root. If the data is an RLP encoded list, it adds the elements to a queue and requests to download each of them. This process recurses until the queue is empty. At this point, the entire state trie is traversed. In this approach, the client would end up with only the latest state snapshot, which is sufficient for regular use. After the fast sync finishes, it will switch to full archival mode from then.

Fast sync is much more resource-friendly to ordinary end users, and thus it is set as the default sync mode on most Ethereum clients. However, it still takes about one week to finish a fast sync from scratch. Figure 1.1 shows the growth of the size of a fast sync node database. The data size of a fast sync node is now roughly 80GB.

3.2.3 Warp Sync

Warp sync[31] is Parity's proposal for a fast way to sync. Warp Sync is an extension to the Ethereum Wire protocol[32], which involves sending snapshots over the network to get the full state at a given block extremely quickly, and then filling in the blocks between the genesis and the snapshot in the background. Warp sync heavily relies on the availability of state snapshots in the network. Warp nodes take consensus critical snapshots every 5000 blocks. A snapshot will contain the previous 5000 blocks as well as the latest state. A snapshot is essentially a set of chunks of 4MiB containing block data and state data. On snapshot begins, Warp sync algorithm looks back 5000 blocks and serialize data to disk. When a

client initiates a Warp sync, it will broadcast to the network trying to find available peers with prospective snapshots. One advantage Warp sync has over fast sync is that, instead of switching to full archival sync mode after the initial sync is finished, Warp node triggers the Warp sync protocol whenever the client finds itself more than 5000 blocks behind the network.

Warp sync worked well in late 2016 and early 2017. It only took less than an hour when it was first introduced. However, due to the fact that Ethereum state data is growing so fast, taking snapshots every 5000 blocks is becoming a burden, and it does not work well with Parity built-in state pruning mechanisms[11]. Therefore, the number of Warp snapshots and Warp nodes are decreasing.

3.2.4 Light Sync

As its name suggests, light sync mode downloads minimum data and provides minimum functionalities. Light Ethereum Subprotocol, or LES, is inspired by Bitcoin's SPV[33] concept. A Bitcoin SPV client only downloads the block headers during the syncing process and then requests transactions from full nodes as needed. With the Merkle root in the block header and a Merkle branch as the authentication proof from the remote full node, the SPV client can know whether the transaction in question is included in the blockchain or not.

Similar to SPV, light sync only downloads the block headers and delegate a lot of verifications and state requests to other peers. Light sync was designed to run on low-capacity environments (embedded IoT devices, smartphones, browser extensions, some desktops, etc.) to maintain a high-security assurance about the current state of some particular part of the Ethereum state or verify the execution of a transaction. It sacrifices part of the decentralization characteristics and requires some amount of trust on other peers.

Light sync modes are experimental in most Ethereum clients. Peers could choose not to accept connections from light sync nodes. In a good setting where a light sync node is able to find and connect to peers, synchronization can finish within an hour. However, empirical experiments show that most of the nodes in Ethereum network are either not accepting connections from light nodes, or have already reached the configured maximum number of peer connections, and therefore rejecting all incoming peer discovery requests. Most of the time, a light node cannot find any nodes to connect to [34]. This makes light sync mode hardly stable to use. In addition, it is desirable that all network participants are able to

check the blockchain with full security guarantees on commodity hardware, both for their own benefit and because maintaining a large number of nodes performing full validation is important for the health of the cryptocurrency[35].

3.2.5 Snapshot Export/Import

A client can export its entire data as a snapshot to the disk, and later import it. Exporting data is essentially serializing Ethereum block data to the disk. The process of importing the snapshot is similar to a full archival sync, and therefore, its performance is not any better.

Strictly speaking, snapshot export/import does not count as a cross-network synchronization method. It is considered more as a way of data backup.

3.2.6 Directory Copy

A client can make a copy of the Ethereum client data file directory, and later run a different client process on top of the copied data directory. The new client process would immediately be on the same page with what the previous client has.

Technically speaking, this is the fastest way to port over all Ethereum data over. However, this method only makes sense within a trusted environment. The new client process that runs on the copied data directory would not retrospect to perform any verifications. The new client process would simply assume the data in the data directory are legit. Therefore, strictly speaking, this does not count as a cross-network synchronization method, but rather more as a way of data backup.

3.3 GOAL

Based on the survey on available sync modes, we observe that block data synchronization across different sync modes is similar. However, state sync has many approaches. Light Sync, which only downloads all block headers, can finish within 1 hour. On the other hand, Fast Sync, which not only downloads all block headers but also downloads the latest state, takes around a week to finish. It is not hard to tell from this fact that the approach to sync state decides the overall synchronization performance. State sync is the bottleneck.

We denote m as the size of the elements in the state trie at any time, N as the total number of records in the data store, and we have n total queries/transactions. Essentially, it takes $O(\log(m))$ disk IOs for one query. In a full archival node, which is essentially a persistent authenticated dictionary that archives all history state tries, N is $O(n \log(m))$. It stores the data in a general purpose on-disk data store, based on a LSM tree. Since keys in Ethereum are hashes, and uses random key-value layout, iterating through all the state naively takes $O(n \log(n \log(m)))$ time. Fetching the latest state trie, which is necessary for new nodes to sync, takes $O(m \log(n \log(m)))$ disk IOs. Our goal is to design the system so that it only takes $O(m)$ disk IOs to fetch all m elements of a state trie while remaining the update at $O(\log(m))$, and space usage at $O(n \log(m))$.

CHAPTER 4: SYSTEM DESIGN

In this chapter, we propose two types of synchronization modes for both normal end users who want only the latest state to send transactions and call smart contracts (Turbo Sync) and data analytic users who want the entire state history (Checkpoint Sync).

4.1 TURBO SYNC

4.1.1 Motivation

State sync involves two parts: downloading the account details and verifying data integrity. Fetching account details by expanding state trie level by level unnecessarily amplifies the network requests and disk IOs. Instead, if we could have the account details transmitted as a whole, and rebuild the state trie from bottom up for verification purpose, the performance should be much better. From this perspective, Parity Warp Sync has made a reasonable attempt towards this direction. However, one of the key problems with Warp Sync in the meantime is that there are simply few latest snapshots available on the network. With the growing traffic happening on Ethereum, building Warp Sync snapshots is itself resource-consuming.

Warp Sync generates snapshots in the same fashion as traversing the state trie. When it generates snapshots, it spawns a thread to walk through the trie and serialize the data. Trie traversal is not only slow, but it can also cause out-of-memory errors if not leveraging a separate standalone message queue software, or even lead to failure when part of the target state are pruned. We figure generating snapshots on demand is not reasonable performance-wise, so in Turbo Sync, the latest snapshot is always ready and is stored separately from the ordinary ADS data store.

4.1.2 Design

Turbo Sync aims to build upon Parity Warp Sync idea to further improve the snapshot generation performance. There are two data stores in a Turbo Sync Ethereum client. One is an ordinary ADS that stores the MPT just like what existing reference implementations have, and the other is a latest state cache which only stores the latest Ethereum account details in key-value pairs. Figure 4.1 shows an example view of the Turbo Sync data stores.

On data updates, records are updated in both the ADS and state cache. When it is time to generate a state snapshot, the client

1. Replicates the state cache
2. Serializes the state cache into a snapshot
3. Chunks it into non-overlapping data parts so that a node that is requesting the snapshot can not only download the chunks from multiple peers, but when some chunks failed to transmit from the peers, it only needs to re-download the failed chunks instead of the entire snapshot

This saves the process of digging through the ADS to traverse the state trie, and thus significantly improves the snapshot generation performance. In addition, since the snapshot generation now involves only a sequential scan over the replicate of the state cache, it avoids the risk of running out of heap memory or the data needed has already been pruned.

4.1.3 Implementation

The key component in Turbo Sync is the state cache. An ideal state cache should have the following characteristics:

- Key value store, so that it suits account address to account detail mappings well
- Sorted, so that the data in the snapshot are in order and deterministic
- Full table scan in linear time complexity, so that there is no overhead regarding serialization

Considering the above characteristics, we stick with LevelDB to be the state cache. We use a similar snapshot format to the one that Parity Warp Sync snapshot uses[36]. In Warp Sync snapshots, the address, details (nonce, balance, storage root, code hash), and the storage items of an account are placed together in an object known as the *Rich Account Structure*. This not only makes the structure clear, but also improves data locality and makes data verification easier.

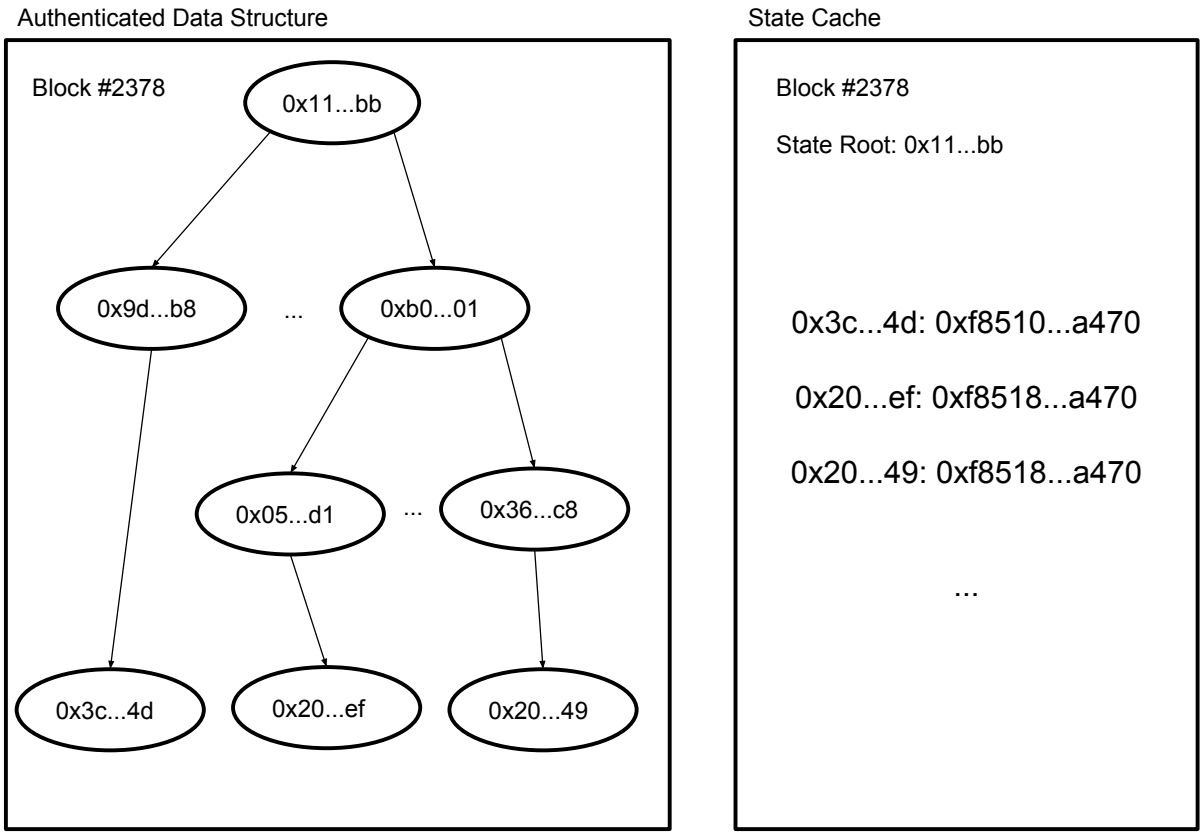


Figure 4.1: **Turbo Sync State Cache the Ordinary Authenticated Data Structure**

Listing 4.1 shows the schema of a Turbo Sync snapshot as well as the value part of the state cache. We choose to use Protobuf[37] as the serialization protocol for its maturity, performance, and cross-platform compatibility.

When it is time to generate a snapshot from the state cache, the client simply iterates through the data store, writing underlying key-value pairs to a local file. The file is then chunked so that snapshot dissemination can be parallelized.

When a client receives all chunks of a snapshot, it will attempt to rebuild the state cache first while at the same time verifying the integrity of the snapshot. The verification involves 3 parts:

- Verify that each account's storage items (if any) aggregate to the correct storageRoot
- Verify that the RLP encoded account detail matches with the provided RLP value

Listing 4.1: Turbo Sync Snapshot Schema in Protobuf

```
syntax = "proto3";
package snapshot;

message StorageItem {
    bytes key = 1;
    bytes val = 2;
}

message StorageItems {
    repeated StorageItem items = 1;
}

message AccountDetail {
    bytes nonce = 1;
    bytes balance = 2;
    bytes storageRoot = 3;
    bytes codeHash = 4;

    bytes rlp = 5;

    StorageItems storage = 6;
}

message Account {
    bytes addr = 1;
    AccountDetail detail = 2;
}
```

- Verify that all account address and RLP encoded account details aggregates to the correct stateRoot

When the state cache is rebuilt and verified, it is time to rebuild the ordinary authenticated data store. The client simply creates an empty persistent MPT, writes all account address and detail key-value pairs to the MPT, and writes all storage items directly into the DB.

Every subroutine above can be run concurrently with multiple threads. State cache leverages a sorted data store so multiple threads can iterate key ranges at different positions when looping through the state cache in `GenerateSnapshot`. Since MPTs are history independent[23], insertion order of the data does not affect the root hash of the MPT. Therefore, both stateRoot verification in `RebuildStateCache` and state trie rebuilding in `RebuildADS` can run in a multithreaded fashion.

There are 3 core subroutines in the Turbo Sync: `GenerateSnapshot`, `RebuildStateCache`, and `RebuildADS`.

Listing 4.2: Pseudocode for `GenerateSnapshot`

```
func GenerateSnapshot(stateCachePath, snapshotFile) {
  // Open the snapshot file
  file := os.OpenFile(snapshotFile)

  // Open the state cache
  db := leveldb.OpenFile(stateCachePath)

  // Write snapshot meta
  file.WriteMeta("stateRoot", db.Get("stateRoot"))
  file.WriteMeta("blockHash", db.Get("blockHash"))

  // Create a database iterator
  iter := db.NewIterator()

  // Start iterate through the entire state cache
  for iter.Next() {
    // Write the key-value pairs from state cache to the snapshot file
    file.Write(iter.Key, iter.Val)
  }
}
```

4.1.4 Chain Reorg

One critical issue from the Turbo Sync design is its ability to handle chain reorgs. Chain reorgs happen when a node on the Ethereum network realizes that the chain it maintains locally is no longer the canonical/longest chain. When this happens, the transactions in the latter part of its chain (i.e. the most recent transactions) are reverted, and the transactions in the newer replacement blocks are executed. This realization could be due to ineffective network propagation, or certain miners performing selfish mining[38].

During chain reorgs, recent transactions are reverted, and new transactions are executed, causing complicated state change. Ethereum MPT is designed to handle chain reorgs gracefully. Once a chain reorg event is detected, the client simply looks back and locates the last state in the old chain that is also belonging to the new canonical chain, and then applies new transactions on top of it, forming new latest states. In our proposed Turbo Sync, things could be more complicated. The state cache always stores the latest state and, unlike ADS, it preserves no history. Reverting state changes in state cache store is not as easy as applying new changes in the ADS.

Meanwhile, Turbo Sync's solution to chain reorg is straightforward. Turbo Sync by default replicates the state cache every 200 blocks (roughly one hour of time). Retention for the replicated state cache is another 400 blocks. The retention of the state cache replica is

Listing 4.3: Pseudocode for `RebuildStateCache`

```
func RebuildStateCache(stateCachePath, snapshotFile) {
  // Open the snapshot file
  file := os.OpenFile(snapshotFile)

  // Open the state cache
  db := leveldb.OpenFile(stateCachePath)

  // Write snapshot meta
  db.Put("stateRoot", file.ReadMeta("stateRoot"))
  db.Put("blockHash", file.ReadMeta("blockHash"))

  // Create a new empty non-persistent state trie
  stateTrie := trie.NewInMemoryMPT()

  // Iterate through all key-value pairs from the snapshot file
  for k, v in file {
    // Unmarshal the value part
    account := protobuf.Unmarshal(v)

    // Verify storage root
    if account.Detail.Storage != nil {
      // Create a new empty non-persistent storage trie
      storageTrie := trie.NewInMemoryMPT()

      // Iterate through all storage key-value pairs of this account
      for item in account.Detail.Storage.Items {
        // Update the storage trie
        storageTrie.Update(item.Key, item.Val)
      }

      // Check the validity of the constructed storage trie root
      assert storageTrie.Root() == account.Detail.StorageRoot
    }

    // RLP encode the account details
    encoded := rlp.Encode(account)

    // Check the validity of the RLP encoding
    assert encoded == account.Detail.Rlp

    // Update the state trie
    stateTrie.Update(account.Addr, account.Detail.Rlp)

    // Insert this key-value pair to the state cache
    db.Put(k, v)
  }

  // Check the validity of the constructed state trie root
  assert stateTrie.Root() == file.ReadMeta("stateRoot")
}
```

Listing 4.4: Pseudocode for `RebuildADS`

```
func RebuildADS(stateCachePath, adsLDBPath) {
  // Open the state cache and the ADS data store
  cacheDb, adsDb := leveldb.OpenFile(stateCachePath, adsLDBPath)

  // Create a new empty persistent state trie
  stateTrie := trie.NewPersistentMPT(adsDb)

  // Create a database iterator
  iter := cacheDb.NewIterator()

  // Start iterate through the entire state cache
  for iter.Next() {
    // Unmarshal the value part
    account := protobuf.Unmarshal(iter.Val)

    // Update the state trie
    stateTrie.Update(account.Addr, account.Detail.Rlp)
    if account.Detail.Storage != nil {
      // Iterate through all storage key-value pairs of this account
      for item in account.Detail.Storage.Items {
        // Insert the storage items into the ADS data store
        adsDb.Put(item.Key, item.Val)
      }
    }
  }
}
```

carefully decided by referring to the official chain reorganization depth expectations advised by Gavin Wood[39]. In this way, there are always at least two, at most three copies of state cache at any time. Namely, we have state cache replicates at block height x and $x + 200$, and the latest state cache at $x + 200 + y$ (where $0 < y < 200$). During a chain reorg with latest common ancestor block being z:

- If $x + 200 <= z < x + 200 + y$: discard the latest state cache, revert to the state cache replicate at $x + 200$, apply all new transactions on top of it, generate new latest state cache

- If $x <= z < x + 200$: discard both the latest state cache and the state cache replicate at $x + 200$, revert to the state cache replicate at x (the earliest state cache replicate available), apply all new transactions on top of it, generate new state cache replicate at $x + 200$ as well as the new latest state cache

- If $z < x$: this case is unhandled. When the chain reorg has to revert more than 400 blocks, the node would have to invalidate the existing state cache. However, base on past research on the security and performance of proof-of-work blockchain [40, 41, 42],

the probability of such chain reorg is 0.0000112% in Ethereum [43]. So far, the deepest chain reorg happens and around 165 blocks are reverted [44].

4.1.5 Asymptotic Performance Analysis of TurboSync

Since the latest state snapshot is always ready in the state cache, Turbo Sync snapshot generation is $O(m)$ disk IO complexity. Network bandwidth usage is $O(m)$ because it does not transmit any internal nodes of the MPT. On receiving state snapshots, reconstruction of the trie is in $O(n \log(m))$. We present the asymptotic comparison between Turbo Sync, Geth Fast Sync, and Parity Warp Sync in Table 4.1.

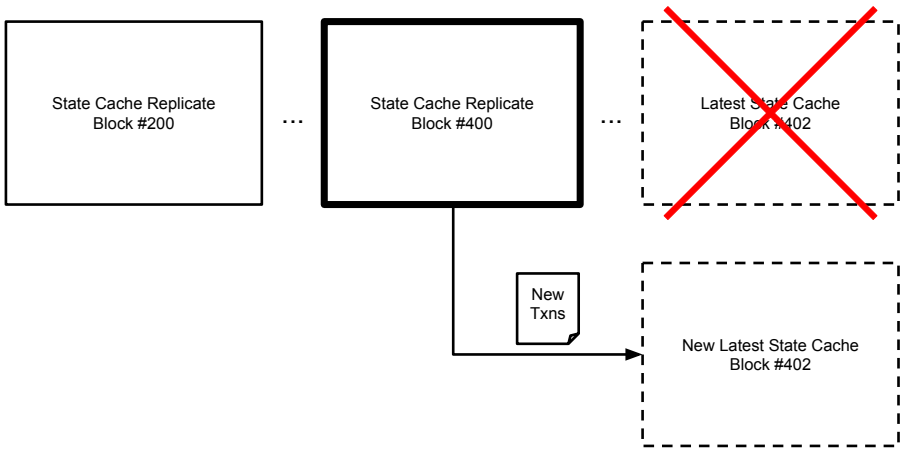
	Turbo Sync	Fast Sync	Warp Sync
Fetching latest state	$O(m)$	$O(m \log(n \log(m)))$	$O(m \log(n \log(m)))$
Network Bandwidth	$O(m)$	$O(n \log(m))$	$O(m)$
Update	$O(\log(m))$ (ADS), $O(\log(m))$ (state cache)	$O(\log(m))$	$O(\log(m))$
Space Usage	$O(n \log(m))$ (ADS), $O(m)$ (state cache)	$O(n \log(m))$	$O(n \log(m))$

Table 4.1: **Turbo Sync vs. Fast Sync vs. Warp Sync**

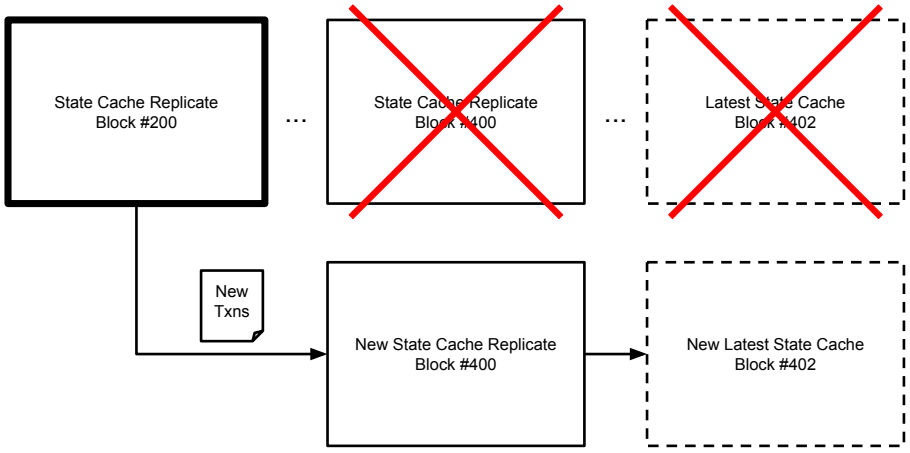
4.2 CHECKPOINT SYNC

4.2.1 Motivation

For data analytic users, block explorer[45], and blockchain service providers[46], it is necessary to attain all history states at every block. Meanwhile, the only way to attain all history states is through a full archival sync where the node sequentially processes all history transactions. The problem with full archival sync is that there is no way to parallel the process because each state at $i+1$ is based on the previous state i. When the node does not have state i, it cannot process any transactions in block i. Currently, a full archival sync can take weeks to months to finish. If the process can run in parallel, not only can a single machine run at full speed towards the sync, but we could also leverage multiple machines to distribute synchronization workloads.



(a)



(b)

Figure 4.2: **Chain Reorg Cases.** (a) shows the chain reorg handling when the latest common block that belongs to both the old chain and new chain is less than 200 blocks into the past, and (b) shows the case when the common block is more than 200 blocks but less than 400 blocks into the past.

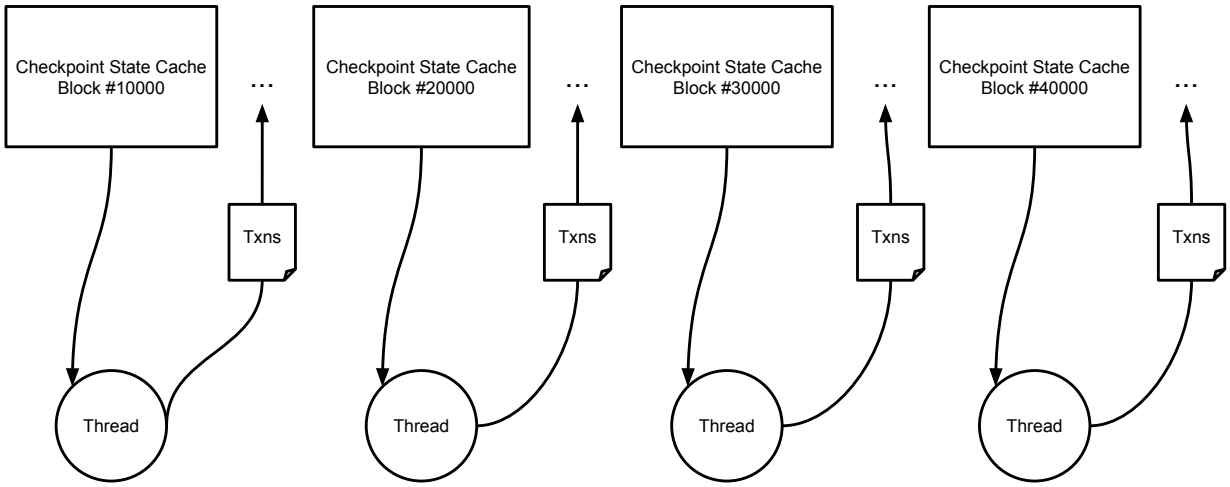


Figure 4.3: **Checkpoint Sync.** This chart illustrates how multiple workers can concurrently process transactions based on checkpoint states.

4.2.2 Design

Checkpoint Sync aims to parallelize the process of full archival sync in the following procedure:

1. Download all the blocks data (including transactions) normally
2. The client allocates threads/workers to Turbo Sync states at evenly distributed block height depending on parallelism factors
3. After checkpoint states are acquired, the client allocates threads/workers to process transactions simultaneously starting from these checkpoint states

4.2.3 Implementation

In almost all sync modes, block data sync starts with constructing a block skeleton where the client downloads blocks at every 192^{nd} block. We add similar logic to the block skeleton building code to issue Turbo Sync requests at every X^{th} block. X is a configurable parameter that depends on parallelism factors such as the number of CPU cores, etc. Multiple Turbo Sync requests do not affect each other because the they are downloading and validating state snapshots at different block height. Therefore, these Turbo Sync requests can be run in parallel.

Checkpoint Sync heavily depends on the availability of state snapshots at all block heights in the network. If only the latest state snapshots exist in the network, the concurrent workers that replay transactions will have uneven workload. The performance of the synchronization will then be bounded by the slowest worker who has the heaviest workload. It is most ideal if full nodes store all the state caches at every block. However, it is possible that not all nodes want to or have the ability to archive all state snapshots. To workaround that, it is possible for nodes to store random retention of state snapshots, so by querying multiple nodes, Checkpoint Sync can eventually find a large sampling of available state snapshots that satisfy an evenly distributed concurrent workload. We will leave this for future work.

After all Turbo Sync snapshots are acquired and rebuilt into ADS, we have states at every X^{th} block. The node will start concurrently replaying transactions starting from these states. From this point on, the processing logic is the same as that of full archival sync except that we are now able to leverage the multi-core computation resource to parallelize transaction processing.

CHAPTER 5: IMPLEMENTATIONS AND EVALUATIONS

In this chapter, we will briefly discuss how to adapt the two new sync modes in Ethereum reference implementations. We will also describe experiments and present evaluation/validation results.

5.1 IMPLEMENTATIONS

Both Turbo Sync and Checkpoint Sync can be implemented and added to any existing Ethereum clients. Since neither of the sync modes invalidates any blocks or transactions in the history, the new sync modes can be added without requiring a fork of any form. In pretty much the same way Parity adds Warp Sync to the client, Turbo Sync is an extension to the Ethereum Wire protocol. Checkpoint Sync is even easier as itself does not require additions to wire protocols.

Turbo Sync can be added as a subprotocol built on the DEVP2P networking layer, with 3-byte identifier `tur`. The Turbo Sync protocol inherits all packets from `eth63` wire protocol, as well as amending the status packet (`0x00`) and adding a few new ones:

- `Status: [0x00, ..., snapshot_ver:uint64]`. In addition to all the fields in `eth63` protocol's status (denoted by `...`), include `snapshot_ver` which signifies the block number of the peer's local snapshot.
- `GetSnapshotMetaByVer: [0x21, snapshot_ver:uint64]`. Request snapshot metadata at the specific block from a peer.
- `SnapshotMeta: [0x22, metadata or nothing]`. Respond to a `GetSnapshotMetaByVer` message with either an empty list if the snapshot of requested version does not exist, or a 1-item list containing the requested snapshot metadata.
- `GetSnapshotData: [0x23, chunk_hash:byte32]`. Request a chunk (identified by the given hash) from a peer.
- `SnapshotData: [0x24, chunk_data or nothing]`. Respond to a `GetSnapshotData` message with either an empty list or a 1-item list containing the raw chunk data requested.

5.2 EVALUATIONS

We implemented Turbo Sync based on Geth 1.8.1. We set up two fresh Geth clients with one synchronizing using Turbo Sync, and the other one with Geth default Fast Sync. The experiments throughout this entire chapter run on a server with 32 Intel(R) Xeon(R) CPU E5-2620 v4 @2.10GHz processors, each with 8 cores, 128GB memory, equipped with large enough SSDs.

We first run and time the Geth Fast Sync up until block height 5738444 (blockHash: `0x64d4ff53be4bcbc504f509558c61dcb5354e1e6ea595ff740b1d530a5c109245`, stateRoot: `0x587ee10ea8af4d603d243b0146275a906d552ec061e1a424a0fa5d4dcea042c0`) on mainnet. The client connects to maximum 25 peers which is set by default[47]. The state synchronization takes 5 days. Since it takes so long to finish Fast Sync, we only run this experiment once. During the sync, peers got dropped and added in the network arbitrarily. Information gathered from this run is thus limited.

The resulting state trie contains 87,954,570 nodes in total where 37,973,685 of them are actual data nodes. The resulting ADS data store that contains the block data, transaction data, and state data has in total 674,332,540 records.

According to previous analysis in Chapter 3 and Chapter 4, fetching all data elements in the ordinary ADS data store is $O(m \log(N))$ where m is the size of data nodes, and N is the total amount of entries (including internal branching nodes) in the data store, while fetching all elements from state cache is $O(m)$. Here in our setting, $m = 37,973,685, N = 674,332,540$. Therefore, we should see at least a 30-times performance improvement.

With the state at hand, we convert and craft a client instance so that it is equipped with a state cache and is able to provide state snapshots. We measure the performance of state data iteration. State data iteration performance is critical in snapshot generation. In Geth, we used the builtin `node.Iterator` to iterate through the state trie, trying to output the same snapshot format we used in Turbo Sync. The builtin `node.Iterator` traverses the trie from the root all the way to leaves. The iteration takes over 6 days. We use the state cache to generate the same snapshot, and it takes merely over half a minute. Turbo Sync speeds up snapshot generation by an astonishing 14,810 times.

At this block height, the generated snapshot file is 21GB in size. However, the state cache

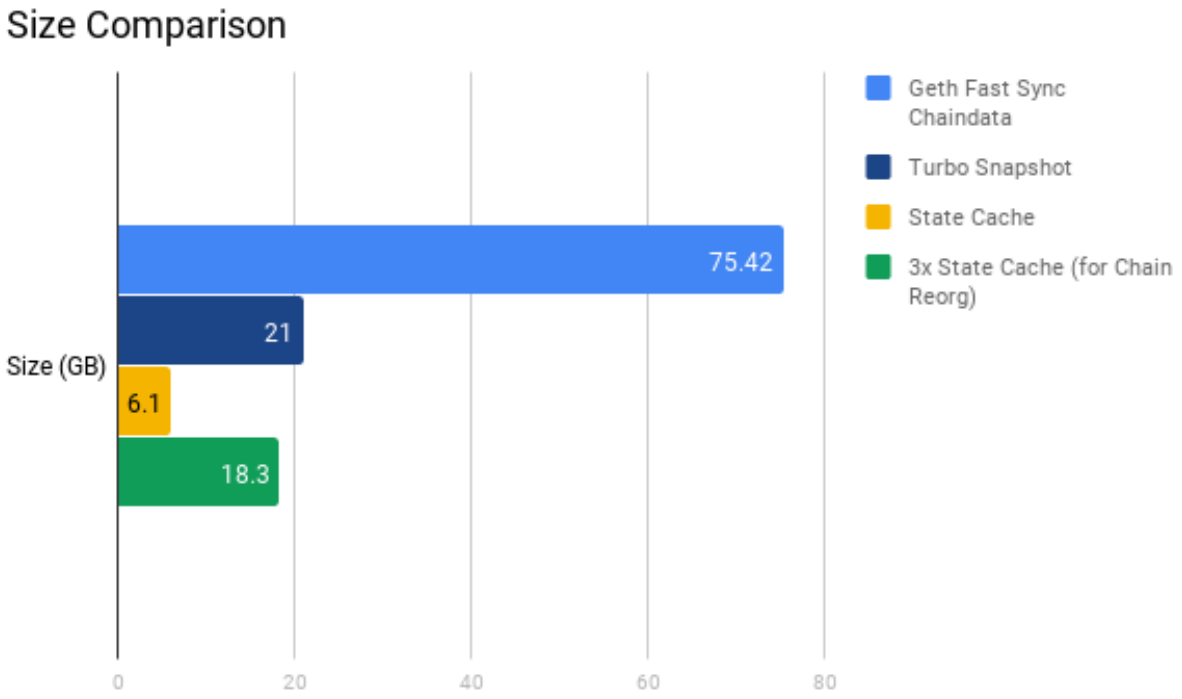


Figure 5.1: **TurboSync Vs. Geth: Size Comparison**

rebuilt based on the snapshot is around 6.1GB thanks to LevelDB compression. Comparing with the entire ADS datastore size (75.42GB), the state cache size is relatively small. It is also manageable to maintain at most 3 copies of the state cache for potential chain reorg. Since the states at a particular block height are always the same set of data, the snapshot file and data store sizes are consistent among multiple runs of the experiment.

We then connect the Turbo Sync client to the crafted peer. With only 1 peer, the Turbo Sync client finishes rebuilding state cache and snapshot integrity verification within 746 seconds. It finishes rebuilding the ordinary authenticated data structure DB within 1166 seconds. In total, state sync finishes within 2000 seconds. Turbo Sync speeds up state sync by 216 times. The experiment was run more than once and the time measured are consistent.

5.3 IMPACT

From these experiments, we show that the new sync modes are able to finish state sync in a little over 30 minutes. The space tradeoff required for state cache is minimum comparing

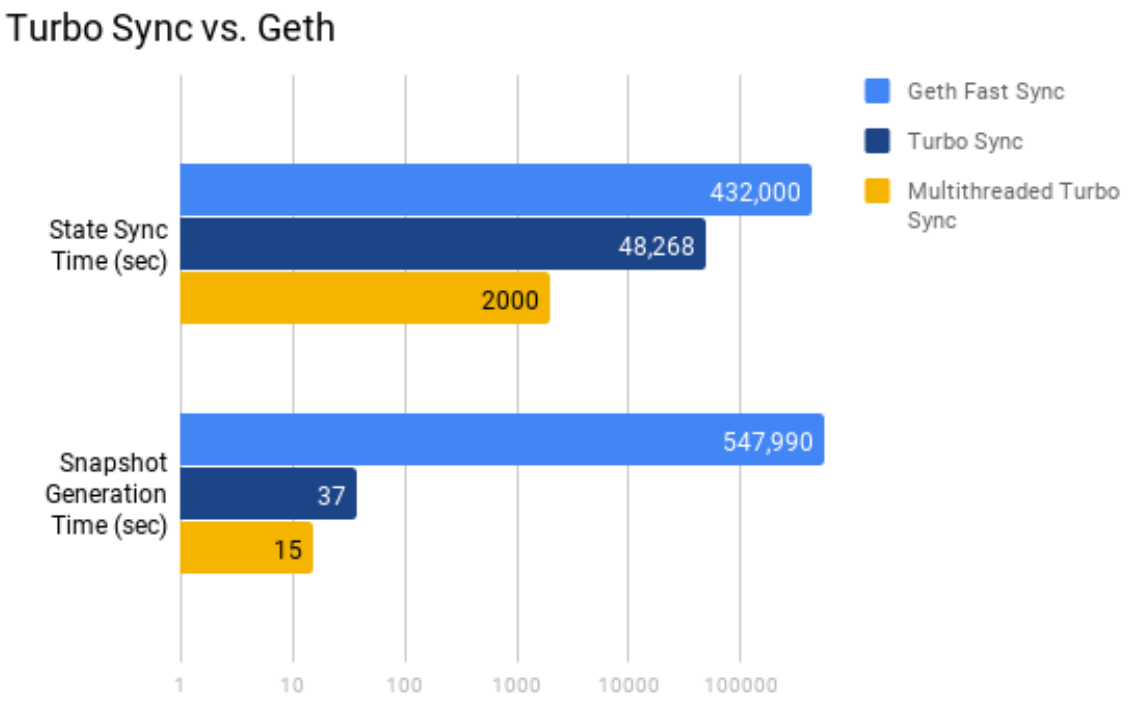


Figure 5.2: **TurboSync Vs. Geth: Time Comparison in Log Scale**

to the ordinary ADS data store. In addition, since it is fast to generate snapshots from state cache, snapshots will be highly available across the network if Turbo Sync is widely adopted. The client can choose to trigger Turbo Sync anytime to catch up with the latest state when it finds itself left behind by too much.

CHAPTER 6: CONCLUSION

In this work, we defined the deviation of three-party authenticated data structure model that is specific to public blockchains. We demonstrated a new query type in this model that corresponds to data synchronization in public blockchain network such as Ethereum. We argued that the Ethereum state data synchronization has become a serious performance issue. We proposed a new architectural design for the ADS system. We adapted the new design to Ethereum which resulted in two new sync modes. The new sync mode is able to speed up Ethereum state sync by 216 times.

CHAPTER 7: FUTURE WORK

7.1 RESOURCE-EFFICIENT DATA ANALYTIC FRAMEWORK

With Turbo Sync and Checkpoint Sync, syncing an Ethereum node with full history states will be no longer time-consuming. Although it might still be considerably demanding in computing resource, we can explore how to build a resource-efficient blockchain data analytic framework (like BlockSci[48], New Kids on the Block[49], Presto-Ethereum[50]) or block explorer (like Etherscan[45]) by trading off storage space and time. One possible direction could be a data analytic framework that processes transactions (in order to acquire EVM traces, etc.) and gathers information on demand. With Checkpoint states available, the data analytic framework can have a better query optimizer that processes transactions and cache process results only when it is necessary.

7.2 BETTER CHAIN-REORG HANDLING

Currently, Turbo Sync's approach to chain reorg is mimicking the ordinary chain reorg flow in MPT, which is applying new transactions on an earlier consensus critical state. However, the base state Turbo Sync applies new transactions on may not necessarily be the most optimal as it can cause the client to revert to a state that is too old. We can explore maintaining a finer-grained write-ahead log (WAL) that records changes to the state cache. In this way, instead of reverting at least 200 blocks-worth of transactions, the client can refer to the WAL to revert only a few writes in state cache during chain reorgs.

7.3 BETTER DATA STORAGE PLAN

Parity has set a good example of exploring custom data storage[51] outside of LevelDB or RocksDB. In fact, key-value store is not necessarily the most ideal interface for blockchain state data. They do not only have to organize in an authenticated data structure paradigm (like MPT), but could also be nested. Take Ethereum for example: the account data contains a storageRoot as part of the value, which points to yet another MPT structure. However, when retrieving details of an account during a query, it is very likely that the storage would need to change. Thus, a well-designed storage plan for Ethereum is critical and is urgently needed.

7.4 MORE DATA IN CACHE DATASTORE

Currently, the state cache only saves the latest account state and storage items. We could also explore putting more data in the cache data store so that they can be transmitted as a snapshot to peers more efficiently. One thing that should be trivial to add is to make a cache data store for both transaction trie and transaction receipt trie.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system,âĂİ http://bitcoin.org/bitcoin.pdf," 2008.

[2] V. Buterin et al., "A next-generation smart contract and decentralized application platform," *white paper*, 2014.

[3] C. Catalini and J. S. Gans, "Initial coin offerings and the value of crypto tokens," National Bureau of Economic Research, Tech. Rep., 2018.

[4] "The endless node sync!!!" https://github.com/ethereum/mist/issues/2466.

[5] "Ethereum wallet needs forever to sync," https://github.com/ethereum/mist/issues/3463.

[6] "geth –fast stalls before crossing finish line," https://github.com/ethereum/go-ethereum/issues/15001.

[7] "Geth: Ethereum golang client," 2018. [Online]. Available: https://github.com/ethereum/go-ethereum/

[8] "Parity: Ethereum rust client," 2018, Parity. [Online]. Available: https://github.com/paritytech/parity/

[9] "eth/63 fast synchronization algorithm," https://github.com/ethereum/go-ethereum/pull/1889.

[10] "Velocity: The fastest parity released," https://paritytech.io/velocity-the-fastest-parity-released/, Parity, 2018.

[11] "Parity warp sync is no longer very warpy," https://github.com/paritytech/parity/issues/6372.

[12] "Ethereum light client protocol," https://github.com/ethereum/wiki/wiki/Light-client-protocol, Ethereum.

[13] R. Tamassia, "Authenticated data structures," in *Algorithms - ESA 2003*, G. Di Battista and U. Zwick, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–5.

[14] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, "A general model for authenticated data structures," *Algorithmica*, vol. 39, no. 1, pp. 21–41, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1007/s00453-003-1076-8

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142488 pp. 121–132.

[16] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia, "Persistent authenticated dictionaries and their applications," in *Proceedings of the 4th International Conference on Information Security*, ser. ISC '01. Berlin, Heidelberg: Springer-Verlag, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=648025.744371 pp. 379–393.

[17] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich, "Versum: Verifiable computations over large public logs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660327 pp. 1304–1316.

[18] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A scalable cloud file system with efficient integrity checks," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420985 pp. 229–238.

[19] C. Kaufman, R. Perlman, and M. Speciner, *Network Security: Private Communication in a Public World, Second Edition*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2002.

[20] R. C. Merkle, "A certified digital signature," in *Proceedings on Advances in Cryptology*, ser. CRYPTO '89. New York, NY, USA: Springer-Verlag New York, Inc., 1989. [Online]. Available: http://dl.acm.org/citation.cfm?id=118209.118230 pp. 218–238.

[21] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535851 pp. 411–423.

[22] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger."

[23] M. Naor and V. Teague, "Anti-persistence: History independent data structures," in *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, ser. STOC '01. New York, NY, USA: ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/380752.380844 pp. 492–501.

[24] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, June 1996. [Online]. Available: http://dx.doi.org/10.1007/s002360050048

[25] "Leveldb," http://leveldb.org/, Google.

[26] "Rocksdb," http://rocksdb.org/, Facebook, 2018.

[27] "Bitcoin core 0.11 (ch 2): Data storage," https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage.

[28] H. Lim, D. G. Andersen, and M. Kaminsky, "Towards accurate and fast evaluation of multi-stage log-structured designs," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/lim pp. 149–166.

[29] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham, "mlsm: Making authenticated storage faster in ethereum," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotstorage18/presentation/raju

[30] A. Akhunov, "Roadmap for turbo-geth," https://medium.com/@akhounov/roadmap-for-turbo-geth-31cbfb1e72b7, 2018.

[31] "Parity warp sync," https://wiki.parity.io/Warp-Sync, Parity.

[32] "Ethereum wire protocol," https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol, Ethereum.

[33] "Spv, simplified payment verification," https://bitcoin.org/en/glossary/simplified-payment-verification, Bitcoin.org.

[34] "Not enough light peers available on the network," https://github.com/ethereum/go-ethereum/issues/15454.

[35] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov, "Improving authenticated dynamic dictionaries, with applications to cryptocurrencies," in *Financial Cryptography and Data Security*, A. Kiayias, Ed. Cham: Springer International Publishing, 2017, pp. 376–392.

[36] "Parity warp sync snapshot format," https://wiki.parity.io/Warp-Sync-Snapshot-Format, Parity.

[37] "Protocol buffers," https://developers.google.com/protocol-buffers/, Google, 2018.

[38] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *CoRR*, vol. abs/1311.0243, 2013. [Online]. Available: http://arxiv.org/abs/1311.0243

[39] G. Wood, "Chain reorganisation depth expectations," https://blog.ethereum.org/2015/08/08/chain-reorganisation-depth-expectations/, 2015.

[40] A. Kiayias and G. Panagiotakos, "Speed-security tradeoffs in blockchain protocols," 2015.

[41] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978341 pp. 3–16.

[42] V. Buterin, "On slow and fast block times," https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times//, 2015.

[43] V. Buterin, "On settlement finality," https://blog.ethereum.org/2016/05/09/on-settlement-finality/, 2016.

[44] V. Buterin, "Security alert [11/24/2016]: Consensus bug in geth v1.4.19 and v1.5.2," https://blog.ethereum.org/2016/11/25/security-alert-11242016-consensus-bug-geth-v1-4-19-v1-5-2/, 2016.

[45] "Etherscan," 2018. [Online]. Available: https://etherscan.io/

[46] "Infura - scalable blockchain infrastructure," 2018. [Online]. Available: https://infura.io/

[47] "Geth command line options," https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options, Ethereum.

[48] H. A. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan, "Blocksci: Design and applications of a blockchain analysis platform," *CoRR*, vol. abs/1709.02489, 2017. [Online]. Available: http://arxiv.org/abs/1709.02489

[49] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber, "New kids on the block: an analysis of modern blockchains," *CoRR*, vol. abs/1606.06530, 2016. [Online]. Available: http://arxiv.org/abs/1606.06530

[50] "Presto ethereum connector – sql on ethereum," 2018. [Online]. Available: https://github.com/xiaoyao1991/presto-ethereum

[51] "Paritydb: Fast and reliable database, optimised for read operations," 2018, Parity. [Online]. Available: https://github.com/paritytech/paritydb/