Scaling Blockchains

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Enrique Fynn

under the supervision of
Fernando Pedone

February 2021

Dissertation Committee

Antonio Carzaniga	Università della Svizzera Italiana, Switzerland
Marc Langheinrich	Università della Svizzera Italiana, Switzerland
Alysson Bessani	Universidade de Lisboa, Portugal
Zarko Milosevic	Informal Systems, Canada

Dissertation accepted on 9 February 2021

Research Advisor	PhD Program Director
Fernando Pedone	**Walter Binder**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Enrique Fynn
Lugano, 9 February 2021

If a boat travels without purpose it
does not arrive at any port.

Alejandro Jodorowsky

Abstract

Blockchains are a new type of state machine replication that have raised interesting challenges. A replicated state machine (RSM) is a well-established approach to building fault-tolerant systems. Because each replica needs to execute the same set of instructions to transition through the same state changes, adding more replicas does not translate directly to an increase in performance. On top of that, RSM can be made to tolerate Byzantine failures, i.e., nodes in the system can have arbitrary behavior. Blockchains distinguish themselves from traditional RSM mainly for having an open membership, being decentralized, and involving economic aspects as a means to counter adversarial attacks. Yet, despite their increasing popularity, current blockchain systems scale poorly and are constrained to exist in isolation, unable to communicate with each other.

In this thesis, we explore techniques to make blockchains scale in two angles: (a) scaling blockchain transaction throughput; and (b) making the state synchronization faster and robust for incoming peers. For (a), we analyze the effects of partitioning a real blockchain state in several shards and how to minimize communication within shards, while keeping the shards balanced. We then propose a protocol that can be applied to increase the throughput of a sharded blockchain or by which blockchains can communicate with each other. For (b), we propose a data structure that can be used by blockchains to enhance scalability by easing the synchronization process and allowing the blockchain's state to be reconstructed without requiring additional trust. The claims in this thesis are sustained by extensive experimental evaluation using real applications from public blockchains, developing new protocols and algorithms, and performing tests in geo-replicated environments.

Acknowledgements

First, I would like to thank Fernando Pedone for his patience, guidance, discussions and good advice during all these years of the PhD. I am also grateful to all professors I interacted with during my Teaching Assistance.

I would like to thank the committee members, Antonio Carzaniga, Marc Langheinrich, Alysson Bessani, and Zarko Milosevic for their patience and feedback, for dedicating their time to read and discuss this thesis with me.

For making the PhD experience more enjoyable and the OpenSpace more tolerable I'd like to thank my colleagues at USI: Theo, Leandro, Pietro, Ioannis, Esteban, Mojtaba, Long, Antti. Also everyone that keeps the university going, from janitors to deans: Thank you.

For endless discussions about branchial space, physics, and climbing, I would like to thank all my friends Leonardo, Álvaro, Vella, Thadeu, Matthieu and Andrea.

I would like to thank the great minds and spirits of Kropotkin, Bakunin, Proudhon, Chomsky, and many other anarchists that prove not only having a high intellect but kindness of heart and love for humanity. One day Earth will be *paradise*.

For a person whom I have only seen kindness, compassion and goodwill—my grandmother Alice Áurea—I am humbled by her kind spirit and grateful for her unconditional love. I would also like to thank all her children, especially Abigail for our fruitful discussions.

Last, I am eternally grateful for my parents, brother, and sister—Sheila, Enrique, Annie, and Erik—for everything from remote yoga sessions to political discussions.

Contents

Figures

Tables

Chapter 1

Introduction

Blockchain has gained much attention since the introduction of Bitcoin [55], and several blockchain systems have been developed thereafter [17]. Initially seen with skepticism by the academic community, blockchain revamped the interest in consensus research.

Blockchains distinguish themselves from traditional State Machine Replication (SMR) [52, 62] by having an open membership, i.e., anyone can join the protocol to secure the network. Blockchains are separated in two main categories: permissioned and permissionless. In a permissionless blockchain system, such as Bitcoin or Ethereum [16], blocks are produced by miners. Each block of a blockchain is cryptographically linked to the previous one (block's parent), forming a chain. To produce a valid block, miners must solve a cryptographic puzzle, a process called Proof-of-Work (PoW). It can happen that miners simultaneously publish their blocks pointing at the same parent, creating a fork. Peers follow an heuristic to decide the correct path of history by greedily choosing the chain that has more work done (heaviest chain). A clear incentive exists to solve the fork as soon as possible so that miners in the canonical blockchain history get their rewards. This incentive makes miners work on top of the chain they believe is the winner as soon as they see a fork, thus making the probability of forks exponentially smaller over time [55]. Attacking a permissionless blockchain is proportional to the amount of work miners did in the blockchain, since attackers need to recreate all cryptographic puzzles from the attack point onwards. Naturally, a miner who owns more than 50% of the hash power is able to attack the system by controlling the produced blocks, but even miners who own less than 50% are able to get more incentives than their contributed hash power [32].

In a permissioned blockchain system miners are in a consortium where they behave similarly to members of traditional Byzantine Fault Tolerant (BFT) algo-

rithms such as PBFT [19], in which case they are named *validators*. Permissioned blockchain usually use a Proof-of-Stake (PoS), which aims to provide a decentralized solution for consensus while still maintaining an open membership as in PoW. Validators *lock* some currency (stake) that is fungible in the underlying protocol's blockchain. If a member misbehaves or is offline during steps in the protocol it loses a portion of the stake (e.g., Cosmos [51], Ouroboros [49], Polkadot [72]) or the validator does not get the rewards (e.g., [6]).

In hindsight, moving away from PoW is a good idea since in 2019 alone it was estimated that only Bitcoin consumed 59 TWh [10], equivalent to the energy consumption of some countries. But there are clear tradeoffs when moving from permissionless and permissioned blockchains. For instance, permissioned blockchains require more trust than permissionless ones. In particular, they face a fundamental problem [14] where it is not possible for an arriving client to differentiate between the valid chain or a fork of the same blockchain.

1.1 Blockchain scalability

Blockchain scalability has many forms, for instance Bitcoin and PoW systems in general scale exceptionally well with the numbers of miners when considering the system security: the more miners there are, the more secure the network becomes. However, systems that use PoW consensus algorithms do not scale well in terms of performance throughput. For instance, one of the problems of PoW systems is that the execution is coupled with the consensus algorithm which makes miners not able to decide on an ordering of transactions before executing them.

Different blockchains provide different means for scalability. For example micro-transactions have different requirements than transactions involving large sums of currency, but blockchain systems such as Bitcoin treat both transactions with the same level of security guarantees. Having different blockchains allows for specialized blockchains crafted for different use cases. A way for communicating among blockchains is crucial for scaling blockchains in this direction.

As blockchain technology reaches mainstream use, it starts to face issues typical of more mature distributed systems technologies. Two formidable challenges to improve blockchain performance are *sharding* and *interoperability*. Several attempts have been made to improve blockchain performance (e.g., [49, 50, 73]). In general, distributed systems scale performance by sharding the application state [25]. If the partitioning is such that most application requests can be executed within a single partition, and the load among partitions is balanced, then

performance scales with the number of partitions. Unfortunately, few applications can be optimally partitioned (i.e., all requests fall within a single partition and load is balanced among partitions). As a result, most partitioned systems must handle requests that span multiple partitions. In the particular case of blockchain, it has been shown that even with a nearly perfect partitioning of the data, existing workloads would result in a substantial number of cross-partition. This aspect is discussed in Chapter 2.

The Achilles heel of scalable blockchain systems is their ability to handle cross-blockchain transactions. Interoperability has been in the blockchain wishlist for some time. Yet, to date, no general mechanism has been proposed to share information across different blockchains. Inter Blockchain Communication (IBC) is necessary for multiple blockchains to co-exist in a heterogeneous way.

Another aspect of scalability is the ability of the network to on-board new peers, a crucial component for blockchains to remain decentralized. For instance, several Bitcoin forks were made for diverging opinions on how to handle this aspect of scalability [29]. When a new peer joining the network has to either download the state or re-execute all state transitions that led to the final system's state, peers often choose the former option since it is more efficient in most cases for long-running blockchains. If the state synchronization is costly, fewer peers will participate in the protocol and the system becomes more centralized.

1.2 Research questions and contributions

This thesis identifies and addresses problems related to blockchain scalability and answers the following questions on how blockchain scalability can be improved:

- Can blockchain systems scale performance through sharding?
- How can blockchains scale with Inter Blockchain Communication?
- How to design scalable applications that can live in many blockchains?
- How to improve scalability with fast blockchain synchronization?

In Chapter 2, we answer the first question by observing how a popular blockchain behaves when sharded. For blockchains to interact with each other, some form of synchronization is required across blockchains. There are two main classes of solutions to handle transactions that involve multiple blockchains: (a) coordinating the blockchains involved in the execution of the transaction,

in a scheme akin to atomic commitment [39, 53]; and (b) moving the state required by the transaction to a single target blockchain and then executing the transaction locally at the target blockchain. In Chapter 3, we claim that **we can use sharding and interoperability to increase blockchain performance with a proposed framework and protocol**. We substantiate the claim with a common mechanism that can be used for both a sharded blockchain and IBC: a *move operation* that allows accounts and arbitrary computation (i.e., *smart contracts*) to consistently migrate from one blockchain to another.

Finally, we explore a different form of scalability to answer the last question of this thesis. In Chapter 4, we design a data structure and protocols and claim **we can speed up state synchronization while tolerating byzantine behavior and preserving original systems guarantees.**

Chapter 2

Partitioning a blockchain

In this chapter we investigate the performance implications of partitioning a blockchain state. We use Ethereum [16] as an example of blockchain to be partitioned. We briefly describe Ethereum, explain how to build a graph with information gathered by interactions with the blockchain's state, and detail the partitioning methods used in the study.

2.1 Ethereum's primer

Ethereum was the first general-purpose blockchain conceived. Users interact with Ethereum's blockchain by sending a transaction from a user account. Transactions might transfer *ether*, Ethereum's underlying currency, to another account or activate a *smart contract*. When transferring currency from an account to another, Ethereum works similarly to Bitcoin. But the ability to execute arbitrary code in the form of contracts grants Ethereum much more flexibility than Bitcoin.

A smart contract can read and modify internal storage (i.e., a database mapping 32-byte keys to 32-byte values), transfer currency to one or more accounts, and trigger the execution of other contracts. Contracts are written in Ethereum-specific binary format and executed by the Ethereum Virtual Machine (EVM), a 256-bit stack-based virtual machine. Contracts can be created and destroyed by other contracts or accounts. After the contract is created, it resides in the blockchain. The initial contract state can be set by using an initialization code that is only executed in the contract's creation and remains unaltered until the contract's destruction.

At the beginning of a transaction, users have to define the maximum ether they are willing to pay for the execution of their transaction (specified in *gas* and *gas price*). Determining the actual cost of a transaction is not obvious. In fact,

there is no way to tell whether a contract will end or not, since this problem is reducible to the halting problem, which is undecidable [21]. Users can estimate the cost of a transaction from the transaction's instructions and the cost of each instruction.

Miners include transactions in a block based on their estimates of the transaction cost and the amount the user is willing to pay for the transaction. When a block is included in the blockchain, the miner that proposed the block receives fees paid by each transaction in the native blockchain currency. The scheme is slightly more complex, for example, to cover cases in which transactions fail or run out of gas, but these aspects are not important in the context of our study.

Ethereum has experienced a rapid growth since its inception. Moreover, Ethereum's consensus rules have been revised (i.e., *forked*) many times. In Fig. 2.1, we can see the system growth in number of vertices (accounts and smart contracts) and edges (transactions). Each vertical dashed line in Fig. 2.1 shows a fork of the system due to the bloat attack [11], an attack that exploited Ethereum's vulnerabilities (i.e., the attack exploited mispriced operations in the EVM). During the attack period, the number of vertices and edges increased by one order of magnitude. Afterwards the system continued to grow quickly, although at a lower pace than prior to the attack. A solution to reduce the significant state growth after the attack was done, eliminating *empty* accounts. We chose not to show the reduced state after the solution was implemented in the figure.

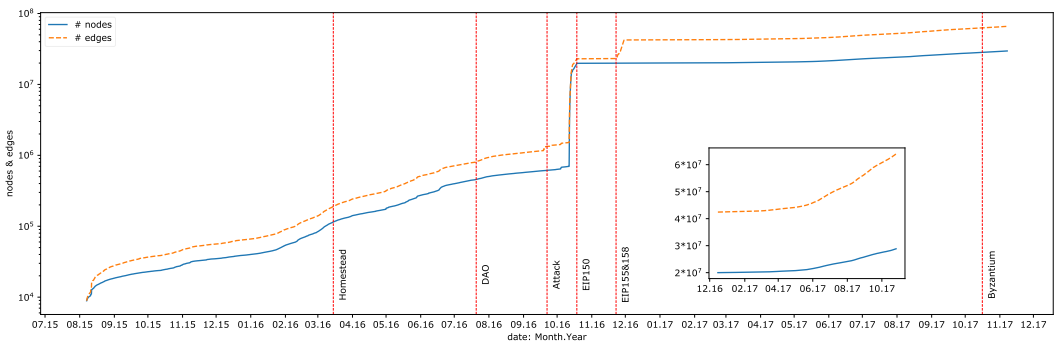


Figure 2.1. Ethereum graph evolution in number of vertices (accounts and smart contracts) and edges (transactions).

2.2 Ethereum's blockchain graph

We build a directed graph based on the transactions in Ethereum from July 2015 to December 2017. In the graph, vertices represent accounts or contracts, and edges represent interactions involving accounts or contracts. All the following cases lead to an edge e from vertex v_1 to vertex v_2 in the graph: a user with account v_1 submits a transaction that transfers some currency to account v_2 or that activates contract v_2; contract v_1 transfers ether to account v_2 or activates contract v_2.

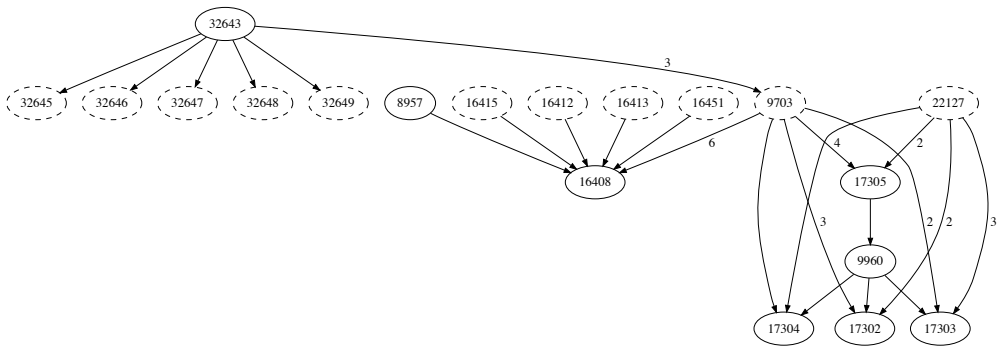


Figure 2.2. Subgraph with accounts (full line nodes), contracts (dashed line nodes), and their dependencies (arrows).

Fig. 2.2 shows a subgraph of the Ethereum graph in September 2015. Full-line nodes represent accounts and dashed-line nodes represent contracts. Edges originating in accounts are single transactions, edges originating in contracts can perform different calls in the same transaction, both to accounts and other contracts. The weight in each edge denotes the number of times edges were formed – when no weight is specified, the interaction happened once. For example, contract 9703 was called 16 times in the subgraph, 3 times with account 32643. As a result of these calls, the contract transferred ether to other accounts, for instance, it transferred ether to account 17305 four times. Notice that in the complete graph, there is no contract without at least one incoming edge, omitted in this graph for simplicity.

2.3 Partitioning methods

We formulate the problem of graph partitioning as follows: Given a graph $G = (V, E)$ and a number of partitions k, a partition p_i with $1 \leq i \leq k$ is a subset of V such that $\bigcup p_i = V$ and $\bigcap p_i = \emptyset$. That is, the graph is partitioned in k partitions and partitions are disjunct.

The problem of balanced graph partitioning is NP-Complete [34] and despite being studied for long (e.g., VLSI designs [46]), there are several ways to define what constitutes a good partitioning. Intuitively, a good partitioning is defined as one that minimizes the edges connecting two partitions (edge-cut) while maintaining each partition balanced.

More formally, let $C(p_i)$ be the set of edges that connect vertices in p_i with vertices in $V \setminus p_i$.

$$edge\text{-}cut = \frac{\sum_{i=1}^{k} |C(p_i)|}{|E|} \tag{2.1}$$

$$balance = \frac{max_{1 \leq i \leq k}(|C(p_i)|) \times k}{|V|} \tag{2.2}$$

From Eq. 2.1 we get the edge-cut percentage and from Eq. 2.2, the relation between the most unbalanced partition and the others. For example, if $k = 2$ and $edge\text{-}cut$ and $balance$ are, respectively, 0.2 and 1.3, then 20% of the edges are across partitions and one partition has 30% more vertices than the average. Ideally, $edge\text{-}cut$ is 0 and $balance$ is 1.

With Eqs. 2.1 and 2.2, only static aspects of the graph are taken into account. We can enrich the graph by assigning weights to vertices and edges to capture the frequency that accounts, contracts, and their interactions appear in the blockchain. By assigning weights to the edges, we can try to avoid cutting frequently used edges; by assigning weights to the vertices, we can better balance the load in the system. We refer to Eqs. 2.1 and 2.2 of a weighted graph as dynamic edge cut and dynamic balance, respectively. The dynamic edge cut and dynamic balance give us a more accurate view of the system's executed cross-shard transactions and load.

In the following we describe the five partitioning methods we used to partition Ethereum blockchain graph.

- *Hashing.* A straightforward way to partition the graph is to hash the vertex unique identifier and use the result (modulo the total number of shards k) to determine the shard the vertex belongs to. This is a common scheme to

shard data. Moreover, if the hash distribution is uniform, we can hope to obtain an optimum static balance among shards. The edge cut (number of multi-shard transactions), however, tends to increase with the number of shards. For instance, when $k = 8$ in our experiments, multi-shard transactions account for 88% of the total, despite the vertices being perfectly distributed.

- *Kernighan-Lin algorithm [48].* In this method, we periodically partition the system based on the transactions executed at each period. The system starts in some partitioned state (e.g., computed using hashing). Each shard identifies vertices that if moved to other shards would minimize edge-cuts based on the transactions executed in each period. Each shard sends to an oracle the selected vertices and with the information from all shards the oracle computes a $k \times k$ probability matrix. The oracle calculates the probability that each shard should move its selected vertices to the other shards so that at the end shards remain balanced. The oracle then sends the matrix to all the shards, which exchange vertices with each other based on the probability matrix. Intuitively, the algorithm tries to reduce dynamic edge-cuts while keeping the shards dynamically balanced. This approach has been used to partition large graphs [59].

- *METIS.* In this method, we periodically partition the graph with METIS [47]. To do so, we input METIS with the current graph (i.e., up to the moment of partitioning) and shard vertices based on METIS output. We configured METIS to minimize edge-cuts while keeping shards balanced. We aim to reduce dynamic edge-cuts by assigning weights to the edges of the graph. When an account (or contract) appears for the first time (or is created), it has to be assigned to some shard. This is done by inspecting all the accounts involved in the transaction and picking the shard that minimizes edge-cuts; if more than one exists, we maximize the balance. There is a subtle problem with this approach, as every time the graph is partitioned METIS can move vertices back and forth between shards, since it is not part of METIS objectives to minimize the number or vertices that change shard between successive executions of the algorithm.

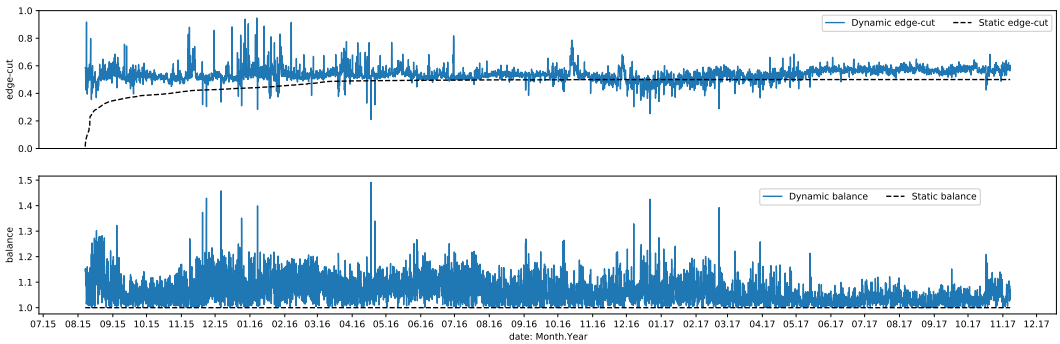
- *R-METIS.* While using the technique above with large graphs we realized that METIS sometimes does not produce good results due to the increasingly large size of the graph. We therefore revisit the previous technique by using as input for METIS a reduced graph. This graph contains all accounts, contracts, and their interactions within a fixed window of time (two

weeks), which starts at the last (re)partitioning.

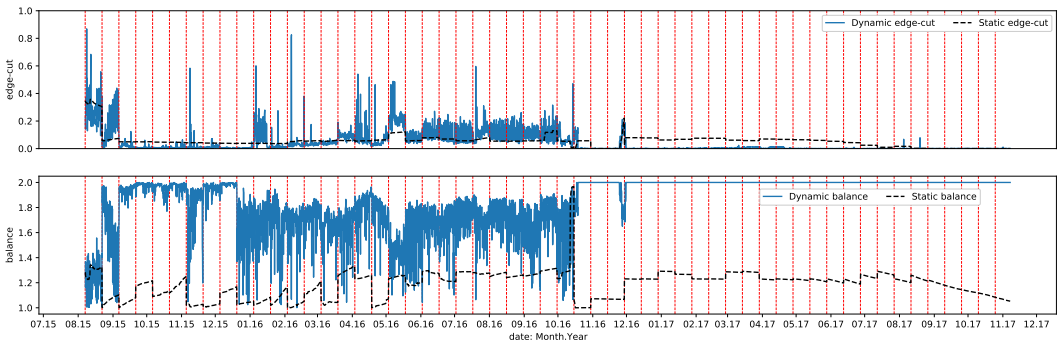
- *TR-METIS*. This is essentially the method above where instead of triggering a repartition at constant time intervals, we set a threshold on the dynamic edge-cut and dynamic balance. When the threshold is reached, we run METIS to compute a new partitioning, which will hopefully reduce the number of transactions across shards. The motivation for the technique is to reduce unnecessary repartitioning.

2.4 Results

We assessed the five partitioning methods described in the previous section in configurations with 2, 4 and 8 shards. In each case, we computed the following metrics: static and dynamic edge-cut and balance and the number of vertices that change shard after the graph is repartitioned (i.e., number of moves).



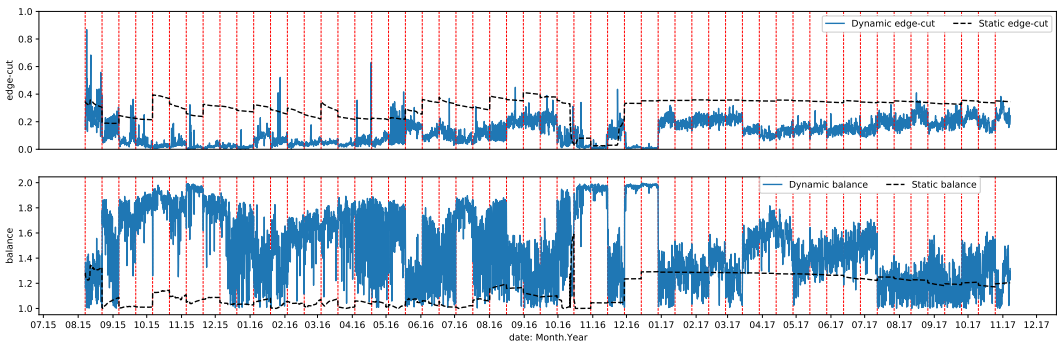
(a) Hashing



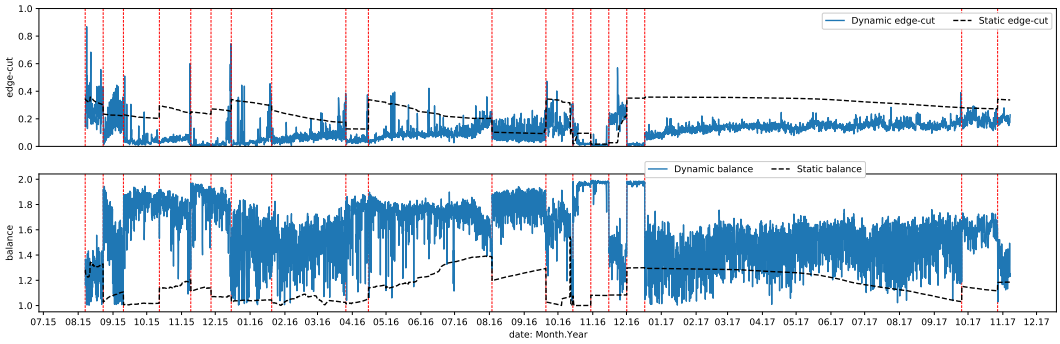
(b) METIS

Figure 2.3. Dynamic edge cut and balance of (a) hashing and (b) METIS when Ethereum is partitioned in two shards.

Figure 2.3 shows the results for hashing and METIS with two shards. Each data point corresponds to a four-hour window. Repartitioning takes place every two weeks in case of a periodic repartitioning, marked by vertical dashed lines in Figure 2.3(b). Hashing provides optimum static balance since each shard is assigned an equal number of vertices. This does not mean, however, that the load experienced by the shards is the same (see dynamic balance). With respect to static edge-cuts, with two shards hashing leads to about 50% of transactions across shards. METIS provides a much lower edge-cut, both static and dynamic, at the expense of dynamic imbalance among shards. Notice that dynamic balance is near two. This happens because vertices in one shard are much more "active" than vertices in the other shard, even though both shards contain similar number of vertices. The effect is particularly visible after the September 2016 attack, when a large number of dummy accounts were created.



(a) R-METIS



(b) TR-METIS

Figure 2.4. Dynamic edge cut and balance of (a) R-METIS and (b) TR-METIS when Ethereum is partitioned in two shards.

In Figure 2.4 we see the edge-cut and balance over time using the R-METIS

and TR-METIS partitioning schemes. R-METIS achieves a better result over time when compared to METIS since it considers only a portion of the total graph, making the problem easier for METIS and at the same time forgetting about unused portions of the graph. We observe that periodic repartitionings do not alter the outcome of the sharding significatively. TR-METIS provides a comparable outcome to R-METIS but having less repartitioning and thus less objects moving across shards.

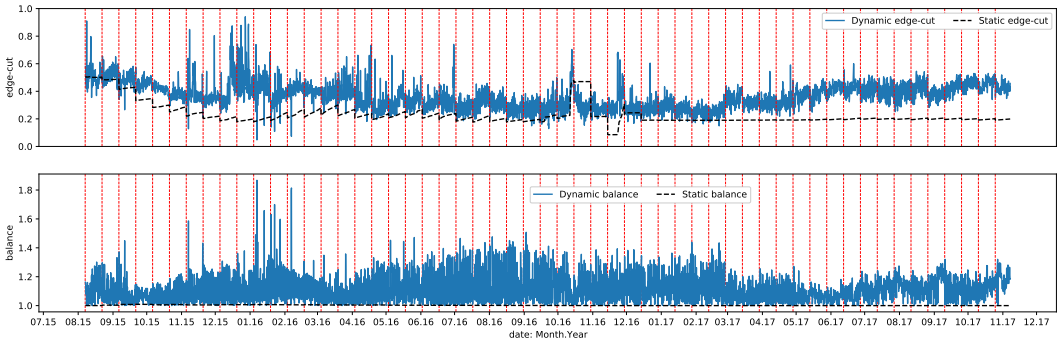
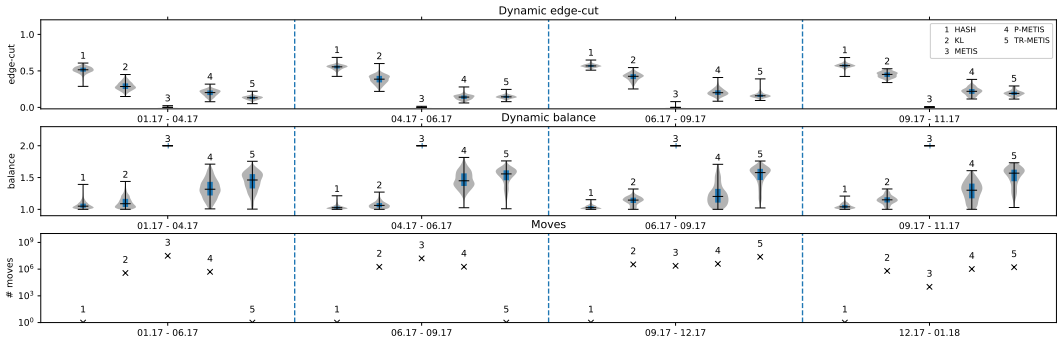
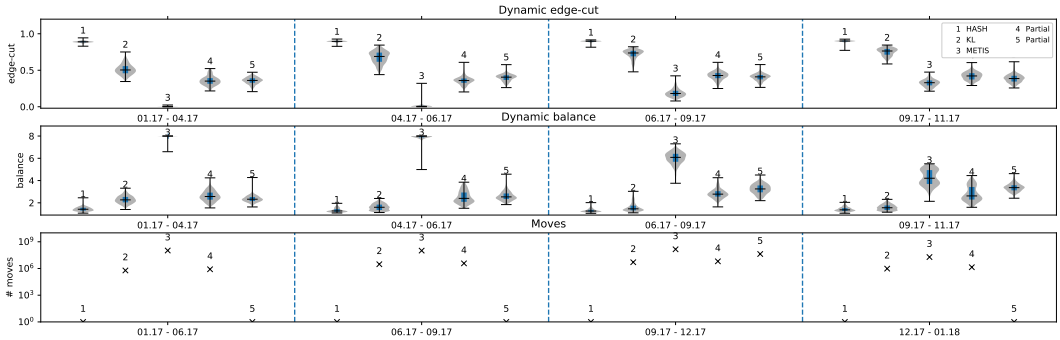


Figure 2.5. The dynamic edge cut and balance of Kernighan-Lin when Ethereum is partitioned in two shards.

In Figure 2.5 we see how the Kernighan-Lin method for partitioning the Ethereum graph performs over time. Since vertices can only swap places with other vertices, the graph remains balanced throughout the experiment. Differently from the METIS algorithms, the Kernighan-Lin method could be done in a distributed setup and does not need a centralized component to function. Furthermore, observe that the same optimization heuristics done for TR-METIS could be done for this experiment.



(a) Configurations with 2 shards



(b) Configurations with 8 shards

Figure 2.6. Box-and-whisker with violin plot (density plot) for five partitioning methods using Ethereum transactions in 2017 (whiskers show minimum and maximum values, bottom and top of the box show first and third quartiles, and the band inside the box shows the median).

Fig. 2.6 shows results for the five partition methods in configurations with 2 and 8 shards, using transactions executed in 2017, which represents the bulk of transactions in Ethereum. We show dynamic edge-cut, dynamic balance, and total number of moves in the period.

We draw the following conclusions from these results.

- There is a clear compromise between edge-cut and balance, and no technique clearly stands out. This suggests that *based on the workload at the time of writing, it is unlikely that Ethereum can be partitioned in such a way as to produce both low edge-cut and good balance among shards*.

- Within a configuration (i.e., 2 or 8 shards), the behavior of the various partition methods does not change substantially over time, although it does

change from one configuration to the other. We take a closer look at this aspect at the end of this section.

- Hashing leads to a fair dynamic balance among shards, albeit at the cost of a large number of dynamic edge-cuts. Interestingly, hashing does not outperform KL with respect to dynamic balance, but performs consistently worse than all other techniques for dynamic edge-cuts. There are no moves since partitioning depends on vertex id only and once assigned to a shard a vertex remains in the assigned shard.

- KL reduces dynamic edge-cuts while maintaining shards balanced. The various iterations of the technique lead to a large number of vertices changing shards. One pitfall of this method is that partitioning optimizes for a local minima.

- METIS trades balance for edge-cuts. However, as previously discussed, this is mostly an anomaly that resulted from the October 2016 attack, when a large number of dummy accounts were created and never used again. Although METIS statically balances the graph, one shard contains most meaningful accounts and transactions execute in a single shard.

- R-METIS considers just the subgraph formed after a repartitioning. This helps ignore vertices created and never used again. Vertices only used once create an artificial balance among shards. With this technique we managed to get a lower dynamic balance. This technique resulted in a more dynamically balanced system.

- TR-METIS improves on the previous technique by reducing the number of vertices moved across shards. This essentially happens because we trigger a repartitioning based on edge-cut and balance values. We adjust thresholds to trigger a repartitioning in such a way that the performance does not diverge much from the previous technique. The result is a dramatic decrease in the number of moved vertices, without compromising edge-cuts and balance, when compared to R-METIS.

Note that reducing the number of accounts and contracts that change shards after a repartitioning of the system is important for performance. If we were to move one vertex from one shard to another, we ought to move the entire state of the vertex. If the vertex is a contract, that would result in moving the entire contract storage to another shard.

Figure 2.7. Dynamic edge-cut, dynamic balance and total number of moves of various techniques with increasing number of shards.

Fig 2.7 compares partitioning techniques with respect to dynamic edge-cut and balance and total number of moved vertices while varying the number of shards. In these executions we used data from the beginning of Ethereum up to the end of 2017. In order to show balance for different configurations in the same graph we normalized the results with the number of shards (i.e., $balance\ value - 1/number\ of\ shards - 1$). As before, low values mean better balanced shards.

In all techniques, dynamic edge-cut becomes worse as the number of shards increases (top graph), although METIS-based techniques outperform hashing and KL. With respect to dynamic balance, Hashing and KL perform better than techniques based on METIS. Hashing and METIS, however, take extreme ends in the balance versus edge-cut tradeoff. The number of moves is large in the METIS algorithm, since the partitioner algorithm does not optimize for this aspect. R-METIS and TR-METIS techniques perform substantially fewer moves because they use a smaller graph.

2.5 Final remarks

This study sheds some light on what one could expect from sharding Ethereum. The results show that even with fairly sophisticated partitioning methods, there is a clear tradeoff between edge-cuts and balance. This is important since scalable performance requires low edge-cut and balanced partitioning.

The results come with some caveats. We assess Ethereum using the real workload, which was not created for a sharded system. It is possible that if Ethereum is ever extended with the ability to handle sharding then applications will be designed in a different way. If sharding is made visible to developers, then multi-

shard operations could be sometimes avoided, at the expense of more complex applications.

When sharding a blockchain, multiple incentives have to be taken into account. In case of a generic framework such as Ethereum, there are three main components that need to be addressed: computation, storage and bandwidth [20]. All of these components play an important role in partitioning. For instance, moving state indiscriminately will have both an impact in the bandwidth and storage of the system. Designing the correct incentives is crucial to a good partitioning scheme.

In our analysis we considered accounts to be sharded, but a simple technique to split one's account in different shards could help improve the number of cut edges. Alternatively, one could move all involved contracts of a transaction to a single shard and execute the transaction on that shard. A method for moving smart contracts is necessary for this operation to work. Furthermore, contracts can use the same operation to adapt to changes in the workload. We develop these concepts and ideas in the next chapter.

Chapter 3

A framework and protocol to scale blockchain throughput

In this chapter, we propose a new method to scale blockchains by providing a primitive operation to enable sharded blockchains to communicate or communication from different blockchains, often called Inter-Blockchain Communication (IBC). We provide a framework in which blockchains can communicate. We target blockchains that hold state within smart contracts and provide a way to move a smart contract from one blockchain to another.

Our framework and protocol allow smart contracts to migrate to different blockchains or shards by using a *move* operation. In our framework, smart contract developers have a great deal of flexibility and can define policies for contract's movements. Flexibility is achieved by reducing transparency – exposing information about other shards or blockchains. To exemplify how smart contracts could be developed using our framework, we modify two popular blockchain applications in Ethereum and show how their throughout can scale.

In the following, we provide the background needed to understand the proposed framework (Section 3.1), state the assumptions needed to support the proposed protocol (Section 3.2), introduce the general idea (Section 3.3), present the protocol operation in detail (Section 3.4), and discuss extensions to the basic protocol proposed in this chapter (Section 3.5).

3.1 Background

Blockchains can grow large in size and complexity. For instance, Ethereum has over three terabytes of log at the time of this writing and it can take several weeks to re-execute all appended transactions. Clients with low storage or computa-

tional power can succinctly prove the validity of an arbitrary piece of the state [65] without re-executing the entire log, provided they maintain and verify all the block headers.

Blockchain systems typically rely on a Merkle-tree or similar data structure to provide data integrity checks. For example, Bitcoin uses a binary Merkle-tree [65], while Tendermint uses a modified AVL tree [44]. For the sake of simplicity, we call these structures "Merkle-trees". Each block header includes the root of a Merkle-tree (i.e., Merkle-root). Data is encoded on the leaves of the Merkle-tree, and parent nodes are labeled with the cryptographic hash of child nodes grouped together, compacting the structure until a unique Merkle-root is reached. The objective of such data structure is to provide a computationally and spatially cheap way to prove the integrity of leaves of the state without necessarily having all the state.

Merkle-trees allow for a peer to hold only block headers and forego downloading all the blockchain state. Peers can ask for a proved piece of partial state (Merkle-proof) at a specific block height from peers that have the state at the requested block height or happen to have the same Merkle-proof. The information provided by these peers can be checked with the Merkle-root stored in the trusted block header. We denote the unique path of a valid Merkle-proof from object v to Merkle-root m as $\{v\} \mapsto m$. The leaf v and nodes $h \in (\{v\} \mapsto m)$ needed to reconstruct the proof must be given to verify the validity of the proof. The verification of Merkle-proofs can be done optimally in logarithmic time and space on the number of nodes of the tree [65].

Figure 3.1 illustrates how Merkle-proofs can prune parts of the tree logarithmically. Blocks from b_0 to b_n are shown linked together in the top. From block b_1 we see the Merkle-proof for $\{v\} \mapsto m$. Given a hash function H, m is b_1's Merkle-root composed by h_0 and h_1 hashes. In the figure, we see the result of asking for the Merkle-proof of v. Anyone can compute the Merkle-root of this proof and accept it only if it is equal to the trusted Merkle-root m stored in b_1. Observe that only v and hatched nodes h_0 and h_3 are needed to verify m.

3.2 Assumptions

We make the assumptions listed next in order for blockchains to support the move operation. Although these assumptions are not strictly necessary to move objects across blockchains, they simplify the Move protocol.

In particular, we assume that blockchains must:

(a) support smart contracts (i.e., arbitrary computation);

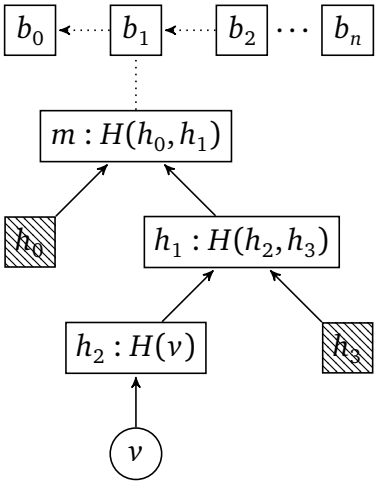


Figure 3.1. Merkle-proof example, h_0, $h_3 \in (\{v\} \mapsto m)$.

(b) use the same execution environment (i.e., virtual machine); and

(c) provide a succinct way to prove state variables (e.g., using a Merkle-tree).

Supporting smart contracts that can execute arbitrary computation allows us to investigate more complex and generic use cases for the protocol. When moved to the target blockchain, we assume that smart contracts can execute the same opcode instructions they executed in the source blockchain, i.e., they use the same execution environment. This simplifies how communicating blockchains are capable to understand each other.

Clients can have information about the Merkle-root of any other blockchain by downloading the correspondent block header. Clients can listen to headers from multiple blockchains all at once. Block headers have a constant size of usually hundreds of bytes and are on average a small fraction of block bodies. For example, in Ethereum block headers are around 2% of the block's body. Moreover, blockchains willing to support the Move protocol must agree on certain configured parameters discussed in Section 3.7.1.

3.3 Overview

The state of a contract is presumably indivisible and must reside as a whole in a blockchain. Accounts and smart contracts are restrained to live in a single blockchain at a time. Therefore, we must ensure that if a contract moves from one blockchain (source) to another (target), it will no longer be "active" in the source

blockchain. When the contract becomes active in the target blockchain, its state must be identical to its state when it became inactive in the source blockchain. This implies that the move operation involving two blockchains must be atomic.

One way to implement an atomic Move operation is to resort to the well-known two-phase commit (2PC) protocol [39], or one of its more resilient variations (e.g., [38]). We refrain from using a 2PC-like Move protocol since it would introduce expensive coordination between the involved blockchains (e.g., members of each blockchain would have to exchange votes in a reliable manner). Instead, we use a two-step approach that divides the move operation into two transactions, *Move1* and *Move2*. In Move1, the state of a smart contract is "locked" in the source blockchain, after which it is guaranteed not to be changed—although transactions can still read the contents of the locked smart contract. In Move2, the smart contract is reconstructed in the target blockchain, after which it can be safely used. To avoid simple attack vectors, the Move2 transaction is only successful at the target blockchain if it contains a proof that the Move1 transaction was successfully executed at the source blockchain.

This two-step approach reduces coordination between the source and the destination blockchains, but it complicates atomicity. For example, moving a contract can remain unfinished if the client fails after submitting the Move1 transaction and before it submits the corresponding Move2 transaction. To account for such cases, we allow any client to execute the Move2 transaction, and thereby complete a possibly unfinished Move operation. In the normal case, however, we expect the same client to execute both transactions.

3.4 The Move protocol in detail

Algorithm 1 details the move operation of contract c from blockchain B_i to B_j. We add a new field to a contract state, referred to as L_c. L_c specifies in which blockchain the smart contract c currently resides. At low level, assigning a new value to L_c in Move1 is implemented with a new EVM opcode, OP_MOVE. The OP_MOVE opcode takes as argument the target blockchain identifier the smart contract is moving to (in this case B_j). When OP_MOVE is executed in c, it changes L_c to B_j, and by consequence blocks the contract state at B_i. Any transactions that try to alter the state of blocked contract c in B_i will be aborted.

Move2 assumes the existence of two boolean functions, V_S and V_P. $V_S(B, m)$ returns true if m is a valid Merkle-root in the blockchain B. $V_P(V \mapsto m)$ returns true if the state V of c is proved by $V \mapsto m$. The smart contract code and other blockchain specific variables (e.g., the amount of currency held by the smart

contract) are omitted in the algorithm but still need to be proved by $V \mapsto m$. Notice that before submitting a Move2 transaction for contract c, the client must acquire the proof $V \mapsto m$ at the source blockchain (discussed later).

The Move1 and Move2 transactions allow application developers to execute special routines when a contract is moved. We illustrate the use of this functionality in the next section.

Algorithm 1 The operations.

```
procedure MOVE1(c, B_j)                    ▷ Move c in B_i to B_j, executed at B_i
    moveTo(·)                              ▷ Execute custom function
    L_c ← B_j                              ▷ Block contract c in B_i
procedure MOVE2(c, V ↦ m)                  ▷ Complete move of c, execute at B_j
    if L_c ≠ B_j then                      ▷ Is c being moved to the wrong blockchain?
        return abort
    if V_S(B_i, m) = false then            ▷ Invalid Merkle-root
        return abort
    if V_P(V ↦ m) = false then             ▷ Invalid proof
        return abort
    for all v ∈ V do
        Call SSTORE(v.key v.value)         ▷ Recreate storage in B_j
    return moveFinish(·)                   ▷ Execute custom function
```

3.4.1 A concrete implementation

We have integrated the Move operation in the Solidity programming language [64]. In our prototype, smart contract developers must implement two functions to allow contracts to move, $moveTo(\cdot)$ and $moveFinish(\cdot)$ (see Algorithm 1). This provides the application developer a great deal of flexibility. For example, in Listing 3.2 we have an excerpt of Solidity code that in few lines ensures that only the contract's owner is allowed to move the contract and the contract must remain at least three days in the target blockchain before being moved again.

```
address owner;
uint movedAt;
  function moveTo(uint _blockchainId) public {
    require(owner == msg.sender, "only owner can move");
    require(now - movedAt >= 3 days, "contract moved less than 3 days ago");
  }
  function moveFinish() public {
    movedAt = now;
  }
```

Figure 3.2. Excerpt of a movable contract in Solidity.

3.4.2 Preventing replay attacks

If a client executes transaction T_{move1} at blockchain B_i to move contract c to B_j, any client can craft a special transaction T_{move2} to be executed in blockchain B_j that reconstructs the state of c in B_j. In T_{move2} the client appends the state of contract c encoded as V in the Merkle-proof $V \mapsto m$. Target blockchain B_j is responsible for verifying that m is accepted by the blockchain as a valid Merkle-root of B_i. Nodes in blockchain B_j verify the correctness of c's state by verifying $V \mapsto m$ and B_i's state root hash. If the proofs are valid, c's state can be safely reconstructed in B_j.

Additional measures should be taken to prevent replay attacks. The attack consists in a (malicious) client crafting a T_{move2} transaction that uses an old state of contract c. The replayed transaction would lead to inconsistencies as transactions that followed the first (and thus legitimate) Move2 transaction would be lost. One remedy would be to have nodes store the contract's nonce, a monotonically increasing number that is increased every time the contract is invoked. For instance, in Figure 3.3 a contract is moved from B_1 to B_2 (transactions T_{move1} and T_{move2}, respectively) and afterwards back to B_1 (transactions $T_{move1'}$ and $T_{move2'}$). It starts with nonce equal to zero and as soon as T_{move2} is executed in B_2 it increments the nonce by one. Afterwards $T_{move1'}(B_1)$ completes in B_2 and changes the nonce to three. When client$_2$ tries to replay transaction T_{move2}, the contract's nonce is one which is less than what was previously seen by B_2 and the transaction aborts.

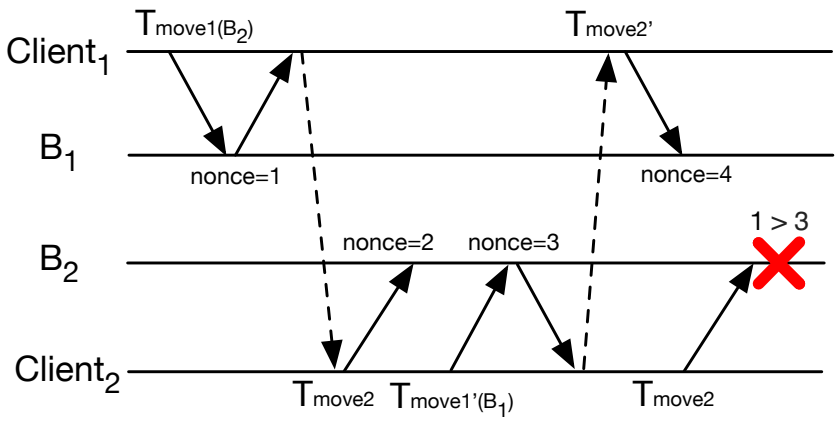


Figure 3.3. Preventing replay on stale data.

3.5 Handling currencies

Most cryptocurrencies rely on an internal currency for generating incentives to maintain the system (e.g., paying miners and transaction fees). In some cases, the currency is the main usage of the system (e.g., Bitcoin). It is important therefore to have a way of transferring this part of the system's state from one blockchain to another. As it turns out, we can devise a simple mechanism that uses our protocol to accomplish the feat. It suffices for smart contracts to be allowed to hold currency in their state. We can then implement smart contract "relays" that can transfer currency from blockchains by creating a token in the target blockchain that is provably locked in the source blockchain, similar to Pegged Side Chains [7]. Currencies can be unlocked when contracts return to the original blockchain.

Assume for example that we would like to transfer e units of currency from $client_1$ to $client_2$, from blockchain B_i to blockchain B_j. We assume the existence of contract c in B_i, that when called by client $client_1$ with input B_j, $client_2$ and value associated e creates a contract r that has e units of currency and lets $client_2$ withdraw e from its state (i.e., it executes Move1(B_j) on creation). The $client_2$ can call T_{move2} on r effectively moving it to B_j where the funds would be available as a B_i's token in B_j.

In Figure 3.4, we can see an example where a client successfully transfers a currency token from blockchain B_1 to a token representation in blockchain B_2. Contract c's function $create$ is called with e units of B_1's associated currency with T_{create}. The transaction creates contract r with e units of currency, seen in the

figure as "$". Afterwards, the same function calls r's $moveTo(B_2)$ changing r's L_c to B_2. The newly created contract has functions to generate tokens which are proved to be backed by e in B_1. The $client_2$ waits for the transaction inclusion in B_1 and sends transaction T_{move2} to B_2, proving that r was moved to B_2. After transaction T_{move2} is included in B_2, $client_2$ calls transaction T_{mint} which executes code in r creating tokens in B_2 that represent the locked coins in B_1.

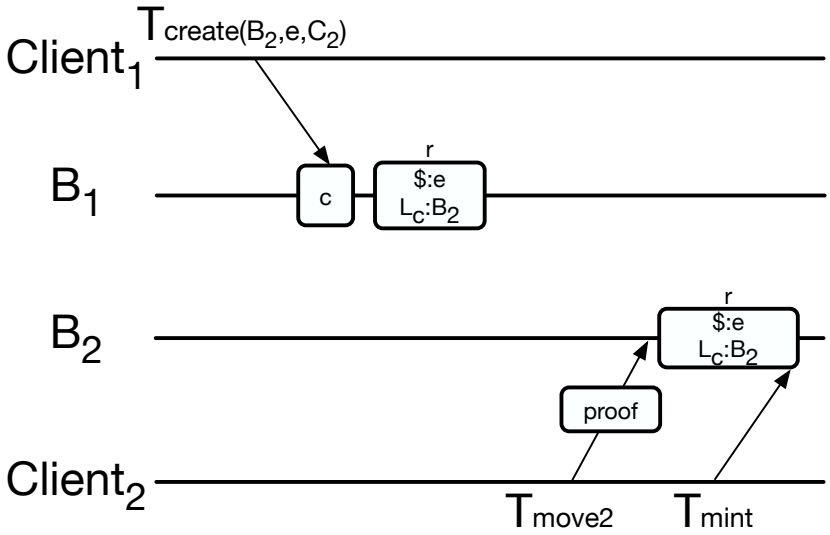


Figure 3.4. Move operation example.

3.6 Additional details

Account identifiers. Although each blockchain maintains its own set of accounts, its identifier can be the same if the interacting blockchains use the same rule to derive such identifiers. Consequently, clients could use the same cryptographic keys to use accounts in different blockchains. It becomes essential to incorporate the blockchain's identification to functions that compute contract addresses to ensure a unique system-wide contract identification to avoid collisions in contract identifiers.

Finding contracts. L_c can only have two logical states: either in the current blockchain or transferred (executed Move1), and if $moveTo(\cdot)$ and $moveFinish(\cdot)$ are implemented correctly, it can always go back to the first state with Move2.

A client who does not know where contract c is located can use L_c to track the contract's location every time it moves.

Stale data. Every time a contract is moved it leaves behind a stale state on the original blockchain, which could be garbage collected, paying attention to guard against the attack previously described. Designing fee incentives to clean the state is left as future work.

3.7 Applications

The Move protocol can be used by blockchains to implement two important blockchain concepts: interoperability and sharding. Interoperability and sharding are related concepts that have increased in importance as blockchain technology matures.

3.7.1 Interoperability

Blockchains coexist nowadays with different transaction designs, cryptographic features, and trust assumptions in a heterogeneous environment. Interoperability provides a way to offload transactions from one blockchain to another, unleashing the potential to scale different applications and experiment with different combinations of blockchains.

Interoperability in permissionless systems is challenging mainly because forks can occur since block propagation time is unbounded [30], which invalidates transactions that build on the losing side of the fork. A way for systems proposing interoperability [7, 72, 51] to interact with permissionless blockchains is by introducing a parameter p that specifies the minimum number of blocks that a transaction's block should be behind the blockchain's head for it to be accepted by the other blockchain. The parameter can be configured according to each blockchain involved in the interoperability protocol.

Miners or validators of blockchains willing to interoperate should maintain a light client that validates Merkle-roots of other blockchains. Proposals for this scheme are discussed in Section 3.12.

3.7.2 Sharding

Sharding enables the blockchain state to be divided into shards responsible for holding a certain portion of the state. Sharding preserves some of the blockchain

assumptions, but there are clear trade-offs in security because the members themselves have to be sharded. The way objects are assigned to each shard plays an important role when sharding a blockchain for scalability. For instance, if objects are randomly partitioned into shards, most of the transactions will likely be cross-shard. The rate of cross-shard transactions also increases with the number of shards, and sharding the state while minimizing the number of cross-shard transactions keeping the various shards balanced is a hard problem as seen previously in Chapter 2.

To cope with changes in load it becomes essential to incorporate a method to move state from shards, offloading one shard in detriment of another. As shards get congested and fees increase, users are tempted to move their contracts to underused shards.

3.8 Use cases

We implemented two applications for smart contracts and show how they scale with the number of blockchains:

- *SCoin*: A token smart contract based on a popular Ethereum token interface.
- *ScalableKitties*: A clone of *CryptoKitties*, a popular Ethereum application where virtual cats can migrate and reproduce in different shards.

3.8.1 SCoin

```
contract STokenI {
  function totalSupply() public view returns (uint);
  function newAccount() public payable returns (AccountI, uint);
  function newAccountFor(address forAddr) public payable returns (AccountI,
      uint);
  event CreatedAccount(address account, uint salt);
}

contract AccountI {
  function balance() public view returns (uint);
  function allowance(address spender) public view returns (uint);
  function transfer(AccountI to, uint tokens) public returns (bool);
  function approve(address spender, uint tokens) public returns (bool);
  function transferFrom(AccountI to, uint tokens) public returns (bool);
  function debit(uint tokens, bytes proof) public returns (bool);
```

```
    function moveTo(uint shardId) public;
    function moveFinish() public;
    event Transfer(address to, uint tokens);
    event Approval(address spender, uint tokens);
  }
```

Listing 3.1. Scalable Token interfaces extending ERC20.

ERC20 [33] is a standard interface widely used in Ethereum for token operations including token transfers. *STokenI* and *AccountI*, defined in Listing 3.1, are interfaces that support all ERC20 operations and allow for contracts to move from one blockchain to another. The main idea of the interface is to use one instance of *AccountI* per user account. Typical ERC20 implementations hold token balances in a map data structure, which multiple blockchains cannot share in our design since we do not allow for contracts to live in two or more blockchains at the same time.

Once created, accounts can freely move from within blockchains using the *moveTo* and *moveFinish* functions. It is left to the developer to restrict or even define a policy for moving accounts between blockchains.

To illustrate the *STokenI* interface, we implement *SCoin*, a scalable token contract that implements *STokenI*. *SCoin* creates instances of *SAccount*, which implements *AccountI*. The implementation of *SCoin* and most functions of *SAccount* are straightforward and application-dependent. We focus next on how to do safe transfers between one *SAccount* to another. To execute a transaction that transfers e tokens from *SAccount* A to B, contract A has to decrease e from its state (called by *transfer* function) and B has to increase e. This is done by calling the *debit* function in B. If contracts A and B are in different blockchains, they have to be first moved to the same blockchain to be able to call each other. Once both contracts are in the same blockchain, how can A know that B is what it claims to be? For instance, one could design a contract B that, when *debit* is called, increases the contract's tokens by an arbitrary amount. A could ask its parent if B was created by the same contract, but A's parent contract might be in a different blockchain. The interface does not specify how contract A can be sure of contract B's origin. It is up to the developer to devise safeguards to prevent incorrect usage. The key idea for *SCoin* is holding a proof in B that it was created by the same contract that created A.

When we create an instance of *SAccount* in *SCoin* it uses a monotonically increasing salt, stored in the instance state. The salt is used to calculate the identifiers of both contracts using the *create2* opcode [71]. With A's salt, B can attest that A was created by the same contract that created B and vice-versa. To execute a transfer from *SAccount* A to B, contract A attests B's origin, decrements

its own balance and calls the function $debit(\cdot)$ in B. Contract B agrees to debit its own account only if A passes the same check, and A can safely add B's fund to its own balance. The checks of origin are done with one inexpensive hash operation. In our implementation case, we take advantage of the way contract identifiers are generated in the EVM. A more generic method could be devised using Merkle proofs with the same proposed interfaces.

3.8.2 ScalableKitties

ScalableKitties is a clone of *CryptoKitties*, a popular Ethereum smart contract that was created in November 23th 2017 and until the writing of this thesis had over four million related transactions. During the apex of its popularity, *CryptoKitties* congested the Ethereum network for several days, accounting for over fifteen percent of all Ethereum transactions [74]. In *CryptoKitties*, cats are collectibles that can be bred to generate more cats following a set of rules (e.g., sibling cats cannot mate). Cats were first generated by the contract's owner. Both the initial and bred cats are sold in an auction smart contract. The game was the first mainstream application built on top of Ethereum, and some cats were sold for more than a hundred thousand dollars at the time. The contract is still actively used today.

We mapped the *ScalableKitties* functions one-to-one to the *CryptoKitties* smart contract but for simplicity we discuss only the functions that are related to the execution of cross-blockchain transactions. Cats are created in two ways: either by having the contract's owner calling a function to generate "promotional" cats or by breeding two cats to generate a third. Breeding is the only operation that can generate cross-blockchain transactions because bred cats can be in different blockchains and need to be moved to the same one. Furthermore, if the owner of cat A wants to breed A with B, it either needs to own both cats or B's owner has to permit B to be sired with A. In the experiments in Section 3.10.1 we replay transactions from the *CryptoKitties* contract on the *ScalableKitties* contract.

3.9 Deployment

We modify Hyperledger Burrow [43] and Ethereum [36] to implement the protocol defined in Section 3.4. The resulting systems allow blockchains to communicate with each other (IBC) or implement sharding. To validate our approach, we conducted two types of experiments: we shard Burrow and analyze how applications can scale performance, and make smart contracts migrate from Ethereum

to Burrow and vice-versa to assess the performance and monetary costs of IBC. Both Burrow and Ethereum implement the EVM model, where each opcode executed by the smart contracts has a cost modeled in *gas* (e.g., a sum between two integers costs 3 gas, while creating a new smart contract costs at least 32000 gas [71]).

Burrow uses Tendermint for consensus, which by design introduces the application's Merkle-root from block n in block $n+1$, but in Burrow the state of block n is saved only in block $n+1$, therefore there is a need to wait for two blocks to prove the transaction inclusion required by Move2. For the experiments we set p (defined in Section 3.7.1) equal to two blocks in Burrow, since clients have no option other than waiting for two blocks to get the proof of inclusion of a Move1 transaction. For Ethereum we set p to six blocks.

Tendermint is configured to wait for five seconds between each consecutive block, the observed latency being slightly higher than this. Ethereum is configured to wait 15 seconds, which is similar to Ethereum's main public network.

All experiments were conducted using a heterogeneous cluster in a local area network with simulated latencies between nodes based on the values published in [27], where the authors evaluate nodes in 14 regions in four continents on Amazon data centers. We emulate this environment in a cluster of servers and randomly allocate nodes to regions.

3.10 Sharding experiments

We evaluate the Move protocol using the two applications described in Section 3.8. The objective of this experiment is to show how the capacity to move smart contracts can significantly improve the performance of the applications. In all sharding experiments, one node hosts all clients and maintains one connection per shard to broadcast the client's transactions. We decided to run one validator in each node and 10 validators per shard due to a limitation of our cluster size, comprised of 80 computers, each one with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8Gb of memory, SATA SSD disks, and 1Gbps ethernet card. With this configuration, we can run a maximum of 8 shards and we decided to run experiments with 1 (no sharding), 2, 4 and 8 shards. To decide which shard to send the contracts to when needed, we use hash partitioning where the contract's shard is decided by the hash of the contract's identification. Using hash partitioning ensures a good balance among shards but implies probably a higher cross-shard rate the more shards there are (see Chapter 2).

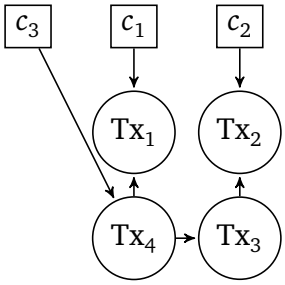


Figure 3.5. Dependency graph example.

3.10.1 ScalableKitties

In order to produce the data for experiments we scanned all transactions involving the *CryptoKitties* contract deployed [1] in Ethereum since its inception.

Since our aim is to replay all transactions from the original smart contract, every transaction that executes in our implementation must happen as it happened in the original contract (e.g. if a transaction succeeded in the original smart contract it must also succeed in our implementation). For this premise to hold we first construct a dependency DAG (Directed Acyclic Graph) of all the transactions and execute them respecting their dependencies. By doing so we are able to execute some of the transactions in parallel. for instance, consider a client that owns cat c_1 and wants to breed its cat with another user's cat c_2. To execute this transaction, the transaction graph of Figure 3.5 needs to be respected. First cats c_1 and c_2 have to be created with Tx_1 and Tx_2, respectively, then c_2's owner agrees with the breeding with Tx_3 and finally Tx_4 breeds c_1 and c_2, creating c_3. Vertices c_1, c_2, and c_3 are pointers to transaction vertices that have a dependency on c_1, c_2, and c_3, respectively. Notice that leaf transactions in the DAG can be executed in parallel. For instance, Tx_1 and Tx_2 can be executed in parallel but Tx_4 can only execute when it becomes a leaf, i.e., both Tx_1 and Tx_3 finish.

We replay *CryptoKitties* transactions on *ScalableKitties* in a sharded environment running multiple instances of the Burrow client. We use the dependency DAG described previously to replay transactions to the contract. Transactions that are appended in a block (executed) are removed from the DAG. Subsequent transactions that become leaves in the DAG are submitted until a limit of 250 outstanding transactions is reached. We pre-process the whole DAG in memory, broadcasting the first transactions, updating the DAG, and possibly sending other transactions whose dependencies are satisfied. The process continues until the

[1] At address 0x06012c8cf97bead5deae237070f9587f8e7a266d

experiment is over. Increasing the number of shards leads to an increase in the number of cross-shard transactions, that in turn reduce the number of leaves in the DAG since cross-shard transactions depend at least on two transactions: Move1 and Move2.

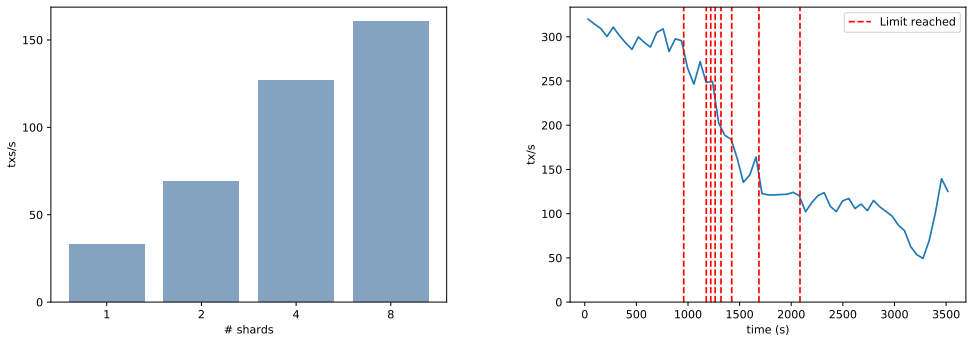


Figure 3.6. ScalableKitties throughput for 2, 4 and 8 shards (left), and aggregated throughput over time for 8 shards (right).

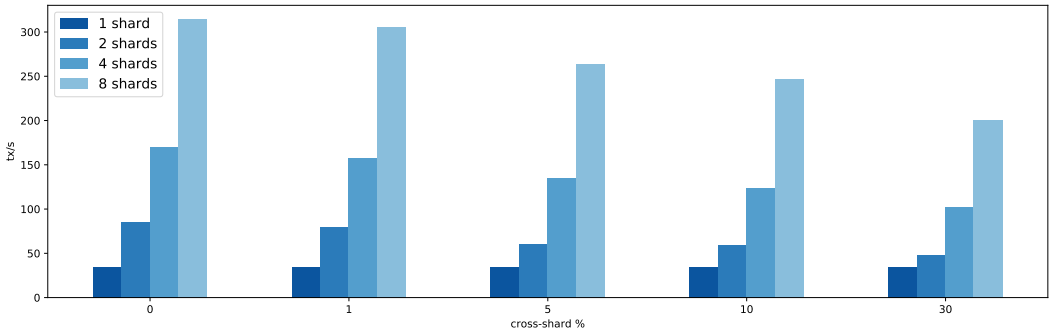


Figure 3.7. Performance with varying number of shards and different cross-shard transaction rates.

Figure 3.6 (left) shows a nearly linear increase in the average number of transactions per second as we increase the number of shards, except when there are eight shards. The reason the throughput with eight shards is lower than expected is that there were not enough ready-to-run transactions in the dependency graph, making the client wait for blocked transactions to finish. This is better visualized in Figure 3.6 (right), where vertical dashed lines mark the point when each one of the eight shards had less outgoing transactions than configured at the beginning of the experiment. A better partitioning could increase the number of

transactions that can run in parallel, since the number of cross-shard transactions would be minimized.

To better understand how varying cross-shard transaction rates can affect performance we conduct experiments in the next section that confirm the performance gains obtained with *ScalableKitties*.

3.10.2 SCoin

We now present results for the SCoin application defined in Section 3.8.1. In these experiments, we benchmark how well the protocol can perform with a single application with varying number of transactions that require cross-shard communication (i.e., tokens transferred between different partitions). We measure latency and throughput tradeoffs with a varying number of cross-shard transactions.

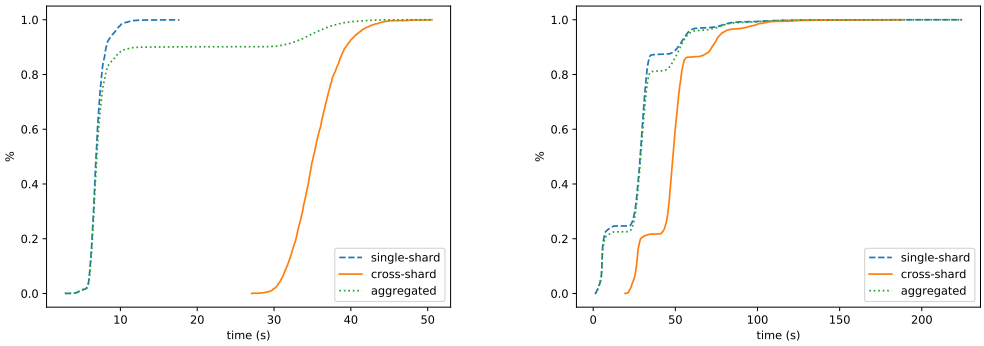
Each client in the experiment tries to execute the *transfer* transactions in a closed-loop. If the transaction is cross-shard, i.e., if a client tries to transfer its token to an account that resides in a different shard, the client first moves its account to the corresponding shard and then executes the *transfer* transaction afterward in the destination shard. Similarly to the previous experiment, we tune the number of clients in the system to avoid significant degradation of the average latency for each client, thus caping each shard to 250 clients. We experiment with different cross-shard transaction rates for different shard numbers.

Figure 3.7 shows the aggregate throughput of the system for a varying number of shards and different percentages of cross-shard transactions. The one shard experiment is shown in every cross-shard rate experiment as a reference. We can see how performance degrades when increasing the number of cross-shard transaction rates, but the throughput grows linearly at different rates of cross-shard. For comparison, the previous experiment had on average 5.86%, 7.93%, 7.85% cross-blockchain transaction rate for 2, 4 and 8 shards, respectively. Any system that reserves part of its allocated resources to process cross-shard transactions will have similar behavior, due to cross-shard transactions occupying the resources that otherwise would be used by single-shard transactions.

Retries

In the previous experiments, to better control the rate of cross-shard transactions, clients submit transactions only if they know the contracts will not be moved when the transaction is executed. To better model transactions in the presence of conflicts we experiment with the *SCoin* contract without any help

from external sources, e.g. an oracle that keeps track and answer queries for the location of smart contracts. Two situations can make clients retry transactions: when performing a single-shard transaction and the interacted contract is moved to another shard or when performing a cross-shard transaction and the called contract is moved to another shard. In the experiment we make clients wait a random time corresponding to the creation of 0 to 10 blocks if the transaction fails for any of these reasons, this is done to prevent contracts moving back and forth in an endless cycle.



(a) Without retries (b) With retries

Figure 3.8. Latency CDF for 4 shards experiment with 10% cross-shard transactions.

The average latency for clients does not change significantly when increasing the number of shards, remaining at around 7 seconds for single-shard transactions and 34 seconds for cross-shard transactions when transactions do not retry. Cross-shard transactions demand two transactions for each move operation, plus waiting for two blocks to prove the contract's state and one final transaction to complete the operation, confirming the expected latency of waiting for five blocks per cross-shard transaction. In Figure 3.8 we can see the cumulative distribution function (CDF) for clients' observed latencies in a scenario with four shards and 10% cross-shard transactions rate. The aggregated latency in Figure 3.8 (a) shows both single and cross-shard transactions when transactions do not retry. We can see that, as expected, around 10% of the transactions take more than 30 seconds to complete.

Figure 3.8 (b) shows the latency when conflicts can happen and transactions sometimes retry. A clear increase in latency is observed when comparing both figures, but when retries can happen the number of times the same transaction fails and has to retry is highly skewed. For instance, 66% of the transactions that

retry, do it just once, and only 1% of these transactions are retried more than three times. Differently from other sharded systems (e.g., [61]) the protocol does not suffer from a *convoy effect* [4], that is, cross-shard transactions do not delay single-shard transactions.

3.11 IBC experiments

This section presents some experiments for inter-blockchain communication (IBC) considering Burrow/Tendermint and Ethereum. These experiments aim to measure the time and gas (which translates to cryptocurrency costs) consumed for operations with five different applications:

- *SCoin*: Transfer one token from one blockchain to another and transfer the virtual currency to another account in the target blockchain.
- *ScalableKitties*: Transfer one virtual cat from one blockchain to another, breed the cat with an existing cat in the target blockchain, and give birth to another cat.
- State 1, State 10, State 100: Transfer the state containing respectively 1, 10 and 100 32-byte state variables from one blockchain to another.

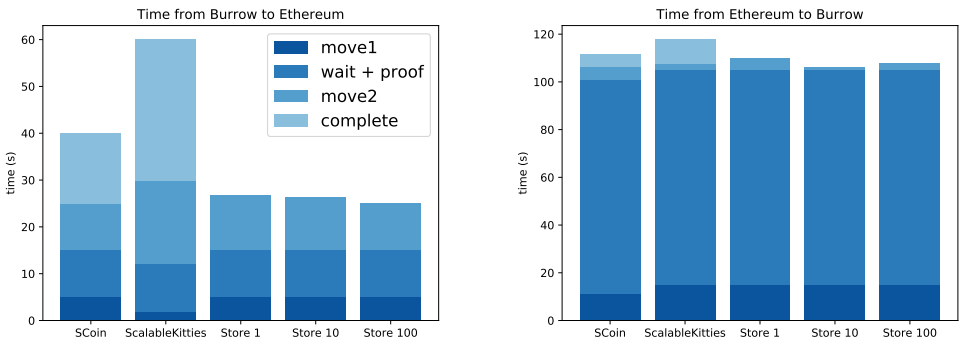


Figure 3.9. Latency for five different inter-blockchain applications.

These operations require moving their corresponding smart contract from the source to the target blockchain, an operation requiring two transactions (Move1 and Move2) and the wait for p blocks in-between transactions. After that, further transactions might need to be executed in the target blockchain. In our examples,

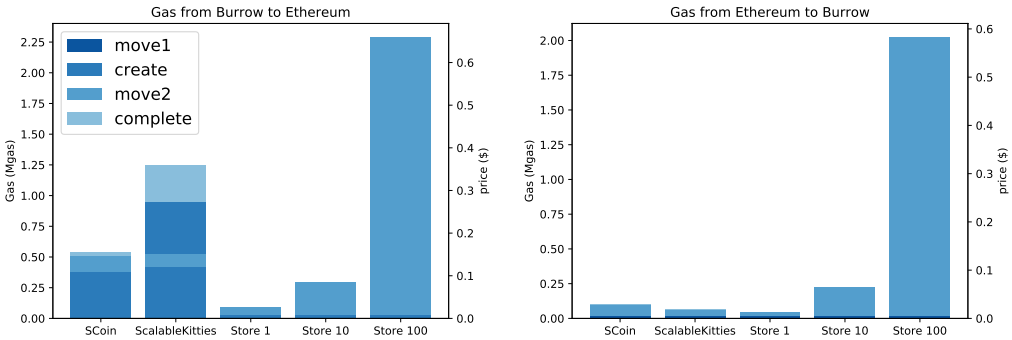


Figure 3.10. Gas and monetary costs for five different inter-blockchain applications.

SCoin requires one further operation to transfer the token to a contract in the target blockchain, while *ScalableKitties* requires transactions *breed* and *giveBirth* to mate and produce a new virtual cat, respectively. The state transfer experiments do not require any further transactions for completion.

Figure 3.9 presents the time required to perform an operation from Ethereum to Burrow and vice-versa. Unsurprisingly, the time to perform a single transaction is bound to the latency between consecutive blocks in each blockchain. To execute Move2 from Ethereum to Burrow, one needs to wait for 6 Ethereum blocks, which translates to approximately 90 seconds and ends up dominating the overall time for every operation.

A good way to analyze the impact of each operation on the system's performance is to measure the gas consumed by each operation. The gas cost is expected to grow linearly with the size of the transferred smart contract state, since creating or modifying state variables are expensive operations in the EVM model. Figure 3.10 shows in the left y-axis the amount of gas paid by each transaction. To better understand these costs, in the right y-axis we present the same costs in US dollars, considering the current average value of one gas as two Gwei (2×10^{-9} Eth) and one Eth as $144 (the price in the middle of December of 2019).

The way Burrow and Ethereum calculate gas prices is different. Besides the actual values per operation being different, some operations pay no gas. For example, in Ethereum if one smart contract or transaction creates a new smart contract it pays an amount of gas per byte of the contract code that is being created, while in Burrow no gas is paid per byte of code. In both systems, the code is immutable and stored in the state Merkle-tree with the key based on the

code's hash.

In Figure 3.10, the vertical hatched bars represent the gas paid by the creation of new contracts in Ethereum. Every recreated contract pays a constant gas based on the size of the moved code. In the *ScalableKitties* application, the *giveBirth* function creates a new contract, thus it pays for the gas again. For *SCoin* and *ScalableKitties*, the gas paid for the code creation corresponds to around 70% of the total gas cost. We note that it is possible to reduce significantly the Ethereum contract creation costs if the contract code is already in the blockchain. This could be done with a small optimization in Ethereum that forfeits the contract's creation costs if the code is already stored in the tree.

3.12 Related work

Scaling blockchains is a hot topic for both industry and academia, and many proposals have appeared in the last years (e.g., [35], [70]). In this chapter, we showed that scalability can be achieved with a novel IBC protocol and sharding. A different approach to scaling blockchains is to shift computation from the blockchain (on-chain) to the outside (off-chain). For example, in the method proposed by TrueBit [66] most of the computation is done off-chain. Cheating participants can be caught by using an on-chain game in which anyone can prove they misbehaved in logarithmic time. Another example of a scaling solution is the lightning network [58], which works on top of Bitcoin, and its Ethereum counterpart, Raiden [60]. Users of such systems create off-chain channels between them in order to minimize on-chain transactions. The efficiency of such systems is highly application-dependent.

Pegged Sidechains [7] focus on transferring assets between proof-of-work blockchains. The idea is to lock assets in one blockchain and recreate them in another blockchain by providing a proof of such locking in the original blockchain. The Move protocol generalizes Pegged Sidechains in several aspects. First, it allows to transfer any state across blockchain systems, not only assets. Second, it applies to both proof-of-stake and proof-of-work approaches; third, the Move protocol shifts control to the application developer, who can develop scalable applications with their own logic (e.g., introducing load balancing mechanisms).

In HyperService [53], the authors create a new programming language and system to make blockchain applications interoperable. HyperService orchestrates the execution of applications that span multiple blockchains. We take a different approach: a smart contract is executed in one blockchain, after the smart contract dependencies are moved to the same blockchain.

In [40], atomic token swaps between multiple blockchains are studied and proposed as a protocol that, similar to ours, is done in two phases. As noted by the author, "atomic cross-chain swap is an atomic cross-chain transaction, but not vice-versa". Our protocol could be used to implement atomic swaps in a similar way as shown in Section 3.5, although a more efficient solution for performing token swaps with more than two blockchains, combining our protocol with the techniques proposed in [40] would be interesting future work. In [1] the authors propose a solution for permissioned ledgers where blockchains implement a common "relay" mechanism for cross-blockchain transactions with smart contracts. The protocol provides a great deal of flexibility but it requires smart contract developers to have a deep understanding of the underlying cross-chain protocol. Without allowing contract's state to migrate, blockchain systems risk having their performance limited by cross-blockchain transaction performance.

PolkaDot [72] aims to create a decentralized *federation* of blockchains by allowing other blockchains (called para-chains) to exist. Existing blockchains can be interfaced by *relay-chains*, but no details are given on how the validation happens on existing blockchains. Similar to our work, blockchains in PolkaDot need well-defined parameters so they can interoperate, e.g., the number of blocks to wait to accept the transaction as being final. Cosmos [51] also aims to provide IBC by allowing multiple blockchains, called *zones*, to communicate with each other. All Cosmos zones run the Tendermint algorithm for consensus. One zone, called *Cosmos hub* acts as a central communication interface for all other zones. Interledger [67] is a proposed protocol for IBC that tackles payments. To achieve safety and liveness, transactions are either escrowed by notaries that run PBFT [19] or use incentives on rational actors. The Interledger approach does not integrate with existing blockchains and is constrained to simple token transfers.

In Omniledger [50], the authors developed a protocol that can process transactions across shards, called Atomix, built on top of Bitcoin's UTXO model. In Atomix, clients can lock their input and are left with the burden of unlocking them in case the transaction aborts. The authors briefly discuss applying Atomix to smart contracts and suggest that Atomix is suitable for scenarios where clients execute simple operations and transactions do not conflict. Our approach exposes IBC primitives to developers to give them more freedom to programmatically condition cross-blockchain operations. Similar to Omniledger, Elastico [54] focus on Bitcoin's UTXO model and there is no need for cross-shard transactions because outputs are assumed to be disjointedly distributed to shards.

Chainspace [5] builds on top of BFT-SMaRt [9], an open source BFT consensus implementation written in Java. One of the subsystems of Chainspace is

S-BAC, a two-phase commit protocol that deals with cross-shard transactions. In our model, we expose sharding primitives that let developers programmatically control the shard designation of smart contracts. Doing so simplifies the protocol and allows for a more organic distribution of objects. Differently from S-BAC, our protocol does not implement aborts and it is the developer's responsibility to avoid contracts stuck on the first phase of the protocol. Chainspace provides helpful insights for further works on sharding smart contracts that could be also applied in the presented protocol (e.g., having shard checkpoints can be beneficial to reduce the bandwidth and storage cost of moving a contract) Similarly, the efficient shard state transfer protocol described in [56] can alleviate this costs.

A protocol with similar goals as ours has been proposed in the Ethereum research forum concurrently with the development of this work [15]. It describes a similar operation to move the state from one shard to another, called *yanking*. Other threads in the same forum further extend the proposed idea, but until the writing of this thesis, it remains a work in progress tailored specifically for the Ethereum environment, and constrained by its own design choices.

3.13 Final remarks

In this chapter, we presented a practical protocol that can be used to develop smart contracts that can move between different blockchains or shard a single blockchain. Our protocol enables smart contract developers to create blockchain applications that interoperate and scale in an ecosystem of multiple blockchains. We developed two applications for our protocol and extensively evaluated their performance and tradeoffs. The simplicity of our protocol opens up possibilities for further improvements, such as decentralized load balancing smart contracts for sharded blockchains.

We explored how to improve throughput scalability by proposing a method in which smart contracts can interact with each other even if they are in different blockchains. Our approach sacrifices transparency and exposes the concept of multiple inter-operating blockchains to developers.

Chapter 4

Optimizing state synchronization

Efficient state synchronization is critical to the operation of blockchain systems. By state synchronization, we refer to the process by which a new peer catches up with the state of other operational peers. In this chapter, we provide data structures and algorithms for robust and fast blockchain state synchronization. By robust, we mean that our solution can tolerate Byzantine failures. By fast, we mean: (i) that validation and state reconstruction can be performed quickly, and (ii) that a peer does not have to pause operation in order to compute a snapshot of the state. The key component of our solution is the design of a novel data structure we call the AVL* tree, targeted specifically to the state synchronization use case.

The rest of the chapter is organized as follows. We first provide the necessary background (Section 4.1). We then describe the basic structure and operations of our AVL* tree (Section 4.2), followed by algorithms that enable fast state synchronization (Section 4.10). We then provide a thorough evaluation of AVL* trees, comparing to IAVL+ tree (Section 4.12). Finally, we present related work (Section 4.13) and conclude (Section 4.14).

4.1 Background

State-of-the-art. In theory, a new peer joining the network could download the entire blockchain, and reconstruct the state by re-executing every transaction that appears in the log. However, this is impractical, since the number of transactions grows too large over time. Therefore, existing blockchain-based systems, such as Ethereum [16], and Cosmos/Tendermint [51], rely on downloading some recent state rather than executing all transactions to reach the same state. Some systems build snapshots of the state, where a snapshot is a serialized represen-

tation of the blockchain data structures. When a new peer joins the blockchain, it downloads the blockchain blocks with transactions (or block headers, with a summary of the block) and the snapshot. The peer then installs the snapshot and replays all transactions since the snapshot was taken, to reconstruct the current system state.

To decrease the time it takes for a new peer to join the blockchain, a snapshot could be divided into *chunks* as in [57], each chunk containing multiple parts of the state. This reduces the overhead of transferring individual state units, while allowing new peers to download chunks from many peers concurrently. If the state is organized in a tree, a natural strategy to assign tree nodes to chunks would be to traverse the tree (e.g., using depth-first search) and build fixed-sized chunks [26]. As we will discuss using this technique, however, the method in which tree nodes are assigned to chunks can have a significant impact on system behavior and performance.

The fact that the state is stored in a Merkle tree allows a peer to validate the consistency of the state by computing a hash on the reconstructed tree. Recall that a Merkle-tree is a tree in which every leaf node stores a cryptographic hash of its value, and every non-leaf node stores the hash of its children, and every block header in the blockchain stores the hash of the tree root constructed by the transactions in a previous block (e.g., block n contains the hash of the tree root computed with transactions in block $n-1$). To validate a snapshot, a peer simply computes the hash on its tree and compares the computed value with the value stored in the trusted block header.

Problem. Unfortunately, this approach to state synchronization suffers from two subtle, but important problems.

The first problem is due to the relationship between chunks and validation. If nodes of the state tree are assigned to chunks in a naïve way—as described above—then the chunks cannot be validated independently. Instead, a peer must download the entire tree before it can compute the hash. This introduces an effective attack vector for misbehaving peers. If a single malicious peer shares an invalid chunk, the new peer cannot identify the problem until it has downloaded all of the chunks. This wastes the resources of both the sender and receiver, in terms of I/O to the network and disk, and can significantly prolong the time it takes for a new peer to join the blockchain. To sidestep this issue, some systems strengthen their assumptions about honest peers in the network (e.g., [51]), which is not satisfactory since it weakens the trust model.

The second problem is with performance. When a peer computes a snap-

shot, it needs to search the tree, serialize all nodes, build chunks, and save them to local storage. In section 4.12, we show that existing blockchain-based systems based on this approach experience periodic hiccups, dropping transaction throughput.

Our approach. The AVL* tree is a Merklized AVL tree in which the tree's leaves are organized into chunks. However, the assignment of nodes to chunks ensures that each chunk always contains a sub-tree of the system state which can be verified independently. This enables batches of leaves to be securely and efficiently downloaded concurrently, while also permitting chunks to be verified for integrity using a compact proof. Finally, the structure of the tree is such that after a transaction is executed, a peer need only to recompute the hash of the affected chunk, and propagate the hash up the tree to the root. It does not need to recompute the entire snapshot. Thus, during normal operation, the peer never has to pause transaction processing.

Challenges. Incorporating leaf batching into a Merkle-ized AVL tree might seem straightforward. In reality, the problem introduces several challenges. First among these is identifying the proper invariants that must be maintained so that the integrity of the chunks can be checked independently. Second is designing the non-trivial changes to the AVL tree's operation methods to preserve the invariants. Third, during state transfer, peers will download different chunks in parallel, and can start reconstructing the tree. In general, inserting the data into a tree can result in different trees. To ensure correctness, we need to guarantee that the tree construction algorithm deterministically builds the same tree on different peers.

Implementation and evaluation. We have implemented the AVL* tree data structure and integrated it into the Tendermint blockchain middleware. Our evaluation shows that during operation, the transaction throughput is consistently higher without pauses for snapshot construction. Moreover, the time it takes for a new peer to join the blockchain is halved, while at the same time tolerating Byzantine peers.

4.1.1 System overview

The techniques presented in this chapter are generally applicable to any blockchain-based system, as they all share a similar high-level design (i.e., state machine

replication). However, to serve as a concrete example for the exposition, and to provide a comparison for prototype implementation and evaluation, we focus on the Tendermint/Cosmos code base in detail.

First, to provide context, we present an overview of the Tendermint architecture, illustrated in Figure 4.1. Tendermint is a framework for building distributed applications that are replicated across many machines. Tendermint consists of two main components: (i) Tendermint Core and (ii) an API known as the Application BlockChain Interface (ABCI).

Application. Tendermint is agnostic to the particular application logic. It could be used to develop distributed key-value stores, à la Zookeeper [42], etcd [31], or consul [24]. Or, it could be used to build cryptocurrencies like Bitcoin and Ethereum.

ABCI. The application interacts with the core functionality of Tendermint through the Application BlockChain Interface (ABCI). The ABCI provides methods for submitting transactions; receiving a callback when a transaction is delivered or committed; and—most relevant to state synchronization —serving snapshots or receiving chunks of a snapshot from a peer.

Tendermint Core. Tendermint Core includes three major components. First, the Mempool component provides a mechanism for clients to propagate transactions to the peers, i.e., a type of reliable broadcast using a gossip protocol. It also provides the means for the application to check whether transactions are well-formed. Second, the Consensus Engine is responsible for ordering transactions and delivering them to the application, and signaling the commit of a block. Finally, the State Sync component allows new peers to join the blockchain by either replaying all transactions or fetching periodic snapshots of the system state.

4.1.2 Merkle trees

The Merkle-tree of a blockchain is multi-version: a new tree is created for every new blockchain block, implemented with copy-on-write. When a new version of the tree is instantiated, the Merkle-root from the current version is copied to a new Merkle-root and made the Merkle-root of the new version. Modified nodes are duplicated and saved under the new tree. Although it is not common for peers to navigate on older versions of the tree, it is required for certain inter-blockchain communication protocols [45], where peers serve as relayers between

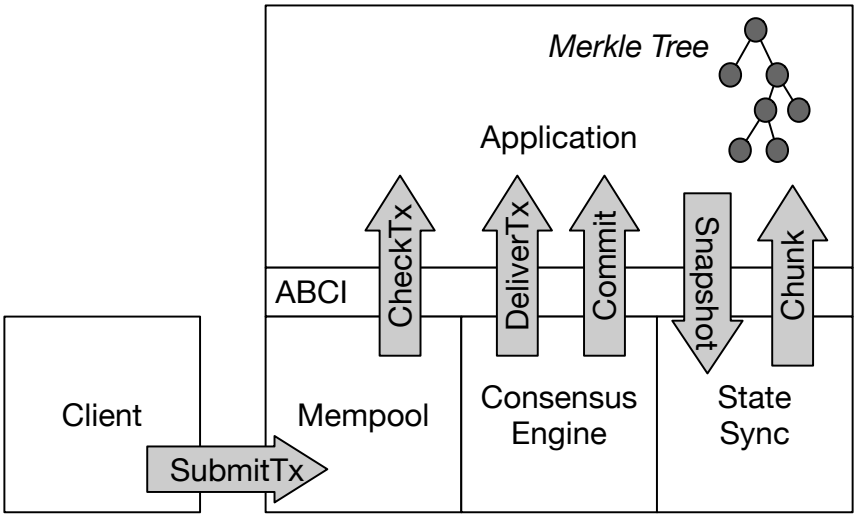


Figure 4.1. Overview of Tendermint architecture.

blockchains, allowing one blockchain to verify state proofs from another at a recent block.

4.1.3 State synchronization problem

A new (or resuming) blockchain peer needs to retrieve and execute transactions from all the blocks it has missed in order to catch up with operational peers. Merkle-trees enable faster catch up techniques, sometimes called *fast sync*. Instead of re-executing all missing transactions, the peer retrieves a recent version of the Merkle-tree (i.e., a snapshot) from other peers and applies only the transactions in blocks that succeed the retrieved Merkle-tree. This technique has been also implemented in popular blockchain clients (e.g., Geth [36], Ethereum's main client).

The obvious question, though, is how should a peer download the snapshot? On the one hand, since a snapshot can be large, the download process can be accelerated by partitioning the tree into chunks, and downloading chunks from several peers in parallel. On the other hand, in blockchain systems, leaf sizes tend to be small (e.g., a hundred bytes), and the number of leaves quite large, creating significant I/O overhead if one were to serve each leaf independently. It makes sense to batch leaves together to reduce the I/O overhead.

Existing systems [51] choose a fixed-size chunk, and assign nodes to the chunk until it is full. A natural implementation strategy is to perform a traversal of the tree (e.g., in depth-first order) and assign nodes to chunks. However, this

approach fails to capitalize on the key property of Merkle-trees: that subtrees can be validated independently. Consequently, the snapshot cannot be validated until a peer has downloaded the complete state. Put another way, the peer must download all chunks before it can check if they are valid. Consequently, a single misbehaving peer serving an invalid chunk can initiate a type of "denial of service" attack, significantly prolonging the time it takes for a new peer to join the blockchain.

In Section 4.2, we describe an alternative approach: we assign subtrees to chunks, so that each chunk can be independently checked for integrity. This allows the blockchain to gracefully handle Byzantine errors during state synchronization.

However, our design does involve a trade-off. Our algorithms do not guarantee a fixed chunk size, which means that we may produce a greater number of chunks than optimal for storing the tree. We evaluate this under-utilization in Section 4.12, and find that the overhead is acceptable.

4.1.4 IAVL+

While the techniques described in this chapter could be used with any Merkleized AVL [2] tree, we focus the discussion on Tendermint's immutable AVL tree (IAVL+) [44]. IAVL+ is a self-balanced ordered binary tree that implements a key-value storage API, where all values are stored in leaves and inner nodes store keys, used to keep the tree ordered. Leaves store the hash of values, and inner nodes the hash of their children and keys. As an immutable data structure, all updates are performed using copy-on-write. In the IAVL+ implementation, nodes have additional metadata used to calculate their hashes, omitted here for simplicity.

To add a key-value pair, the tree is traversed from its root until a leaf is found. Then, a new inner node is created, from which the found leaf and the new leaf (with the key-value to be included) will descend.

When the height difference between the left and right subtrees of an inner node is greater than one, the tree is said to be imbalanced. To re-balance the tree, one or two rotations are needed involving the lowest imbalanced subtree, whose root is called pivot node. Figures 4.2 (b) and (c) show imbalanced trees with inner node 20 as pivot. A left rotation at the pivot is enough if the subtree on the right of the pivot is higher than the subtree on the left of the pivot (Figure 4.2 (c)). But a right rotation at the right child of the pivot must happen first, if the left side of the pivot's right child is higher than the right side of the pivot's right child. This is what happens in Figure 4.2 (b) since the left side of inner node

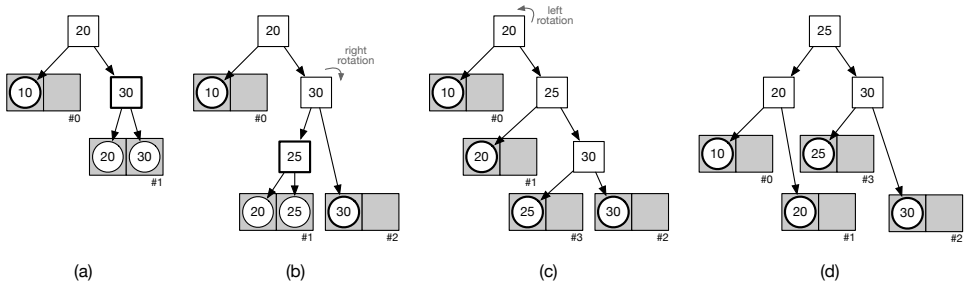


Figure 4.2. Insertion and rotations in AVL* (white square: inner node; white circle: leaf; gray rectangle: chunk; bold square/leaf: chunk root). (a) a balanced AVL tree, (b) an imbalanced AVL tree after the insertion of 25, (c) an imbalanced AVL tree after right rotation of previous tree at inner node 30, (d) a balanced AVL tree after left rotation of previous tree at pivot (inner node 20).

30, on the right of pivot 20, is higher than its right side. The cases for right and left-right rotations are symmetric.

4.2 The AVL* chunked tree

The AVL* tree is a Merkle-ized, balanced binary search tree. It is similar to Tendermint's IAVL+ and preserves the same invariant: the height difference between its left and right children is at most 1. The key difference between the two is that the leaf nodes in an AVL* tree are organized into *chunks*, or batches, that can be downloaded in parallel during state synchronization, and independently checked for integrity.

4.3 Chunks

A chunk is a set of nodes that are grouped together, and serves as the unit of access for the persistent store, communication, and integrity check. Grouping the nodes of the tree into batches facilitates parallel downloads. But, the assignment of nodes to chunks is important. If done incorrectly, it would not allow individual chunks to be checked for integrity. Each chunk has a *root* which is defined as follows:

Definition 1. *The root of a chunk C, $root(C)$, is the lowest (i.e. deepest) node that has all the nodes in C as descendants.*

Given this definition, the important invariant preserved by AVL* when assigning nodes to chunks is as follows:

Property 1. *For any chunks C_a and C_b, neither $root(C_a)$ is a descendant of $root(C_b)$ nor $root(C_b)$ is a descendant of $root(C_a)$.*

Property 1 ensures an overall minimal proof size for checking the integrity of a chunk. A client peer can check the integrity of chunk C with C and the proof that $root(C)$ is a valid node. If $root(C)$ is valid then all data it stores is valid, including the hash of its subtree. The client peer then computes the hash that should be stored in $root(C)$ from the leaves in C all the way up to $root(C)$ and then checks whether the computed hash matches the hash stored in $root(C)$.

Trivially, we see that this property would be satisfied if we were to assign every node to the same chunk, or assign each node to its own chunk, begging the question of how big a chunk should be? There is a trade-off: if a chunk is too large, then we lose the ability to download multiple chunks concurrently, but if a chunk is too small, we lose the benefits of batching. The chunk size is a constant set when the tree is instantiated. In our experiments, we empirically evaluate different chunk sizes.

The key challenge in designing the AVL* tree is updating the insertion and delete algorithms of the AVL tree to respect this invariant (e.g., despite tree rotations).

4.3.1 Data structures

In an AVL* tree, inner nodes are stored in volatile memory only (DRAM); leaves are embedded in chunks, which are stored in volatile memory and in a persistent store. Table 4.1 details the structure of inner nodes, leaf nodes, and chunks.

An inner node contains a *key*, used to search the tree; a pointer to a *chunk*, if the key is the root of the chunk; pointers to *left* and *right* nodes (inner or leaf); the *height* of the node; a *hash*, discussed below; and a boolean *is_leaf* that asserts that a pointer to the node references an inner node.[1]

A leaf node contains a key-value pair; a pointer to a *chunk*, if the key is the root of the chunk; an *i_node* pointer to an inner node, if the leaf's key is also an inner node; the *height* of the node, which is zero if the key is not stored as an inner node or the height of the inner node that stores the same key as the leaf; a *hash*; and a constant boolean *is_leaf*.

[1]For simplicity, we assume that pointers can reference either inner nodes or leaves.

Inner node	
key	unique item identifier
chunk	pointer to chunk, if chunk root
left	pointer to the left node, inner or leaf
right	pointer to the right node, inner or leaf
height	the height of the node
hash	needed by the Merkle-ized tree
is_leaf	false
Leaf node	
key	unique item identifier
value	arbitrary data held in the node
chunk	pointer to chunk, if chunk root
i_node	pointer to matching inner node, if any
height	0 or i_node height (when sending chunk)
hash	needed by the Merkle-ized tree
is_leaf	true
Chunk	
cid	unique chunk id, starting in 0
version	version number of the chunk
size	number of leaves in the chunk
root	pointer to the chunk root, inner or leaf
leaf[0..($C_p - 1$)]	set of leaf nodes in the chunk

Table 4.1. The data structures used in the AVL*.

A chunk contains a unique chunk identifier *cid*, a number in $0..(m-1)$, where m is the number of chunks; the *version* of the chunk, the number of leaves stored in the chunk, *size*; a pointer to the chunk *root*, which can be an inner node or a leaf; and a set *leaf* with all the leaves stored in the chunk. A chunk can store up to C_p leaves, a parameter of the system.

Hashes are computed after all changes in the tree have been performed, that is, the blockchain block the tree corresponds to has been fully processed. The hash of a leaf takes as input the key, value, height of the tree, and the chunk id and version, if the leaf is the chunk root. The hash of an inner node includes the key, the hashes of its children, and the chunk id and version, if the leaf is the chunk root.

4.4 Search

Searching for a key in the AVL* tree is just like a search in a regular binary tree. The only difference is that if the search traverses a part of the tree that is not

stored in main memory, then the corresponding chunk is read from disk and its entire subtree is stored in main memory.

4.5 Insertion

The insertion (see Algorithm 2) starts by searching down the tree for a leaf with a key that is immediately smaller or bigger than the key to be inserted. This will determine the position of the new leaf, either left or right of the found leaf, with the key-value element to be inserted. When searching down the tree, the root of a chunk is eventually found. From Property 1 (see Section 4.3), only one root chunk is traversed when inserting an element in the tree. If the root chunk points to a full chunk, before searching further, the chunk is split. The split, shown in Algorithm 3, does not change the structure of the tree but ensures that there is an available position in the chunk to accommodate the new leaf. To insert the new element, one inner node is created, pointing to the found leaf and the new leaf. When unrolling the recursion that leads to the insertion of an element, the height of every visited tree node is recomputed. If the subtree rooted at the visited node becomes imbalanced, a rotation is executed as defined in detail in Algorithm 5 and later in the text.

For example, in Figure 4.2 we add a node with key 25 in the tree depicted in (a), which has two chunks. Chunk #0 has one leaf and chunk #1 is full with two leaves. The insertion starts from the tree root and traverses to the right child at node 30, the chunk root of chunk #1, which is at maximum capacity, and splits it, creating chunk #2. The algorithm continues traversing the tree until leaf 20 is reached. Since the new key is bigger than the leaf's key, a new inner node is added copying the value of the new key and rearranging the leaves accordingly, resulting in the tree (b) before balancing is done.

To split a chunk, we create two new chunks and run a depth-first search (DFS) from the left and right children of the chunk root to be split. Each call to DFS builds a new chunk that is assigned to the left and right children of the old chunk root. At the end of the split the old chunk is deallocated and its chunk root set to ϵ.

Algorithm 2 Insert

```
next_cid := 0                                          ▷ next chunk identifier
procedure INSERT(node, key, val)                       ▷ node is a pointer to root
    return INSERT_AUX(node, key, val, ϵ)               ▷ return ptr to new node
procedure INSERT_AUX(node, key, val, last_chunk)
    if node = ϵ then                                   ▷ if the tree is empty:
        chunk := NEW_CHUNK(next_cid)                   ▷ first chunk
        node := INSERT_IN_CHUNK(chunk, key, val)
        node→chunk := chunk                            ▷ node becomes chunk root
        chunk→root :=node                              ▷ ditto
        return node
    if node→chunk ≠ ϵ then                             ▷ if node is the chunk root:
        c := node→chunk                                ▷ let c be this chunk
        if c →size = C_p then                          ▷ if c is a full chunk:
            SPLIT_CHUNK(node)                          ▷ split node (i.e., c's root)
    else
        c := last_chunk                                ▷ chunk root is a higher node
    if node→is_leaf then                               ▷ if node is a leaf:
        node_chunk := node→chunk                       ▷ keep its chunk, if any
        node→chunk := ϵ                                ▷ node can't be chunk root
        if key < node→key then                         ▷ normal AVL left insert
            left := INSERT_IN_CHUNK(c, key, val)
            node := NEW_INNER(node→key, left, node, 1)
        else                                           ▷ normal AVL right insert
            right := INSERT_IN_CHUNK(c, key, val)
            node := NEW_INNER(key, node, right, 1)
        node→right→inner_node := node                  ▷ leaf points to its inner
        node→chunk := node_chunk                       ▷ inner gets kept chunk
        if node→chunk ≠ ϵ then
            c→root := node
    else                                               ▷ if node is an inner node:
        if key < node→key then                         ▷ go down left or...
            node→left := INSERT_AUX(node→left, key, val, c)
        else                                           ▷ ...go down right...
            node→right := INSERT_AUX(node→right,key,val,c)
    UPDATE_HEIGHT(node)                                ▷ new height from children heights
    return BALANCE(node)                               ▷ if needed, rotate to keep balance
procedure UPDATE_HEIGHT(node)
    node→height :=                                     ▷ height gets max children height plus one
                max(node→left→height,node→right→height) + 1
```

Algorithm 3 Auxiliary procedures

```
procedure BALANCE(node)
    h := node→left→height - node→right→height
    if h < -1 then                    ▷ if right branch is bigger than left branch:
        return ROTATE_RL(node)                                  ▷ rotate [right] left
    if h > 1 then                     ▷ if left branch is bigger than right branch:
        return ROTATE_LR(node)                                  ▷ rotate [left] right
    return node
procedure NEW_INNER(key, ln, rn, ht)                            ▷ create inner node
    new_inode := allocate new inner node
    new_inode→key := key
    new_inode→left := ln
    new_inode→right := rn
    new_inode→height := ht
    return new_inode
procedure NEW_CHUNK(cid)                                        ▷ create chunk
    new_c := allocate new chunk
    new_c→cid := cid
    new_c→size := 0
    new_c→root := ϵ
    return new_c
procedure INSERT_IN_CHUNK(c, key, val)                          ▷ insert key,val
    c→leaf[c→size].key := key
    c→leaf[c→size].value := val
    node_addr := pointer to c→leaf[c→size]
    c→size := c→size + 1
    return node_addr
procedure DELETE_FROM_CHUNK(c, key)
    for node in c→leaf do
        if node.key = key then                                  ▷ found leaf
            SWAP(node, c→leaf[c→size])                          ▷ swap with the last
    deallocate c→leaf[c→size]                                   ▷ free last leaf
    c→size := c→size - 1                                        ▷ decrement number of leaves
    if c→size = 0 then                                          ▷ chunk is empty
        next_cid := next_cid - 1                                ▷ decrement number of chunks
        last_chunk := GET_CHUNK(next_cid)                       ▷ get chunk with highest id
        last_chunk→cid := c→cid                                 ▷ replace chunk's id
        deallocate c                                            ▷ free empty chunk
```

```
procedure SPLIT_CHUNK(node)                          ▷ split chunk rooted at node
    new_c := NEW_CHUNK(node→chunk→cid)
    DFS(node→left, node, new_c)
    node→left→chunk := new_c
    new_c→root := node→left                          ▷ assign left chunk
    next_cid := next_cid + 1
    new_c := NEW_CHUNK(next_cid)
    DFS(node→right, node, new_c)
    new_c→root := node→right                         ▷ assign right chunk
    node→right→chunk := new_c
    deallocate node→chunk
    node→chunk := ϵ                                  ▷ x is no longer chunk root
    return
procedure DFS(ptr, pnt, c)
    if ptr→is_leaf then
        c→leaf[c→size].key := ptr→key
        c→leaf[c→size].value := ptr→value
        c→leaf[c→size].i_node := ptr→i_node
        if ptr→key < pnt→key then                    ▷ assign parent
            pnt→left := pointer to chunk→leaf[c→size]
        else
            pnt→right := pointer to chunk→leaf[c→size]
        c→size := c→size + 1                         ▷ one more leaf in chunk
    else
        DFS(ptr→left, ptr, c)
        DFS(ptr→right, ptr, c)
    return
```

4.6 Deletion

Deleting a node from the AVL* tree is similar to deleting a node from an AVL tree. Algorithm 4 shows the full procedure to delete a key k. We start by searching the tree to find a pivot node p such that (i) p's left node is a leaf, or (ii) p is an inner node with key k. Then, there are four cases to consider:

- Case (a) (line 11 in Algorithm 4): p's left child is the leaf with key k. In this scenario, p's left node is deleted, p's right node takes the place of the pivot, and the original pivot is deleted. This case happens when we want to remove key 10 in Figure 4.2 (a), where the pivot is inner node 20.

- Case (b) (line 20): p's right child is the leaf with key k. In this case, p is necessarily the inner node with the key to be deleted. Both p and its right child are deleted and p's left child takes the place of the pivot. This case happens when we want to remove key 30 in Figure 4.2 (a), where the pivot is inner node 30.

- Case (c) (line 29): The left child of p's right child is the leaf with key k. The leaf with k is deleted, p is replaced with p's right child, and the original pivot is deleted. For this case, consider the deletion of key 20 in Figure 4.2 (a), where the pivot is inner node 20.

- Case (d) (line 43): Otherwise, we find the inner node x with the lowest value at the sub-tree on the right of p, delete x's left leaf, replace pivot p with x, and delete the inner node p. We illustrate this case with the deletion of key 20 in Figure 4.2 (b), where the pivot is inner node 20 and x is inner node 25.

Akin to the AVL-tree deletion, when unrolling from the recursion that found pivot p, the nodes involved in the deletion have their heights updated and possibly rotated (lines 70 and 71). Differently from the insertion, a deletion may involve rotations at all nodes in the way up to the root.

Deletion of a node in an AVL* tree differs from deletion in an AVL tree in two aspects. First, in cases (c) and (d) above, when an inner node x replaces a deleted inner node (i.e., the pivot). If x is a chunk root, then it will no longer be root, and the root of the chunk it referred to will be a node that descends from x.

Second, when deleting a leaf, it may happen that a chunk ends up with no leaves. This situation requires attention since the validity of a chunk is attested by the chunk root (discussed in Section 4.10), and an empty chunk has no root. To handle this case, when a chunk becomes empty, we take the chunk with the current largest unique id, assign to this chunk the id of the empty chunk, and decrement by one the number m of existing chunks (see Section 4.3.1). This ensures that every chunk with id in $0..(m-1)$ has at least one node, and therefore a chunk root.

Algorithm 4 Delete

```
procedure DELETE(node, key)
    if node.key = key and node.is_leaf then                ▷ only one node
        DELETE_FROM_CHUNK(node.chunk, key)
        return ϵ
    return DELETE_AUX(ϵ, node, key, ϵ)                     ▷ return ptr to new node

procedure DELETE_AUX(node, key, last_chunk)
    if node→chunk ≠ ϵ then                                 ▷ if node is the chunk root:
        c := node→chunk                                    ▷ let c be this chunk
    else
        c := last_chunk                                    ▷ chunk root is a higher node
    if node→left→key = key and node→left→is_leaf then
        if c = ϵ then                                      ▷ chunk is necessarily on the left
            c := node→left→chunk
        DELETE_FROM_CHUNK(c, key)
        promoted := node→right
        if node→chunk ≠ ϵ then
            promoted→chunk := node→chunk                   ▷ node is a chunk root
        deallocate node                                    ▷ no need for inner-node
        return promoted
    if node→key = key and node→right→is_leaf then
        if c = ϵ then                                      ▷ chunk is necessarily on the right
            c := node→right→chunk
        DELETE_FROM_CHUNK(c, key)
        promoted := node→left
        if node→chunk ≠ ϵ then
            promoted→chunk := node→chunk                   ▷ node is a chunk root
        deallocate node                                    ▷ no need for inner-node
        return promoted
    if node→key = key and node→right→left→is_leaf then
        if c = ϵ then                                      ▷ chunk is lower
            if node→right→chunk ≠ ϵ then                   ▷ chunk on right
                c := node→right→chunk
                node→rightNode→rightNode→chunk = c
                node→chunk = ϵ
            else
                c := node→right→left→chunk                 ▷ chunk on the left
```

```
        DELETE_FROM_CHUNK(c, key)
        aux := node→right
        node→key := aux→key
        node→right := aux→right
        deallocate aux
        return node
    if node→key = key then
        n, p := DELETE_LEAF(node→right, c)
        node→key = n→key
        node→right := p
        deallocate n
    else
        if key < node→key then
            node→left := DELETE_AUX(node→left, key, c)
        else
            node→right := DELETE_AUX(node→right, key, c)
    UPDATE_HEIGHT(node)                ▷ new height from children heights
    return BALANCE(node)               ▷ if needed, rotate to keep balance
procedure DELETE_LEAF(node, c)
    if node→chunk ≠ ϵ then             ▷ if node is the chunk root:
        c := node→chunk                ▷ let c be this chunk
    else
        c := last_chunk                ▷ chunk root is a higher node
    if node→left→is_leaf then          ▷ found leaf
        if c = ϵ then
            c := node→left→chunk
        DELETE_FROM_CHUNK(c, node→left→key)
        if node→chunk ≠ ϵ then
            node→right→chunk := node→chunk
        return node, node→right
    else
        n, new_root := DELETE_LEAF(node→left, c)
        node→left := new_root
        UPDATE_HEIGHT(node)            ▷ new height from children heights
        return n, BALANCE(node)        ▷ if needed, rotate to keep balance
```

4.7 Re-balancing

If, as a result of an insertion or a deletion, there is a height difference between two child subtrees, then the parent tree must be re-balanced, following Algorithm 5. Similar to the AVL tree, a rotation in an AVL* tree happens if the height difference from the children of a node is greater than one (Algorithm 3 line 1). There can be four types of rotations: right-left (line 1), left-right (line 8), left (line 15), and right (line 29). Left and right rotations happen with a single rotation to the left or right, respectively. When rotating to the left on a pivot, the right child of the pivot takes the place of the pivot (called new pivot), the right child of the old pivot becomes the left child of the new pivot, and the left child of the new pivot becomes the old pivot. For the right-left rotation, first the right child of the pivot is rotated to the right and finally the pivot is rotated to the left. The right and left-right rotations are symmetrical to the cases explained before. The tree can be balanced with at most two rotations after an insertion.

If the rotation involves a chunk root, then there are two cases to consider. First, if the root of the chunk is the pivot of the rotation, then the node that takes the position of the pivot becomes the new chunk root. Second, if the pivot of the rotation is not the root of a chunk, but the node that takes the position of the pivot is the root of a chunk, then we split the chunk before doing the rotation. This is done to ensure Property 1. As a consequence, there will be extra splits even when a chunk is not full; in our experimental evaluation these splits amount for around 4% of the total number of splits.

Continuing from the example in Figure 4.2, the tree (b) is imbalanced after the insertion. Since the height of the root's right child is greater than the height of the root's left child, a left or a right-left rotation is triggered. Since the root's right child has a higher height on its left child than the right one, we do a right-left rotation. First we rotate inner node 30 to the right and then rotate the inner node 20 (root) to the left. When rotating the inner node 30 to the right, we observe that its left child is a chunk root and split the chunk before continuing with the rotation. After this step, (the tree is depicted in (c)) observe that the tree has an extra chunk #3 created by the split and still is imbalanced. After the final rotation to the left, the tree is finally balanced as shown in (d).

Algorithm 5 Rotations

```
procedure ROTATE_RL(node)                          ▷ normal AVL [right] left rotation
    rl_height := node→right→left→height
    rr_height := node→right→right→height
    if rl_height > rr_height then
        node→right := ROTATE_R(node→right)
        UPDATE_HEIGHT(node)
    return ROTATE_L(node)

procedure ROTATE_LR(node)                          ▷ normal AVL [left] right rotation
    ll_height := node→left→left→height
    lr_height := node→left→right→height
    if ll_height < lr_height then
        node→left := ROTATE_L(node→left)
        UPDATE_HEIGHT(node)
    return ROTATE_R(node)

procedure ROTATE_L(node)                           ▷ node is the pivot
    if node→chunk ≠ ϵ then                         ▷ if subtree rooted at node in a chunk:
        node→chunk→root := node→right
        node→right→chunk := node→chunk             ▷ node's right child...
        node→chunk := ϵ                            ▷ ...becomes new chunk root
    else                                           ▷ else, rotation may involve two chunks
        if node→right→chunk ≠ ϵ then               ▷ if r child is chunk root:
            SPLIT_CHUNK(node→right)
    new_pivot := node→right                        ▷ rotation in three steps: one, ...
    node→right := new_pivot→left                   ▷ ...two, and...
    new_pivot→left := node                         ▷ ...three!
    UPDATE_HEIGHT(node)                            ▷ update moved node height
    UPDATE_HEIGHT(new_pivot)                       ▷ update moved node height
    return new_pivot

procedure ROTATE_R(node)                           ▷ node is the pivot
    if node→chunk ≠ ϵ then                         ▷ if subtree rooted at node in a chunk:
        node→left→chunk := node→chunk              ▷ node's left child...
        node→chunk := ϵ                            ▷ ...becomes new chunk root
    else                                           ▷ else, rotation may involve two chunks
        if node→left→chunk ≠ ϵ then                ▷ if l child is chunk root:
            SPLIT_CHUNK(node→left)
    new_pivot := node→left                         ▷ rotation in three steps: one, ...
    node→left := new_pivot→right                   ▷ ...two, and...
    new_pivot→right := node                        ▷ ...three!
    UPDATE_HEIGHT(node)                            ▷ update moved node height
    UPDATE_HEIGHT(new_pivot)                       ▷ update moved node height
    return new_pivot
```

4.8 Multi-versioning

An AVL* tree is multi-versioned, and a new version of the tree is created for every new blockchain block. For performance, a new tree is created using copy-on-write. Differently from the IAVL+ tree, in the AVL* tree the unit of allocation is a chunk. This means that when a node is modified in the new version of the tree, the complete chunk that contains the node is copied. This chunk-based allocation suggests a tradeoff: small chunks reduce the overhead of creating the new tree, but increase the number of chunks that need to be propagated upon state synchronization. We experimentally evaluate this tradeoff in Section 4.12.

4.9 Correctness

We initially argue that, in the absence of rotations, the insertion and deletion algorithms preserve Property 1.

In the case of an insertion, the property could only be invalidated when creating new chunks. When splitting a chunk, two new disjoint chunks are created and assigned to the left and right subtrees as the original chunk is extinguished. These steps do not violate Property 1 since they do not create descendants below or above each chunk root.

To see why deletion preserves Property 1, assume node x replaces the pivot, and x is root of its chunk. From the protocol, a descendant y of x becomes chunk root in place of x. Since x was chunk root, none of the nodes in its subtree are chunk root, and thus, no descendant of y is a chunk root.

We now argue that a right rotation preserves Property 1; the same reasoning applies for a left rotation. When rotating a node x to the right, x's left child is assigned as the right child of x's original left child. The only case a rotation would lead to a violation of Property 1 is when there are chunk roots in x's siblings. To deal with this case, before the assignment, we split the left chunk root in case of a right rotation (and right chunk root in case of a left rotation). After the split, the assigned node is a chunk root and can be safely moved to the opposite part of the tree.

4.10 Robust and fast state synchronization

The AVL* tree is designed to enable robust and fast state synchronization by allowing peers to exchange, reconstruct and validate chunks of the tree in parallel. However, simply batching the downloads of the state is not sufficient. A peer

must be able to re-construct the tree correctly. In other words, in general, there may be many ways to build a balanced binary tree from the same data. To be correct, we must ensure that all peers build the same tree. This is ensured if a peer collects all chunks that make up a tree, these chunks are valid, and the reconstruction of the tree is deterministic. More formally, the tree reconstruction algorithm ensures the following two properties:

Property 2. *Let T be the AVL* tree built by a honest peer after executing the n-th block of the blockchain, and $C_T = \{C_0, ..., C_{m-1}\}$ the chunks of T. Upon receiving chunk C_k from a peer, a client peer can check whether C_k is valid (i.e., $C_k \in C_T$) before receiving any other chunks in C_T.*

Property 3. *Let T and T' be two AVL* trees with the same nodes. If we build T' by inserting node by node following their order of height in T, then T and T' are isomorphic.*

Chunks in the IAVL+ do not necessarily contain subtrees, and thus they cannot be reconstructed or verified in parallel with the current implementation of the IAVL+ snapshotting.

We now describe a procedure seen in Algorithm 6 to reconstruct the tree that satisfies Properties 2 and 3. To check that T is valid, the client needs the trusted Merkle-root hash of T and the number m of chunks in this tree. Both the Merkle-root hash and the number of chunks in the tree must be included in the block headers of the blockchain to ensure that they are trusted. In a blockchain, this information is available in the headers of a block that succeeds the n-th block (e.g., in the $(n+1)$-th block).

A client peer requests a chunk by its identifier (i.e., a value in $0..(m-1)$) and receives as a response the requested chunk with the proof of inclusion of the chunk's root. The peer then rebuilds the subtree rooted at the chunk's root with all the nodes in the chunk. The procedure that checks the validity of the chunk (Property 2) proceeds in two steps. In the first step, the peer verifies the proof's validity by reconstructing the path from the chunk's root until the Merkle-root based on the hashes provided in the proof. The proof is valid if and only if the reconstructed Merkle-root matches the trusted one. In the second step, the peer recomputes the hashes of the subtree from the leaves up to the chunk root. The chunk is valid if the computed hash of the chunk root matches the hash provided, known to be valid from the first step.

To ensure that the client peer builds a proper subtree with the leaves in a chunk (Property 3), the peer first sorts all leaves by height. Recall from Section 4.3.1 that the height stored in a leaf is either 0, if the leaf's key is not an

inner node, or the inner node's height in the tree. Then, the peer builds the tree by adding one node at a time, in descending order of the node height (i.e., root first). Multiple subtrees can be built in parallel to speed up the process. When the last chunk is built, the peer links all the chunk subtrees: The peer sorts all subtrees in descending order of their root node height, at which point the tree's root lies necessarily in the first position of the array. Each subtree is then added sequentially in the tree. When the last element is added the tree is complete. The correctness of this procedure is shown below. After the tree is built, the peer recomputes all the hashes.

4.10.1 Correctness

We initially consider Property 2. The first step of the protocol checks the validity of the chunk root using the proof of integrity of a single node of the tree. This follows immediately from the properties of Merkle trees. A valid chunk root provides a trusted hash of its subtree. Then, in the second step, we compute the hashes from the leaves all the way up to the chunk root. If the trusted and the computed hash match, then the chunk is valid.

We now consider Property 3. Let h be the height of T, and $T(n)$ be a subtree of T that contains nodes from height h down to height n. The proof is by backwards induction on n.

Base step. Trivially true for $n = h$ since there is a single node with height h in subtrees $T(h)$ and $T'(h)$, the root.

Inductive step. We assume that $T(n+1)$ and $T'(n+1)$ are isomorphic and show that $T(n)$ and $T'(n)$ are isomorphic too, for $0 \leq n < h$. Let k be a node in $T(n)$ that is not in $T(n+1)$. Since T is a binary search tree, there is only one path in $T(n+1)$ that leads to k. From the inductive hypothesis, $T(n+1)$ and $T'(n+1)$ are isomorphic, thus, k will end up in the same location in $T'(n)$. Since $T(0) = T$ and $T'(0) = T'$, we conclude that T and T' are isomorphic.

Algorithm 6 Reconstruction

```
hash: trusted Merkle-root hash, stored in the block header
n_chunks: number of chunks of tree, stored in the block header
c_roots: set with all chunk roots, initially empty
t_parts: set with all tree parts, initially empty

procedure BUILD_SUBTREE(chunk)
    c_root := chunk→root                                   ▷ chunk root in chunk
    if c_root→height = 0 then                              ▷ if root has no inner node...
        return c_root, {c_root}                            ▷ no subtree to build
    c_root := NEW_INNER(c_root→key, ϵ, ϵ, root→height)     ▷ create inner node for chunk root
    parts := {c_root}                                      ▷ all tree parts, initially only chunk root
    L := chunk→leaves                                      ▷ prepare to sort leaves:
    SORT(L)                                                ▷ sort by height in descending order
    for i in 0..(|L|-1) do                                 ▷ first pass: create inner nodes
        if L[i]→height > c_root→height then                ▷ above root?
            parts := parts ∪ {NEW_INNER(L[i]→key, ϵ, ϵ, l→height)}   ▷ to be used after all chunks received
        if 0 < L[i]→height < c_root→height then            ▷ below root?
            inner_node := NEW_INNER(L[i]→key, ϵ, ϵ, L[i]→height)   ▷ create inner node
            INSERT_NODE(c_root, inner_node)                ▷ insert it in subtree
    for i in 0..(|L|-1) do                                 ▷ second pass: insert leaf nodes
        INSERT_NODE(c_root, L[i])                          ▷ insert leaf in subtree
        chunk→leaf[i].height := 0                          ▷ adjust leaf height
    return c_root, parts
procedure INSERT_NODE(x, r)                                ▷ insert node r in (sub)tree x
    if x = ϵ then
        return r                                           ▷ insert here, end recursion
    if r→key < x→key then                                  ▷ search down left
        x→left := INSERT_NODE(x→left, r)
    else                                                   ▷ search down right
        x→right := INSERT_NODE(x→right, r)
```

```
30: procedure RECEIVE(chunk, cproof)
31:     c_root, s_parts := BUILD_SUBTREE(chunk)        ▷ build subtree rooted at
    chunk root and all tree parts
32:     if VALID(hash, c_root, cproof) then                    ▷ is chunk valid?
33:         c_roots := c_roots ∪ {c_root}                  ▷ keep all chunk roots
34:         t_parts := t_parts ∪ s_parts                      ▷ and all tree parts
35:         if |c_roots| = n_chunks then                  ▷ received all chunks?
36:             SORT(t_parts)                  ▷ sort parts by height in desc order
37:             t_root := t_parts[0]                ▷ tree root is the deepest node
38:             for i in 1..|t_parts| do                          ▷ include all parts
39:                 INSERT_NODE(t_root, t_parts[i])
40:             COMPUTE_HASH(t_root)                   ▷ compute all tree hashes
41:             return t_root
42:     return ϵ
```

4.11 Micro-benchmarks evaluation

We evaluate the AVL* tree using micro-benchmarks. All tests were conducted using two computers, each one with a 32-core AMD Opteron 6366 processor running at 2.2GHz, 132Gb of memory and SATA SSD disks. The computers are connected with 1Gbps ethernet cards in a local-area network (LAN), and the round-trip time for packages between them is around 0.1ms.

Baseline comparisons. The experiments use two alternative data structures for baseline comparisons. The first is the IAVL+ tree used in Cosmos/Tendermint. The second is a Merkle-ized B^+ tree [23]. We are unaware of any blockchain-based system that uses B^+ trees. But, they are widely used in database and file systems to minimize the number of I/O operations, and therefore present a natural comparison. Similar to AVL*, a B^+ tree groups data values in cells. To Merkle-ize the tree, each cell contains the hashes of every key stored in the cell. Leaves of the Merkle-ized B^+ are stored in a *record* and linked together to facilitate tree traversals and range queries.

The integrity of a leaf can be verified by recomputing the hashes all the way to the tree root. Since the B^+ tree is not a binary tree, proofs cannot be pruned purely in a logarithmic way as the AVL*. Information about the other nodes that share the proof's path have to be included in the proof. A more elaborate design could be made for the Merkle-ized B^+ tree where each inner-node contains a binary Merkle-tree to amortize the extra proof's cost but it is beyond the scope

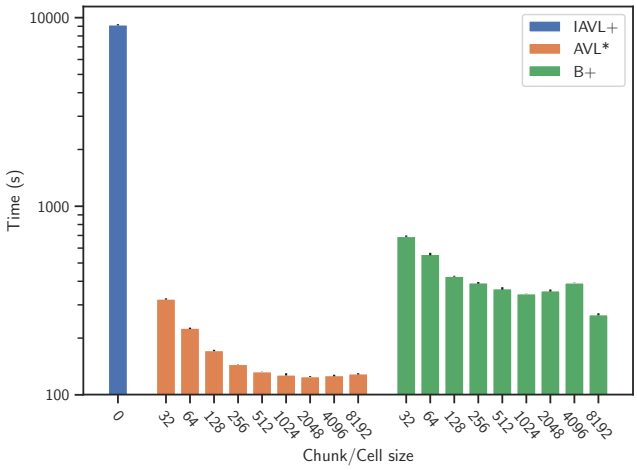
of this thesis and left as future work.

4.11.1 State synchronization

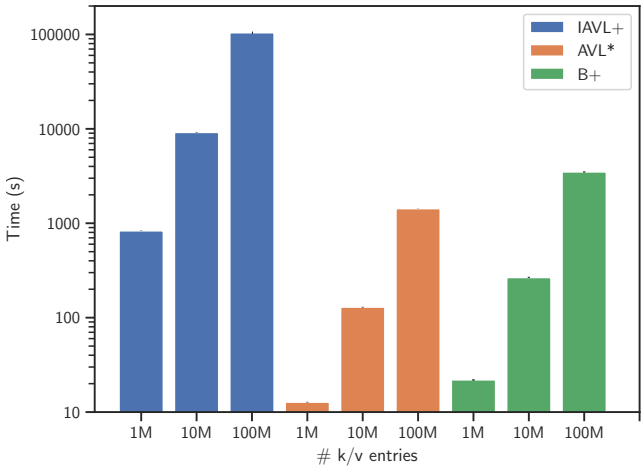
End-to-end state transfer. The first experiment compares the end-to-end state transfer time for the AVL*, IALV+, and B^+ trees. The end-to-end state transfer time includes the time to serialize the tree on the server peer, transfer across the network, and reconstruct the tree at the client peer. For the IAVL+, the tree nodes are requested until the last node is reached and the transmission ends. For the AVL*, chunks are sent together with information about each leaf height, and similarly for the B^+ tree.

Figure 4.3a shows the time on the y-axis fixing the number of entries at 10 million. For the AVL* and B^+ trees, we vary the size of the chunk or cell, and show the different sizes on the x-axis. We observe that time decreases with increased chunk sizes. The results show that state transfer time for AVL* and B^+ trees is about 2 orders-of-magnitude less than that of IAVL+ trees. AVL* trees are also faster than B^+ trees.

Next, we fixed the chunk size to 8KB and varied the number of entries in the tree. We again measured the end-to-end state transfer time. The results are shown in Figure 4.3b. As expected, we see that the transfer time increases linearly with the number of elements.



(a) Varying chunk/cell size, with 10M key/value entries

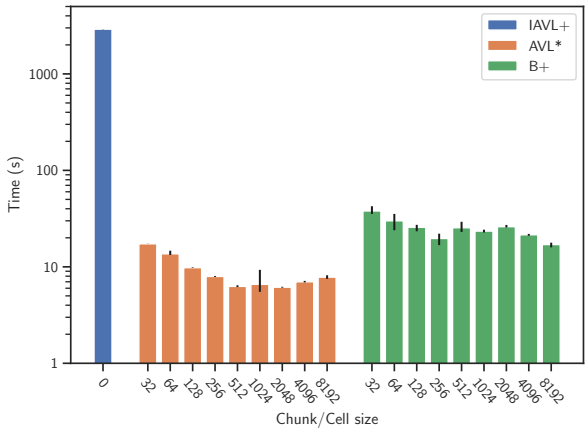


(b) Varying key/value entries, with 8KB chunks/cells

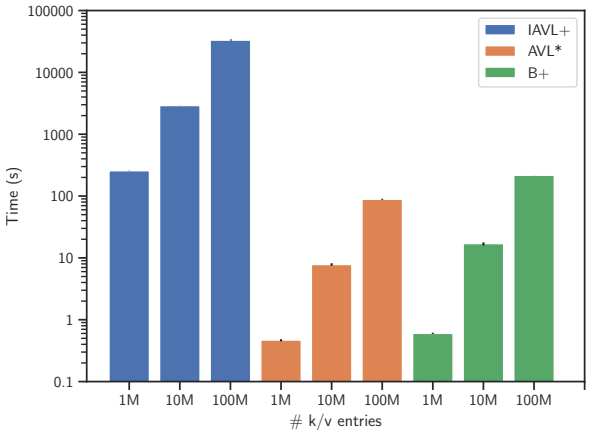
Figure 4.3. Time to serialize, transfer and recreate tree.

Tree serialization. To further understand why state synchronization is faster for AVL* trees, we performed a second experiment to measure just serialization time. We repeat the methodology as in the end-to-end experiments. The first experiment fixes the number of entries at 10 million, and varies the chunk/cell size (Figure 4.4a). The second experiment fixes the size of the chunk/cell to 8KB, and varies the number of elements (Figure 4.4b).

The results show that the serialization time for IAVL+ is three orders of magnitude slower than the other trees. The main reason is that IAVL+ nodes require more I/O than fetching data in chunks/cells as the AVL* or B^+.



(a) Varying chunk/cell size, with 10M key/value entries

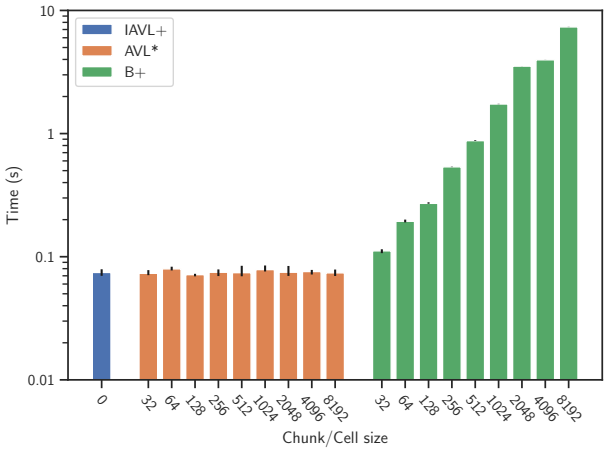


(b) Varying key/value entries, with 8KB chunks/cells

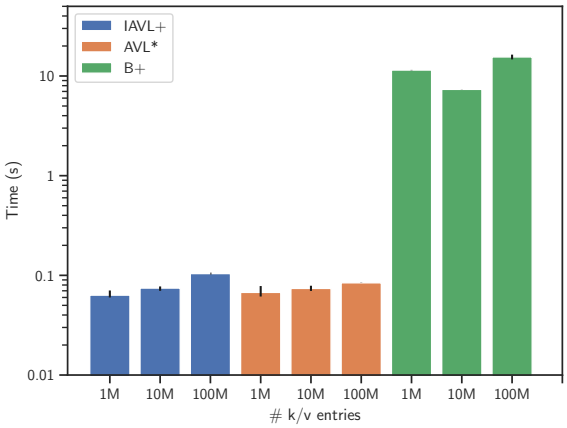
Figure 4.4. Time to serialize the whole tree on sender side.

4.11.2 Validation time

One hypothesis introduced in this chapter is that AVL* trees allow for fast tree validation. The next set of experiments evaluates this claim. We measure the time to validate 10 thousand random leaves from the tree under two varying scenarios: (i) with a fixed number 10 million entries and varying the tree's chunk/cell size (Figure 4.5a); and (ii) a fixed 8KB chunk/cell size, and varying number of elements (Figure 4.5b).



(a) Varying chunk/cell size, with 10M key/value entries

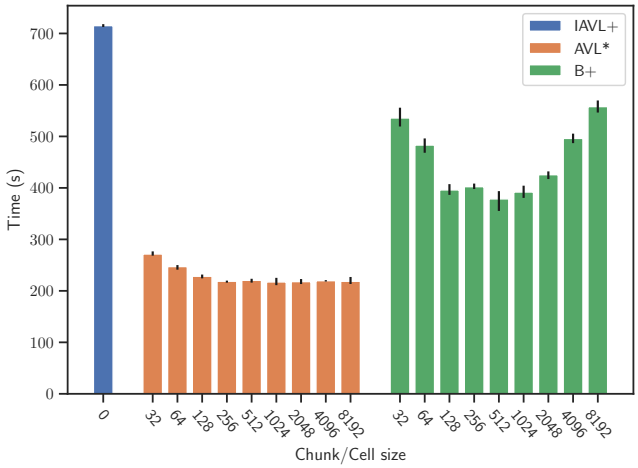


(b) Varying key/value entries, with 8KB chunks/cells

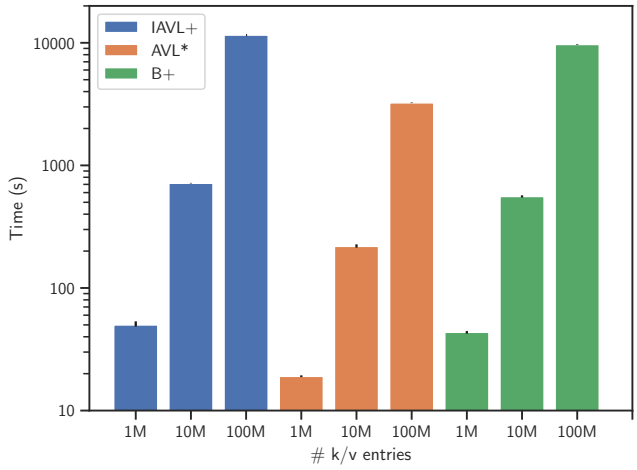
Figure 4.5. Validation time.

The results show that IAVL+ and AVL* trees take similar time for validation. However, the time taken to validate the same elements in the B^+ tree grows linearly with the block size, since B^+ proofs grow linearly with the cell size, whereas proofs from the AVL-family trees grow logarithmic with the number of elements.

4.11.3 Tree construction



(a) Varying chunk/cell size, with 10M key/value entries



(b) Varying key/value entries, with 8KB chunks/cells

Figure 4.6. Time to create a new tree, element by element.

The third set of experiments quantify the time it takes to construct a new tree from scratch, inserting one key-value pair at a time. For a workload, we used 20-byte keys and 100-byte values defined at random. We used the same methodology as above. The first experiment fixes the number of entries at 10 million, and varies the size of the chunk/cell size (Figure 4.6a). The second experiment fixes the size of the chunk/cell to 8KB, and varies the number of elements (Figure 4.6b).

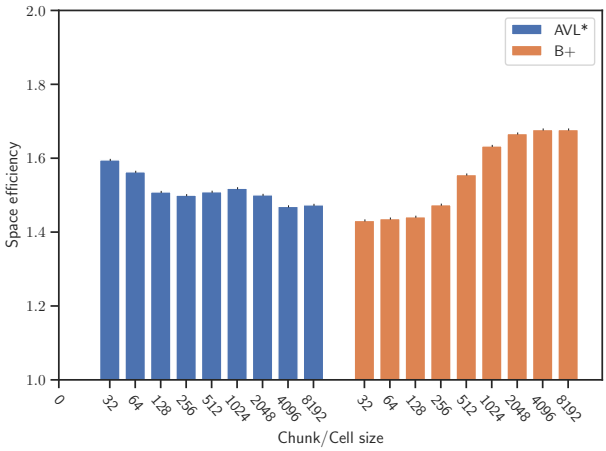
In Figure 4.6 we observe that the AVL* performs more than two times faster

than the IAVL+, the main reason being that neither the AVL* nor the B^+ tree store intermediate nodes from the tree in disk, saving twice as much data. Differences in time from the AVL* and B^+ tree are due to larger proofs in the B^+ tree experiment.

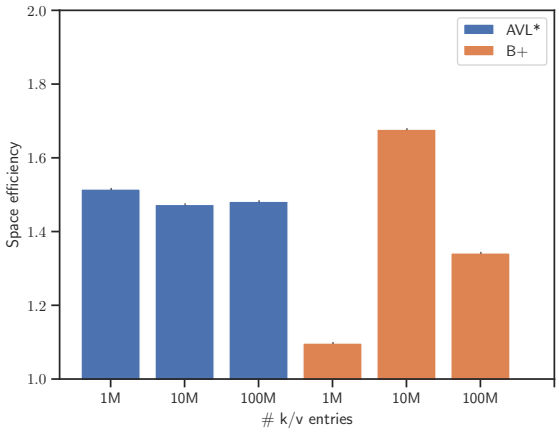
4.11.4 Space efficiency

The IAVL+ tree reserves space proportional to the number of nodes. That is, it only requests memory for a node if it will be in the tree. In contrast, AVL* and B^+ trees request memory in chunks/cells that may not be filled. We define *space efficiency* as the ratio between the number of chunks for the AVL* tree or cells for the B^+ tree divided by the ideal case where a minimal number of chunks or cells are needed. For instance, a space efficiency of 2 means that there are twice as much chunks as needed in the ideal scenario. The last set of experiments measure the space-efficiency of the AVL* and B^+ trees. The worst-case scenario for the B^+ tree is when all cells are half full [23] (i.e., space-efficiency of 2).

In Figure 4.7a we measure the space efficiency of the AVL* and B^+ trees when varying the chunk/cell size of the tree. Both the AVL* and B^+ trees maintain on average 1.5 space efficiency. Observe that the space efficiency of the B^+ tree increases with the size of cells. In Figure 4.7b we vary the number of elements in the tree. We observe great differences in space efficiency for different elements of the B^+ tree while the AVL* remains almost constant.



(a) Varying chunk/cell size, with 10M key/value entries



(b) Varying key/value entries, with 8KB chunks/cells

Figure 4.7. Space efficiency (1 is optimum).

To understand why the space efficiency of the B^+ tree varies in Figure 4.7b, we measured the space efficiency after each element is inserted and compared both the AVL* and B^+ trees. In Figure 4.8 the x-axis is the number of elements in the tree and in the y-axis the space efficiency. While the space efficiency average is the same for each tree (horizontal dashed line), the space efficiency of the AVL* remains almost constant while in the B^+ it swings as elements are inserted. The reason for this behavior is that the B^+ tree tends to fill all cells before it changes its height since elements are chosen for insertion from a uniform distribution. Once the cells are nearly full (i.e., space efficiency closer to one), further inserts will lead to splits, which worsens the space efficiency. After most cells have been

split (i.e., space efficiency tending to two), they are at half capacity and start tp fill again. The issue does not happen with the AVL*, since the height of the tree is not tightly coupled with the number of chunks. Chunk splits only increase the number of chunks by one, and splits by rotations account for only 4% of the total number of splits.

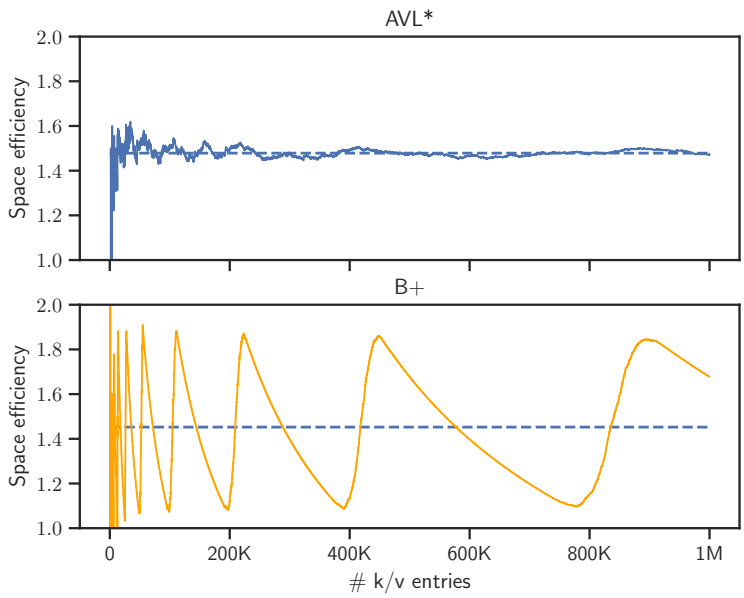


Figure 4.8. Space efficiency with varying number of entries inserted in random order, with 8KB chunks/cells.

We test inserting elements in order, the worst-case scenario for space efficiency for both the AVL* and B^+. In Figure 4.9 we can see experimentally how the space efficiency approaches 2 for both experiments. Given a tree with k number of chunks/cells. The space efficiency oscillates because the number of chunks or cells is incremented once every $k/2$ elements are added, while the ideal number is incremented once every k elements are added, hence the space efficiency metric bounces in the interleaving of the two. The space efficiency of both trees asymptotically approaches two. For n elements, the number of chunk or cells for the first $k/2+1$ elements is one, for larger values is $2 \times n/k - 1$, since all chunks or cells are half occupied except for the last. The number of ideal chunks is always n/k, and $lim_{n \to \infty}(2 \times n/k - 1)/(n/k) = 2$.

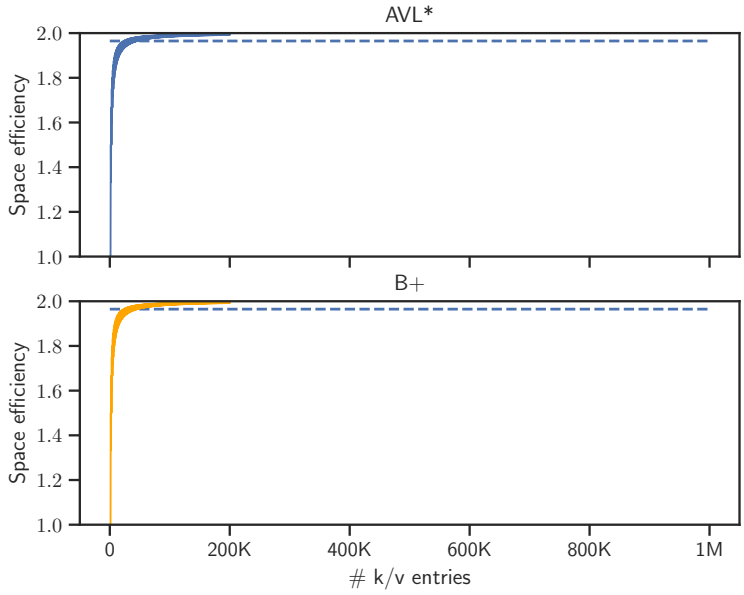


Figure 4.9. Space efficiency with varying number of entries inserted in order, with 8KB chunks/cells.

4.12 Evaluation in a real scenario

To evaluate the behavior of our data structure and algorithms in a real scenario, we integrated them into Tendermint [13] and conducted experiments under different conditions (Section 4.12.1). There are three sets of experiments. The first experiments measure the performance of an execution under high load (Section 4.12.2). In the second set of experiments, we consider the performance of state synchronization (Section 4.12.3). In the third set of experiments, we gradually introduce Byzantine peers and show how malicious behavior impacts the performance of state synchronization (Section 4.12.5).

4.12.1 Environment and application

All tests were conducted in a wide-area network (WAN) using Amazon's Elastic Computing (EC2) platform. We evaluated the system with 10 peers (small setup) and 80 peers (large setup). Our large setup is a fair approximation of Cosmos/Tendermint's current production system, with 125 peers. Peers were deployed in datacenters in seven Amazon regions: three datacenters in North

America (Oregon, Ohio, Canada Central), two in Asia (Tokyo, Hong Kong), three in Europe (Paris, Frankfurt, and London), one in South America (Sao Paulo), one in the Middle East (Bahrain), and one in Africa (South Africa). We used t3.xlarge and r5.xlarge instances.

We used most of Tendermint's default parameters, and set the mempool cache with 50k transactions, and block interval of 1 second for executions with 10 peers and 5 seconds for executions with 80 peers. These parameters led to the best results for throughput and latency for both the IAVL+ and the AVL* experiments. In the setup with 10 peers, each peer is connected to every other peer; in the setup with 80 peers, each peer is connected to 25 random peers.

We developed a key-value store application using Tendermint's ABCI, and benchmarked the application using the IAVL+ tree and the AVL* tree. Clients submit transactions that add new key-value pairs to the store. A key contains 20 bytes and a value contains 100 bytes, both generated randomly by clients. We evaluated the IAVL+ and the AVL* trees with chunks that can contain up to 10k and 100k key-value pairs, amounting to roughly 1MB and 10MB chunks, respectively. We considered executions in which clients include 10k, 100k, 1M and 10M key-value entries to the store.

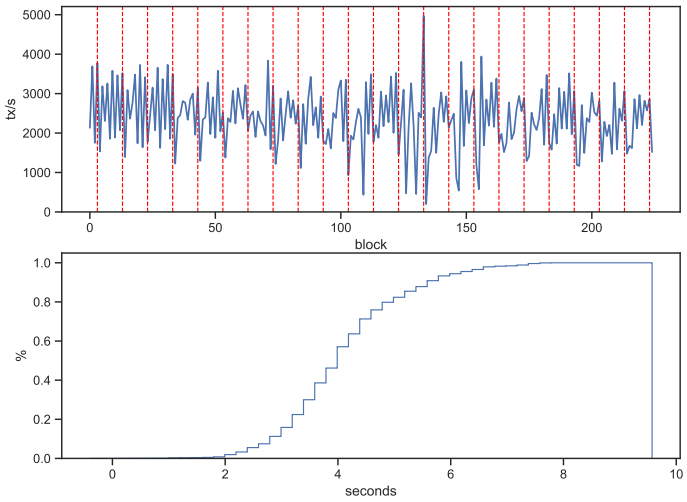
4.12.2 Steady-state operation

In the first set of experiments, we assume that a client has already synchronized state, and we ask the question: *how does using an AVL* tree, rather than an IAVL+ tree, impact Tendermint during steady-state operation?* We evaluate this question according to three metrics: (i) throughput and latency for transaction processing, (ii) the time it takes to compute a snapshot, and (iii) the space utilization.

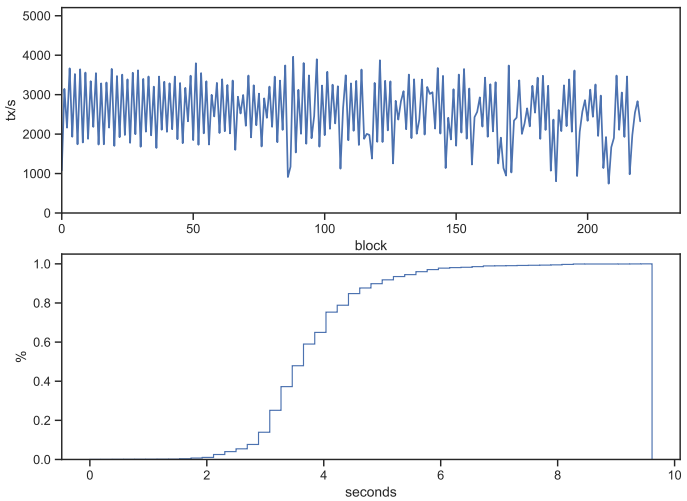
In all of these experiments, clients submit 120-byte transactions that write a key/value pair in the store. Each client operates in a closed-loop, meaning a client only submits a new transaction after it receives the response for the previously submitted transaction. The client subscribes to the blockchain and delivers the blockchain blocks. Latency is computed as the time it takes for a transaction to be included in a block. Throughput is the number of transactions in a block divided by the time it took for the block to be ordered (i.e., interval between the current block and the previous one). In the experiments with the IAVL+ tree, snapshots are built every ten blocks.

Steady-state performance. Figure 4.10 show the throughput as a time series (top) and the latency CDF (bottom). In the figure, the blockchain is composed of 10 peers and chunks of size 10k. Figure 4.11 report results for the same

experiment but with chunks of size 100k. Figures 4.12 and 4.13 show the results for the same set of experiments when the number of peers is increased to 80.

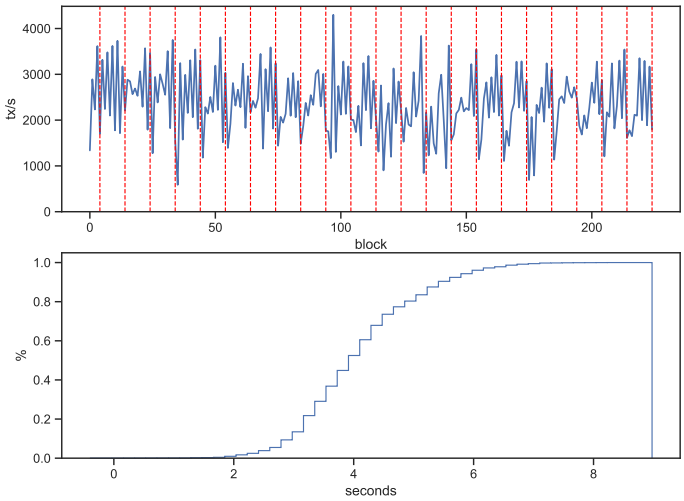


(a) IAVL+

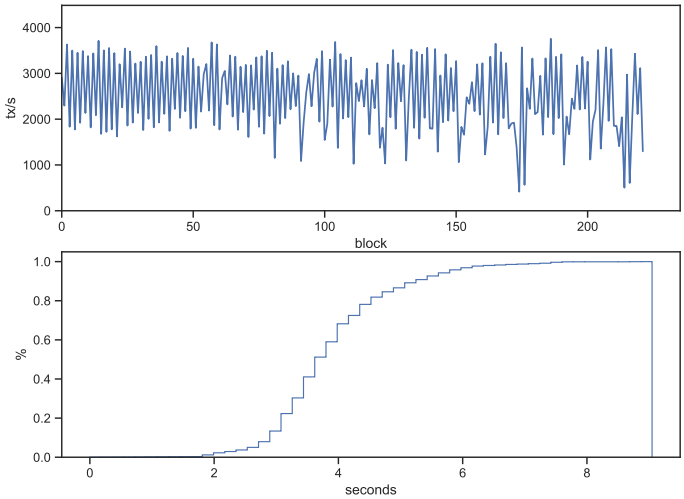


(b) AVL*

Figure 4.10. Throughput and latency of transaction execution, 10 peers, 1M key/value pairs, and 10k chunk size.

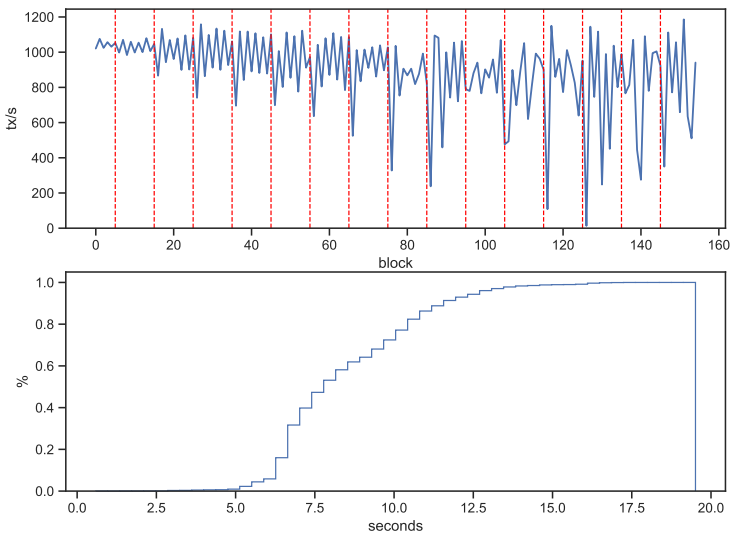


(a) IAVL+

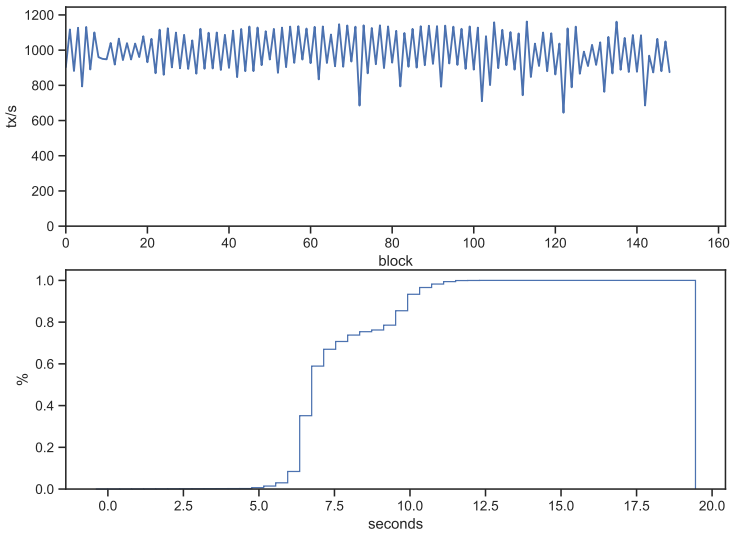


(b) AVL*

Figure 4.11. Throughput and latency of transaction execution, 10 peers, 1M key/value pairs, and 100k chunk size.

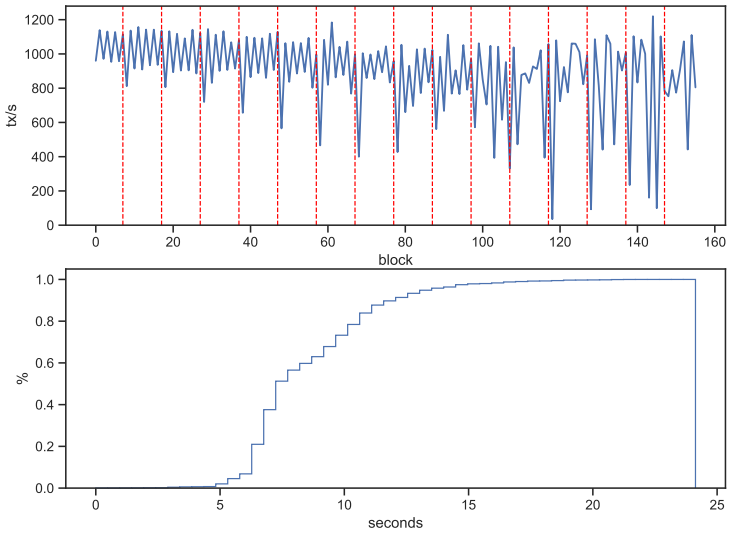


(a) IAVL+

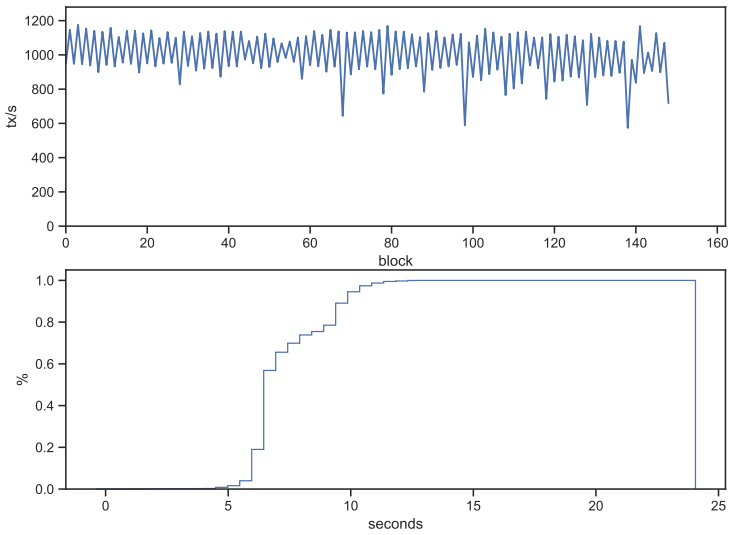


(b) AVL*

Figure 4.12. Throughput and latency of transaction execution, 80 peers, 1M key/value pairs, and 10k chunk size.



(a) IAVL+



(b) AVL*

Figure 4.13. Throughput and latency of transaction execution, 80 peers, 1M key/value pairs, and 100k chunk size.

The graphs show that using the AVL* tree results in more predictable perfor-

mance. One of the reasons for the volatility of the IAVL+ tree is that there are periodic drops in performance that occur during a snapshot operation. In the graph, the dashed-red lines indicate the time that a snapshot occurs.

Although it is somewhat difficult to tell from the graphs, the performance of AVL* is not only more predictable, but it is also better, on average, than the IAVL+. Table 4.2 shows the mean values and the relative improvement of AVL* over IAVL+ for all these experiments.

	10 peers							
	10k chunks				100k chunks			
	Throughput (tx/s)		Latency (s)		Throughput (tx/s)		Latency (s)	
	average	std.	average	std.	average	std.	average	std.
IAVL+	2373.86	764.03	4.18	1.1	2362.76	713.46	4.16	1.02
AVL*	2538.63	798.46	3.85	0.94	2491.69	802.94	3.96	0.99
var.	+7%		-8%		+5%		-5%	
	80 peers							
	10k chunks				100k chunks			
	Throughput (tx/s)		Latency (s)		Throughput (tx/s)		Latency (s)	
	average	std.	average	std.	average	std.	average	std.
IAVL+	893.28	218.33	8.63	2.27	892.82	235.59	8.71	2.6
AVL*	988.42	124.39	7.64	1.53	1001.25	131.8	7.54	1.55
var.	+11%		-11%		+12%		-13%	

Table 4.2. Throughput and latency (average and standard deviation) for IAVL+ and AVL* in different configurations.

Time for a snapshot. One of the major differences between IAVL+ and AVL* is that AVL* trees do not need to pause execution to compute a snapshot—the snapshot is taken "spontaneously" during normal execution. Because snapshot computation has a significant impact on the IAVL+ performance, we wanted to quantify the overhead.

Figure 4.14 shows the time it takes to compute a snapshot with IAVL+ trees as the number of tree leaves (i.e., key-value pairs) is increased, varying from 0 to one million leaves. As the tree gets bigger, snapshots take more time to complete. That happens because each snapshot has to serialize the whole tree. Changing the chunk sizes does not have a significant impact on the experiment's performance.

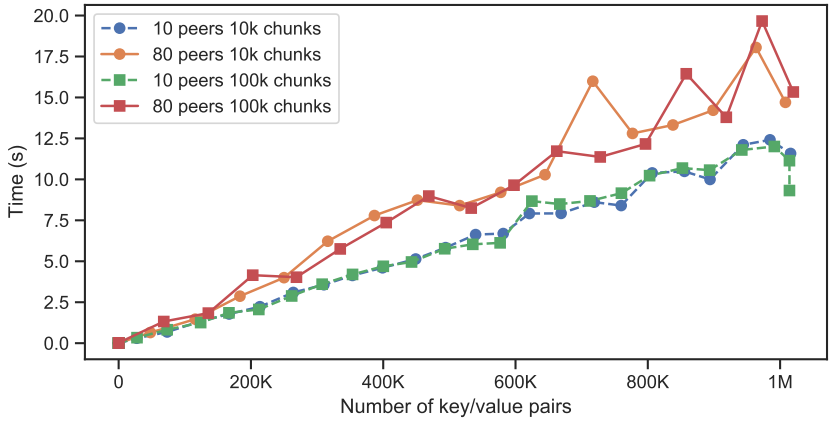


Figure 4.14. Time for an IAVL+ snapshot.

Space efficiency. The trade-off for using an AVL* is space efficiency, as the AVL* tree may use more chunks than optimal as seen in Section 4.11. Note that when creating a snapshot for the IAVL+ tree, the space efficiency is always 1 because the serialization process always serializes the tree from scratch.

Figure 4.15 shows a magnified version of Figure 4.8, where the space efficiency of the AVL* is measured as we insert one million random keys into an empty tree. The space efficiency stabilizes at around 1.45 for random data, which we believe is an acceptable overhead.

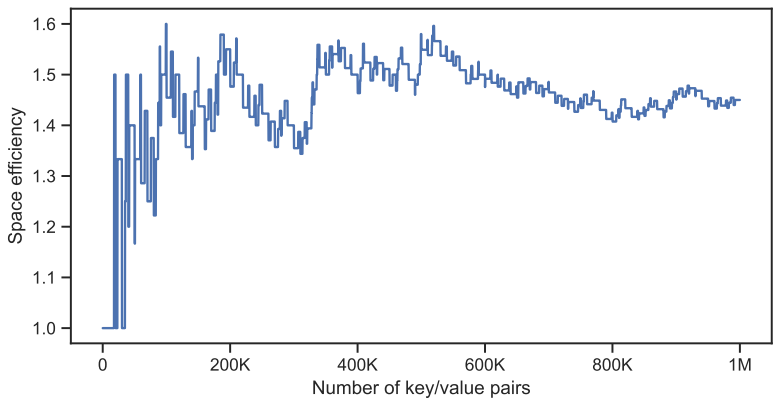


Figure 4.15. Space efficiency for the AVL*.

4.12.3 State synchronization

The second set of experiments evaluates the performance of the state synchronization operation. The main metric of concern is the time it takes to perform synchronization for a new peer joining the blockchain.

In these experiments, all peers but one are pre-initialized with a full tree. When the experiment begins, the new peer recovers the state by downloading chunks in parallel from the operational peers. We vary three parameters: (i) the size of the tree, (ii) the number of validators (10 and 80), and (iii) the size of the chunks (10k and 100k).

Figures 4.16 and 4.17 show the state synchronization times for IAVL+ and AVL* for experiments with 10 and 80 peers respectively. The results for 10k chunks and 100k chunks are similar; thus, we show results for 100k chunks only. We report the results as a ratio, with the absolute time printed at the top of each column. There are two important features to note. First, for both IAVL+ and AVL*, the synchronization time increases linearly with the size of the tree, and it does not depend on the size of the system. Second, the synchronization time for AVL* is roughly half the time required by IAVL+.

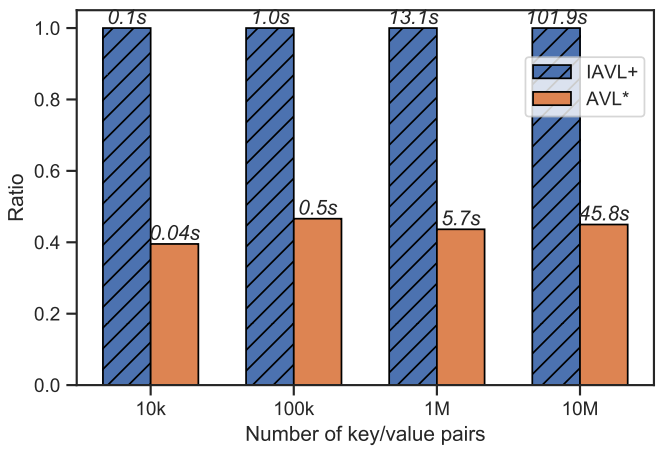


Figure 4.16. Time for a peer to recover from scratch, 100k chunks, 10 peers, varying number of key/value pairs.

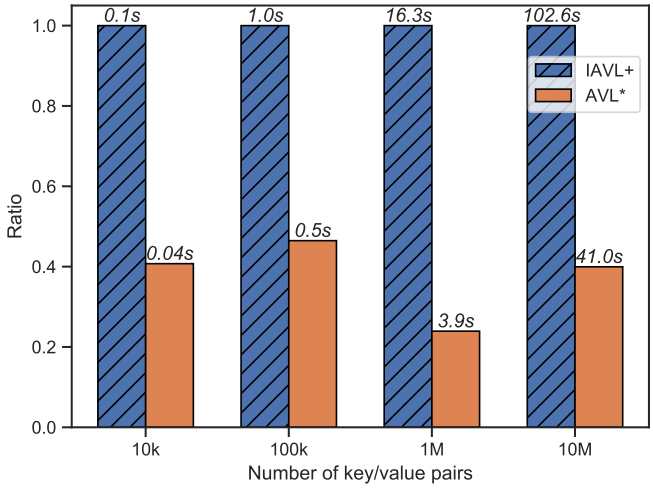


Figure 4.17. Time for a peer to recover from scratch, 100k chunks, 80 peers, varying number of key/value pairs.

We do note one detail. Recall that the AVL* does not guarantee fixed-sized chunks. So, in these experiments, the AVL* chunks tend to be smaller than those made by snapshots in the IAVL+. Snapshots from the AVL* have around 16% more chunks than the IAVL+ tree. However, although there are more chunks in the application using the AVL* tree, the data stored in each chunk is increased, since it includes the proofs of inclusion for each chunk.

Overall, the state synchronization time when using the AVL* is on average 58% faster compared to the IAVL+ tree, despite having to download more chunks.

4.12.4 Comparison with micro-benchmarks

When comparing the micro-benchmarks done in Section 4.11 with results of the experiments in this section we do not see the same gains in performance as in the previous section. The main reason for this behaviour is that the performance gains are shadowed by other Tendermint's bottlenecks in a real setup. Faster times to save the tree in disk as seen in micro-benchmarks are still observed during the experiments in this section, as using the IAVL+ produces hiccups in performance when the slower operations to save data to disk are done.

State synchronization experiments for micro-benchmarks are done in a LAN setting and use direct methods to rebuild the state, as for the same experiments in the real use-case use Tendermint's APIs which introduce an extra layer of ab-

straction and are done in a WAN which further shadows benefits observed while doing the micro-benchmarks experiments.

4.12.5 State synchronization with malicious peers

A key benefit of an AVL* tree over the IAVL+ tree is that it can gracefully recover from Byzantine or malicious behavior from peers. An IAVL+ cannot be checked for validity until the entire tree has been downloaded and reconstructed. If, after reconstructing the tree, the tree's root hash is different from the one in the trusted block header, then the client peer must refetch all chunks again. In contrast, the AVL* tree allows the client peer to detect invalid chunks easily, and remove misbehaving peers from its list of trusted peers.

To quantify the performance of AVL* in the presence of malicious peers, we again performed the state synchronization experiment, but introduced malicious peers that respond with invalid chunks. In these experiments, we used a fixed number of 80 peers and a tree with one million entries. We varied the number of malicious peers from 1 to 26. We performed the experiment five times, and report mean synchronization time, as well as the maximums and minimums in bars and whiskers. Figures 4.18 and 4.19 show the results.

In Figure 4.18, we see that, as expected, the presence of malicious peers degrades the state synchronization time. The state synchronization time grows linearly with the number of malicious peers. Because a peer can respond to multiple requests for chunks in parallel before they are verified, the number of invalid chunks sent by peers can vary. In Figure 4.19 we see that the number of chunks that need to be re-fetched is proportional to the state synchronization time.

From Section 4.12.3, we know that without any Byzantine peers, it takes 16.3 seconds for a client peer using IAVL+ to complete state synchronization (with 80 peers, 100k chunks and 1M key-value pairs). With one Byzantine peer, this time would double since the client peer would have to start from scratch. And even after retrying, there is no guarantee that the second attempt would succeed. Thus, a coordinated attack could substantially increase the state sync time of IAVL+. AVL* outperforms IAVL+ even under attack by 20 Byzantine peers.

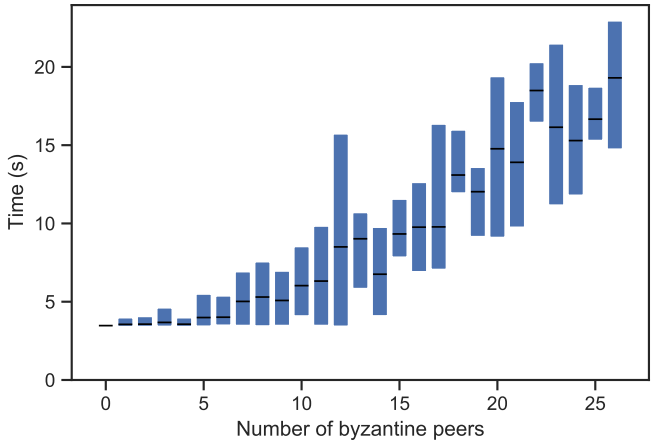


Figure 4.18. Time for state synchronization.

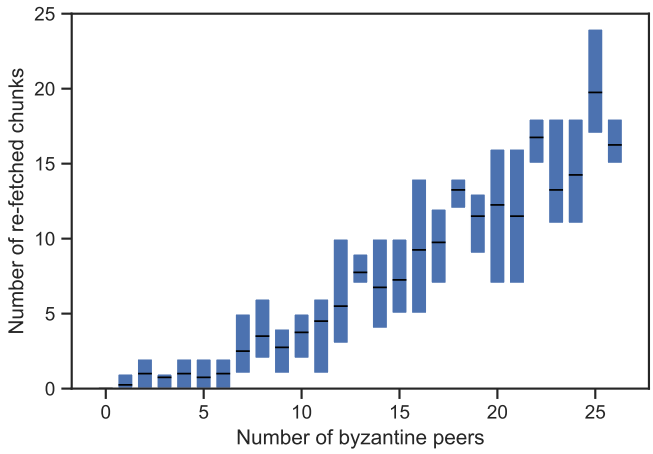


Figure 4.19. Number of re-fetched chunks.

4.13 Related Work

Many systems for state machine replication with Byzantine actors have addressed the problem of fast state synchronization without executing the entire transaction log. This is typically done by taking snapshots of the state, often called checkpoints. Such checkpoints can then be downloaded by new or recovering peers. While PBFT [18] proposed the use of a Merkle tree for its checkpoints, the Upright [22] and BFT-SMaRt [8] systems consider Merkle trees and copy-on-write

semantics to be too invasive in general to the application developer. Upright outlines three simple approaches to state transfer [22], and BFT-SMaRt [8] describes a detailed Collaborative State Transfer protocol, where the full state is downloaded from a single peer and verified against hashes from other peers. In PBFT, a binary Merkle-tree is built at each checkpoint by partitioning the application state in 4KB pages. The pages are stored as leaves of the Merkle-tree using copy-on-write to only persist pages that have been modified since the previous checkpoint. This approach is not efficient to encode a key-value storage, since operations on the key space (e.g., searches) do not have logarithmic complexity.

Unlike SMR systems, which either consider Merkle-trees too expensive [22, 8], or construct them only for state synchronization at periodic checkpoints [18], many blockchain systems already use Merkle-ized data structures to store the state. The primary use case for such Merkle-trees is to facilitate light clients, who can efficiently query for particular leaves of the tree and verify their integrity, without ever downloading the entire state or transaction history. The use of Merkle-trees for state synchronization has received much less attention, but as the state of blockchain systems grow, synchronizing it becomes more expensive.

In Geth [36], state synchronization is performed by requesting individual nodes of the tree. Peers do not have a way of knowing how long the state synchronization will last, because they do not know the total number of nodes [37]. Since the Merkle-tree is part of the consensus rules (i.e., Merkle-roots are stored in block headers), peers can verify that a received node from the tree is correct. However, given the small size of nodes (less than one KB), their randomized distribution in the underlying database, and the large size of the state (tens of GBs), requesting nodes individually leads to performance degradation for peers requesting and providing nodes. Batching nodes is a promising solution, however, it is challenging to batch nodes in a manner that can be securely verified by peers, and limits attacks on honest peers. For instance, in OpenEthereum [57], snapshots are taken periodically by serializing the entire state, and dividing it in large chunks. The hashes of each chunk are published in a manifest file. Since the manifest is not part of the consensus process, there is no way to verify that a chunk is correct before downloading all of them. Successfully completing the state synchronization in such a system thus depends on retrieving a correct manifest, which requires strong assumptions, for instance, that a particular peer can be trusted or that a majority of connected peers are correct. This is stronger than the usual assumption of a single (though unspecified) correct peer commonly used in blockchain systems.

Other blockchain systems have proposed to take snapshots of the Merkle-tree by periodically dividing it in chunks. In Codechain snapshots [63], chunks are

built from nodes within a certain depth from a common root, and the hash of the snapshot is included in the block header so chunks can be verified incrementally by peers. Chunks may contain entire sub-trees, or may be limitted to the upper nodes in a sub-tree. In Tendermint IAVL+ snapshots, tree nodes are serialized in order and assembled into chunks of a given size [3]. However, since Tendermint's block header does not currently support snapshot hashes, chunks cannot be incrementally verified by peers. Hence the need for a tree that incorporates chunking directly into its structure.

Motivated by their use in the blockchain context, numerous different Merkle-tree designs have been proposed lately. TurboGeth [68] separates the key-value storage from the Merkle tree structure, and batches Merkle tree nodes in chunks to reduce the number of lookups during Merkle operations. This has the effect of greatly improving performance without changing the structure of the hash tree itself.

Sparse Merkle-trees [28] have been adopted by other blockchain projects [69]. Recent advances in cryptography have even offered a glimpse into generalizations of Merkle trees called accumulators, which enable O(1) proofs of set-membership and state-less blockchain clients [12].

While there has been a wide diversity of proposed and implemented tree designs, the AVL* is the only known tree to target both the light client and state synchronization use cases found in blockchain systems.

4.14 Conclusion

State synchronization is a significant bottleneck for blockchain-based systems. In this chapter, we presented a novel extension to a Merkle-ized AVL+ tree that incorporates chunks. This extension allows peers to download portions of the state in parallel, and validate each chunk independently. We have also described an algorithm for deterministic tree reconstruction, ensuring that all peers have the same state. We have extensively tested our algorithms using micro-benchmarks and in a geo-distributed environment that shows the benefits when reconstructing the state from snapshots and gracefully coping with Byzantine failures.

By using the AVL* tree we can achieve a fast and robust state synchronization. Making peer synchronization more achievable and thus increasing blockchain scalability.

Chapter 5

Conclusion

Blockchains continue to increase in popularity, revealing a multitude of different designs motivated by multiple viewpoints. Since each different blockchain introduces new concepts (e.g., anonymity in ZCash [41]), the emergence of a protocol to integrate heterogeneous blockchains is more likely to have impact than a single blockchain incorporating all different features. Furthermore, blockchain original goals of being decentralized is pushed to centralization in different ways. For example, large mining pools end up centralizing the system and higher hardware requirements discourage new peers from joining the network.

In this thesis, we explored how to scale blockchains in terms of performance and state synchronization to answer the questions raised in Chapter 1:

- Can blockchain systems scale performance through sharding?
- How can blockchains scale with IBC?
- How to design scalable applications that can live in many blockchains?
- How to improve scalability by faster blockchain synchronization.

For the first question, we evaluated how a popular blockchain behaves when its state is sharded with several different methods using centralized and decentralized algorithms. We investigated how to shard the state in an efficient way that minimizes communication across shards while keeping the system's load balanced. We tried to maximize the system's potential performance by minimizing synchronization between shards and at the same time, keeping the shards balanced. We analyzed two years of historical data to substantiate our findings.

For the second question, we designed a framework and protocol exposing primitives to developers regarding other blockchains. We modified two real

blockchain systems that implement the *move protocol* for transferring state within blockchains. To answer the third question, we fit two popular applications in Ethereum to our framework and showed that the *move protocol* provides flexibility for smart contracts to be written in a scalable way. We extensively experimented with our framework and protocol with traces based on real-world applications.

Finally, for the last question, we proposed AVL*, a data structure in which blockchain systems can reduce the catch-up time for new incoming peers while keeping the blockchain robust (i.e., resistant to attacks from the blockchain's original assumptions). We modified Tendermint's tree data structure and tested an application in a geo-distributed scenario over AWS with more than one hundred different instances scattered in all Amazon's regions across the globe.

5.1 Future work

Designing IBC protocols for integrating blockchains faces some issues, among of which, how to deal with forks in each blockchain. In Chapter 3, we have seen that most IBC designs define a parameter p, in which a transaction is only considered for IBC when p blocks have been appended after the transaction's block. In this case, there is an assumption that a blockchain involved with IBC will not fork its state after p blocks. How to deal with forks in blockchains while developing IBC protocols remains an open research question.

Making blockchains inter-operable typically requires changes in the protocol (as proposed in Chapter 3) for each of the involved blockchains. Some blockchains do not wish to incorporate changes that require trust in their protocols (e.g., Pegged Sidechains [7] in Bitcoin). A trustless solution to integrate blockchains remains an open research question.

In Chapter 3, the design of the *move protocol* allows only for programmable blockchains using the EVM to communicate. The protocol could be abstracted to allow for more blockchain designs that do not depend on the EVM to communicate while maintaining the same abstractions for developers.

In Chapter 4 we briefly introduced a Merkle-ized B^+ tree. Despite disadvantages when comparing it to the AVL*, the Merkle-ized B^+ could be modified to incorporate shorter proofs.

Bibliography

[1] Abebe, E., Behl, D., Govindarajan, C., Hu, Y., Karunamoorthy, D., Novotny, P., Pandit, V., Ramakrishna, V. and Vecchiola, C. [2019]. Enabling enterprise blockchain interoperability with trusted data transfer (industry track), *arXiv preprint arXiv:1911.01064* .

[2] Adel'son-Vel'skii, G. M. and Landis, E. M. [1962]. An algorithm for organization of information, *Doklady Akademii Nauk*, Vol. 146, Russian Academy of Sciences, pp. 263–266.

[3] *ADR 053: State Sync Prototype* [2020]. `https://github.com/tendermint/tendermint/blob/master/docs/architecture/adr-053-state-sync-prototype.md`.

[4] Ahmed-Nacer, T., Sutra, P. and Conan, D. [2016]. The convoy effect in atomic multicast, *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, IEEE, pp. 67–72.

[5] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D. and Danezis, G. [2018]. Chainspace: A Sharded Smart Contracts Platform, *The Network and Distributed System Security Symposium (NDSS)*.

[6] *Avalanche* [2020]. *`https://www.avalabs.org/`* .

[7] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J. and Wuille, P. [2014]. Enabling blockchain innovations with pegged sidechains, *`https://www.blockstream.com/sidechains.pdf`* .

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

[9] Bessani, A., Sousa, J. and Alchieri, E. [2014]. State machine replication for the masses with bft-smart, pp. 355–362.

[10] *Bitcoin energy consumption* [2019]. https://digiconomist.net/bitcoin-energy-consumption.

[11] *Bloat attack* [2020]. `https://ethereum.stackexchange.com/questions/11312/how-was-the-state-bloat-attack-that-led-to-the-eip-150-hardfo`

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

[13] Buchman, E. [2016]. *Tendermint: Byzantine fault tolerance in the age of blockchains*, Master's thesis.

[14] Buterin, V. [2014]. Weak subjectivity, https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/.

[15] Buterin, V. [2018]. Cross-shard contract yanking, `https://ethresear.ch/t/cross-shard-contract-yanking/1450`.

[16] Buterin, V. et al. [2014]. A next-generation smart contract and decentralized application platform, `https://github.com/ethereum/wiki/wiki/White-Paper`.

[17] Cachin, C. and Vukolić, M. [2017]. Blockchains consensus protocols in the wild, *arXiv preprint arXiv:1707.01873* .

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

[19] Castro, M., Liskov, B. et al. [1999]. Practical byzantine fault tolerance, *OSDI*, Vol. 99, pp. 173–186.

[20] Chepurnoy, A., Kharin, V. and Meshkov, D. [2018]. A systematic approach to cryptocurrency fees, Cryptology ePrint Archive, Report 2018/078. `https://eprint.iacr.org/2018/078`.

[21] Church, A. et al. [1936]. A note on the entscheidungsproblem, *J. Symb. Log.* **1**(1): 40–41.

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

[23] Comer, D. [1979]. Ubiquitous b-tree, *ACM Computing Surveys (CSUR)* **11**(2): 121–137.

[24] *consul* [2020]. `https://www.consul.io`.

[25] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P. et al. [2013]. Spanner: Google's globally distributed database, *ACM Transactions on Computer Systems (TOCS)* **31**(3): 8.

[26] *Cosmos SDK* [2020]. `https://github.com/cosmos/cosmos-sdk`.

[27] Crain, T., Natoli, C. and Gramoli, V. [2018]. Evaluating the red belly blockchain, *arXiv preprint arXiv:1812.11747* .

[28] Dahlberg, R., Pulls, T. and Peeters, R. [2016]. Efficient sparse merkle trees, *Nordic Conference on Secure IT Systems*, Springer, pp. 199–215.

[29] De Filippi, P. and Loveluck, B. [2016]. The invisible politics of bitcoin: governance crisis of a decentralized infrastructure, *Internet Policy Review* **5**(4).

[30] Decker, C. and Wattenhofer, R. [2013]. Information propagation in the bitcoin network, *IEEE P2P 2013 Proceedings*, IEEE, pp. 1–10.

[31] *etcd* [2020]. `https://etcd.io`.

[32] Eyal, I. and Sirer, E. G. [2014]. Majority is not enough: Bitcoin mining is vulnerable, *International conference on financial cryptography and data security*, Springer, pp. 436–454.

[33] Fabian, V. and Vitalik, B. [2015]. Erc20 token standard.
URL: *https://eips.ethereum.org/EIPS/eip-20*

[34] Garey, M. R., Johnson, D. S. and Stockmeyer, L. [1974]. Some simplified np-complete problems, *STOC*.

[35] Gazi, P., Kiayias, A. and Zindros, D. [2018]. Proof-of-stake sidechains., *IACR Cryptology ePrint Archive* **2018**: 1239.

[36] *go-ethereum* [2020]. `https://github.com/ethereum/go-ethereum`.

[37] *Go Ethereum FAQ* [2020]. `https://geth.ethereum.org/docs/faq`.

[38] Gray, J. and Lamport, L. [2006]. Consensus on transaction commit, *ACM Trans. Database Syst.* **31**(1): 133–160.

[39] Gray, J. N. [1978]. Notes on data base operating systems, *Operating Systems*, Springer, pp. 393–481.

[40] Herlihy, M. [2018]. Atomic cross-chain swaps, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, ACM, pp. 245–254.

[41] Hopwood, D., Bowe, S., Hornby, T. and Wilcox, N. [2016]. Zcash protocol specification, *GitHub: San Francisco, CA, USA* .

[42] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. [2010]. Zookeeper: Wait-free coordination for internet-scale systems, *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, USENIX Association, USA, p. 11.

[43] *Hyperledger Burrow client* [2020]. `https://github.com/hyperledger/burrow`.

[44] *IAVL+ implementation* [2020]. `https://github.com/tendermint/iavl`.

[45] *IBC Protocol* [2020]. `https://github.com/cosmos/ics/blob/master/spec.pdf`.

[46] Kahng, A. B., Lienig, J., Markov, I. L. and Hu, J. [2011]. *VLSI physical design: from graph partitioning to timing closure*, Springer Science & Business Media.

[47] Karypis, G. and Kumar, V. [1998]. A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM Journal on scientific Computing* **20**(1): 359–392.

[48] Kernighan, B. W. and Lin, S. [1970]. An efficient heuristic procedure for partitioning graphs, *Bell system technical journal* **49**(2): 291–307.

[49] Kiayias, A., Russell, A., David, B. and Oliynykov, R. [2017]. Ouroboros: A provably secure proof-of-stake blockchain protocol, *Annual International Cryptology Conference*, Springer, pp. 357–388.

[50] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N. and Ford, B. [2017]. Omniledger: A secure, scale-out, decentralized ledger., *IACR Cryptology ePrint Archive* **2017**: 406.

[51] Kwon, J. and Buchman, E. [2019]. Cosmos whitepaper, `https://cosmos.network/resources/whitepaper`.

[52] Lamport, L. [1978]. Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**(7): 558–565.

[53] Liu, Z., Xiang, Y., Shi, J., Gao, P., Wang, H., Xiao, X., Wen, B. and Hu, Y.-C. [2019]. Hyperservice: Interoperability and programmability across heterogeneous blockchains, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 549–566.

[54] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S. and Saxena, P. [2016]. A secure sharding protocol for open blockchains, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 17–30.

[55] Nakamoto, S. [2008]. Bitcoin: A peer-to-peer electronic cash system.

[56] Nogueira, A., Casimiro, A. and Bessani, A. [2017]. Elastic state machine replication, *IEEE Transactions on Parallel and Distributed Systems* **28**(9).

[57] *OpenEthereum WarpSync* [2020]. `https://openethereum.github.io/wiki/Warp-Sync`.

[58] Poon, J. and Dryja, T. [2016]. The bitcoin lightning network: Scalable off-chain instant payments.

[59] Presta, A. and Alon, S. [2014]. Large-scale graph partitioning with apache giraph.
URL: *https://code.facebook.com/posts/274771932683700/large-scale-graph-partitioning-with-apache-giraph/*

[60] *Raiden* [2020]. `https://raiden.network`.

[61] Schiper, N., Sutra, P. and Pedone, F. [2010]. P-store: Genuine partial replication in wide area networks, *2010 29th IEEE Symposium on Reliable Distributed Systems*, IEEE, pp. 214–224.

[62] Schneider, F. B. [1990]. Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys* **22**(4): 299–319.

[63] *Snapshot Sync Proposal* [2020]. `https://research.codechain.io/t/snapshot-sync-proposal/21`.

[64] *Solidity language* [2020]. `http://solidity.readthedocs.io/en/latest/index.html`.

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

[66] Teutsch, J. and Reitwießner, C. [2017]. A scalable verification solution for blockchains, `https://people.cs.uchicago.edu/teutsch/papers/truebit.pdf`.

[67] Thomas, S. and Schwartz, E. [2015]. A protocol for interledger payments, *https://interledger.org/interledger.pdf* .

[68] *Turbo-Geth programmer's guide* [2020]. `https://github.com/ledgerwatch/turbo-geth/blob/master/docs/programmers_guide/guide.md`.

[69] *Urkel Tree Implementation* [2020]. `https://github.com/handshake-org/urkel`.

[70] Wang, G., Shi, Z. J., Nixon, M. and Han, S. [2019]. Sok: Sharding on blockchain, *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp. 41–61.

[71] Wood, G. [2014]. Ethereum: A secure decentralised generalised transaction ledger, *Ethereum Project Yellow Paper* **151**.

[72] Wood, G. [2016]. Polkadot: Vision for a heterogeneous multi-chain framework, *https://pdfs.semanticscholar.org/f76f/652385edc7f49563f77c12bbf28a990039cf.pdf* .

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

[74] Zhong, Q. T. and Cole, Z. [2019]. Analyzing the effects of network latency on blockchain performance and security using the whiteblock testing platform.