Toward High-Performance Blockchains

by

Liuyang Ren

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Hans-Arno Jacobsen
Professor
Dept. of Electrical and Computer Engineering
University of Toronto

Supervisor(s): Paul Ward
Associate Professor
Dept. of Electrical and Computer Engineering
University of Waterloo

Internal Member: Guang Gong
Professor
Dept. of Electrical and Computer Engineering
University of Waterloo

Internal Member: Sagar Naik
Professor
Dept. of Electrical and Computer Engineering
University of Waterloo

Internal-External Member: Bernard Wong
Associate Professor
David R. Cheriton School of Computer Science
University of Waterloo

Author's Declaration

Statement of Contributions

This dissertation includes first-authored and peer-reviewed materials that appear in conference proceedings and a book published by the Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE), and the CRC Press.

ACM's policy on the reuse of published materials in a dissertation is as follows:

> *"Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included."*

The following list serves as a declaration of the Versions of Record for works included in this dissertation:

Portions of Chapter 1, 2, 4, and 3:
Liuyang Ren, Wei-Ting Chen, and Paul A. S. Ward. SnapshotSave: Fast and Low Storage Demand Blockchain Bootstrapping. In Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC'21). https://dl.acm.org/doi/10.1145/3412841.3441912.

Portions of Chapter 1, 2, 5, and 3:
Liuyang Ren, Paul A. S. Ward, and Bernard Wong. Improving the Performance of Blockchain Sharding Protocols with Collaborative Transaction Verification. In Proceedings of the 2021 IEEE International Conference on Blockchain (Blockchain 2021). https://ieeexplore.ieee.org/document/9680599.

Portions of Chapter 1, 2, 6, and 3:
Liuyang Ren, and Paul A. S. Ward, and Bernard Wong. Toward Reducing Cross-Shard Transaction Overhead in Sharded Blockchains. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS'22, Best Student Paper) https://dl.acm.org/doi/abs/10.1145/3524860.3539641.

Liuyang Ren, and Paul A. S. Ward. Understanding the Transaction Placement Problemin Blockchain Sharding Protocols. In Proceedings of the 2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON 2021). https://ieeexplore.ieee.org/document/9623200.

Portions of Chapter 2:
Liuyang Ren, and Paul A. S. Ward. Distributed consensus and fault tolerance mechanisms. In Book Essentials of Blockchain Technology, Chapman and Hall/CRC, 2019. https://www.taylorfrancis.com/chapters/edit/10.1201/9780429674457-1

Abstract

The decentralized nature of blockchains has attracted many applications to build atop them, such as cryptocurrencies, smart contracts, and non-fungible tokens. The health and performance of the underlying blockchain systems considerably influence these applications. Bootstrapping new nodes by replaying all transactions on the ledger is not sustainable for ever-growing blockchains. In addition, poor performance impedes the adoption of blockchains in large-scale applications with high transaction rates.

First, in order to address the bootstrapping problem of already-deployed UTXO-based blockchains, this thesis proposes a snapshot synchronization approach. This approach allows new nodes to synchronize themselves with the rest of the network by downloading a snapshot of the system state, thereby avoiding verifying transactions since the genesis block. In addition, snapshots are stored efficiently on disk by taking advantage of the system state database.

Second, although sharding improves the performance of blockchains by distributing the workload among shards, it leaves the duplicated efforts within a shard unhandled. Specifically, every node has to verify all transactions on the ledger of its shard, thus limiting shard performance to the processing power of individual nodes. Aiming to improve the performance of individual shards, this thesis proposes Collaborative Transaction Verification, which enables nodes to share transaction verification results and thus reduces the per-node workload. Dependency graphs are employed to ensure that nodes reach the same system state despite different transaction verification and execution orders.

Finally, cross-shard transactions rely on expensive atomic commit protocols to ensure inter-shard state consistency, thus impairing the performance of sharded blockchains. This thesis explores ways of lessening the impact of cross-shard transactions. On the one hand, a dependency-aware transaction placement algorithm is proposed to reduce cross-shard transactions. On the other hand, the processing cost of the remaining cross-shard transactions is reduced by optimizing the atomic commit protocol and parallelizing dependent transaction verification with the atomic commit protocol.

The above techniques are devoted to addressing the bootstrapping and performance problems of blockchains. Our evaluation shows that the first technique can significantly expedite the initial synchronization of new nodes, and the other techniques can greatly boost the performance of sharded blockchains.

Acknowledgements

I would like to thank my supervisor, Paul A. S. Ward, for his guidance, support, and patience throughout my Ph.D. studies. His constant encouragement made this thesis possible. My grateful thanks also go to professor Bernard Wong, who provided the experimental platform for me, reviewed my paper drafts, and gave feedback on my research.

I would like to extend my thanks to my thesis examining committee for reviewing this thesis, attending my defense, and providing suggestions for revision.

In addition, I would like to thank my colleagues and friends Xinan Yan, Linguan Yang, and Hua Fan for discussing research problems, methodologies, and experimental design with me. I would also like to thank W.T. Chen especially for helping me with my academic writing skills.

I would also like to thank my mother for her confidence in me throughout my post-graduate education.

Finally, I want to thank my past self who struggled to find research problems and ideas. It is her perseverance that supports me proceeding to the end.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The past 14 years have witnessed the emergence and evolution of blockchains. Blockchain applications have expanded from cryptocurrencies to smart contracts and Non-Fungible Tokens (NFTs) [103], but blockchain technology has not reached its full maturity and demands improvement in various aspects.

Compared with conventional banking systems, the core advantage of blockchains is decentralization, which enables trust authority absence and censorship resistance [44]. The cornerstone of maintaining decentralization is a large number of participants, so blockchains should encourage new nodes to join the networks. However, the current bootstrapping process of many blockchains requires new nodes to download all blocks and replay all transactions since the genesis block [78], which takes days or even weeks, depending on the hardware capacity and the network bandwidth [41][38][86].

In addition to slow bootstrapping, blockchains also suffer from low performance due to their need for every node in the chain to verify and execute all transactions. For example, the maximum throughput of Bitcoin [78] is 7 transactions per second [29][68], and the number of Ethereum [105] is not substantially better—only 15 transactions per second [7]. To improve the performance, various designs have been proposed—e.g., shortening block intervals [29][93], incorporating off-chain blocks [67] [68], allowing one miner to consecutively propose multiple blocks [39], journaling aggregated transaction effects to blockchains [32] [85], sampling nodes to participate in the consensus protocol [46], sharding [71] [62] [107] [30], etc. Among these techniques, sharding is a very promising approach. It has been widely used in distributed databases [26][24], extensively explored by blockchain researchers[71] [62] [107] [30][2], and adopted by Ethereum 2.0 [35]. The high-level idea of sharding is to partition a system into independent shards that concurrently process disjoint subsets of the total workload, so that system performance can scale with the number of nodes. However, some existing designs prevent sharded blockchains from fulfilling their full potential. First, within a shard, nodes unnecessarily repeat transaction verification. Second, the common transaction placement algorithm creates too many cross-shard transactions, which are more expensive to process than their single-shard counterparts.

1.2 Problem Statement

This thesis explores solutions to the bootstrapping and performance problems. Since sharding is one of the most promising approaches to scalable blockchains, we seek performance improvement over sharded blockchains.

The first problem addressed in this thesis is to facilitate the joining of new peers by developing an efficient bootstrapping protocol. Although the current block synchronization approach allows new peers to independently reproduce the up-to-date system state without trusting other peers, it is particularly slow and unsustainable for two reasons: 1) a new peer has to verify every transaction, which is computationally intensive, and 2) blockchains grow longer when new transactions are appended to it, which means the bootstrapping time increases over time. The challenges of designing a new bootstrapping protocol for existing blockchains include security, backward compatibility, and storage overhead minimization. Ethereum fast synchronization [48] and CoinPrune [73] both shorten bootstrapping time through snapshot synchronization, but the former does not apply to already-deployed blockchains and incurs high storage overhead, whereas the latter neither detects corrupted data efficiently nor minimizes storage usage. The protocol proposed in this thesis overcomes all the challenges.

The second problem addressed in this thesis is to improve the performance of individual shards. Although sharding removes the duplication of work between shards, it does not eliminate the unnecessary duplication of work within individual shards. Nodes in the same shard run a consensus protocol to agree on the order of the blocks, and as long as the nodes process the blocks in the agreed order, they will end up with the same system state, effectively performing state machine replication (SMR) [90][52]. When processing a block, a peer verifies and executes every enclosed transaction. Verifying transactions is two orders of magnitude more expensive than executing transactions due to the involvement of digital signatures. With various previous approaches to optimizing consensus [61][30][46][84][89] and block dissemination [107][70], transaction verification becomes a newly exposed bottleneck [95][72]. *Sharded verification* of Red Belly Blockchain [28] and *signature verification sharding* of Mir-BFT [95] both improves individual shard performance by sharing verification results, but the work in this thesis is the first attempt to incorporate transaction dependencies, which is also the main challenge of this work.

The last problem addressed in this thesis is to reduce cross-shard transaction overhead in sharded blockchains. Because each shard usually stores a disjoint subset of the system state [62] [107] [30] [2] [55], transactions modifying more than one subset inevitably incur cross-shard communication. Moreover, since blockchains operate in trustless environments, expensive digital signatures must be employed to ensure the authenticity and integrity of cross-shard messages. The communication and signature overheads make a cross-shard transaction consume more network and CPU resources than a single-shard transaction. Nonetheless, as the most common transaction placement algorithm [71][62][107][30], hashing placement creates a huge number of cross-shard transactions, e.g., 95% of Bitcoin transactions become cross-shard in a 16-shard system. With so many cross-shard transactions, sharded blockchains can hardly approach their full potential. The challenges of designing a transaction placement algorithm are twofold. First, the algorithm must be able to reduce cross-shard transactions to a very low level without causing load imbalance. Second, the algorithm should not introduce extra trust entities. Previous work on reducing cross-shard transactions either relies on additional trust points or applies only to account-balance blockchains. Thus this thesis proposes a novel transaction placement algorithm that is fully decentralized and applies to UTXO-based blockchains.

1.3 Approach

Bootstrapping

We noticed that blockchains share some similarities with Write-Ahead Logs (WAL) in database management systems (DBMS) [77] [97]. They both record changes to the system state sequentially and are replayed during either initial synchronization or recovery. However, many popular DBMS (such as Postgres, MySQL, and Oracle) deploy checkpointing to shorten the recovery time [91][19][3], but blockchains like Bitcoin are still missing this feature and are confronted with costly bootstrapping.

In this thesis, SnapshotSave, a snapshot synchronization mechanism, is introduced for the fast bootstrapping purpose. The data structures used in SnapshotSave enable old peers to efficiently create snapshots of their states and deliver snapshots to new peers. The bootstrapping protocol is carefully designed to ensure snapshot integrity. SnapshotSave also minimizes snapshot storage overhead by making use of the system state database. Specifically, it only stores a part of the snapshot that has been changed since the creation of the last snapshot. I have implemented and evaluated a prototype of SnapshotSave. The results show that, in our experimental setting, SnapshotSave reduces the synchronization time from 7.97 hours to 2.59 minutes and saves 40% of storage space. Although we describe SnapshotSave in the context of Bitcoin, it also applies to other UTXO-based blockchains.

Individual Shard Performance

To implement SMR within a shard, nodes only need to execute transactions in the same order. Verifying transactions is not necessary. In fact, the sole purpose of verifying a transaction is to determiner whether the transaction should be executed or disregarded. Therefore, this thesis proposes Collaborative Transaction Verification (CTV), which enables each peer to verify fewer transactions without compromising fault tolerance. CTV improves performance by delegating transaction verification of a block to a verification committee, which is a subset of peers in the shard. Peers in different verification committees verify transactions in parallel, so we are faced with the challenge of respecting transaction dependencies to ensure all peers reach the same system state despite different transaction verification orders. To overcome this challenge, we utilize a dependency graph of pending transactions. Another challenge is fault tolerance, i.e., faulty peers may refuse to share verification results and cause other peers to wait indefinitely. To tackle this issue, we employ independent transaction verification as a slow path and limit how long a peer waits for verification results. Evaluation results with real-world workload show that CTV can boost individual shard performance by 2.6x.

Cross-Shard Performance

We observed that transactions with (transitive) dependencies between them are more likely to be referenced together by future transactions than unrelated transactions. This is because a (transitive) transaction dependency reflects the connection between users involved in the transactions, and users with connections are more likely to be collaborate in the future than users without connections, especially considering that a user may control multiple identities and transfer cryptocurrencies between them. Based on this observation, we propose Rooted Graph Placement (RGP). RGP tends to place a transaction to the shard that includes most of its ancestor transactions so that its future descendant transactions will have a better chance of executing within a single shard.

RGP can reduce cross-shard transactions but not eliminate them, so we also devise two techniques for efficiently processing the remaining cross-shard transactions. The first technique is Dependent Transaction Pre-verification, which parallelizes the atomic commit protocol of cross-shard transactions with the signature verification of their dependent transactions. This design shortens the execution latency of the dependent transactions. The second technique utilizes the fact that RGP places most cross-shard transactions to one of their input shards. For such shards, the request for locking input UTXO(s) and the request for generating output UTXO(s) can be merged into one message, which reduces signatures and messages involved in cross-shard transaction processing.

1.4 Contributions

This thesis makes three main contributions:

1. It presents a snapshot synchronization approach that significantly shortens the bootstrapping time of new joining peers. Snapshots are stored efficiently by making use of the system state database.

2. This thesis proposes a transaction verification result sharing mechanism that can boost the performance of individual shards in sharded blockchains. It is the first work that considers transaction dependencies while allowing nodes in the same shard to verify transactions in parallel.

3. This thesis presents a novel transaction placement algorithm that can significantly reduce cross-shard transactions without introducing extra trust points. Furthermore, the cost of the remaining cross-shard transactions are lowered by two techniques developed in this thesis. These designs greatly lessen the impact of cross-shard transactions on the performance of sharded blockchains.

1.5 Organization

The rest of this thesis is organized as follows: Chapter 2 provides background to our research. Chapter 3 compares and contrasts our work with related studies. Next, Chapters 4, 5, and 6 describe the design and evaluation results of the above contributions in detail, respectively. Finally, Chapter 7 concludes this thesis and gives future research directions.

Chapter 2

Background

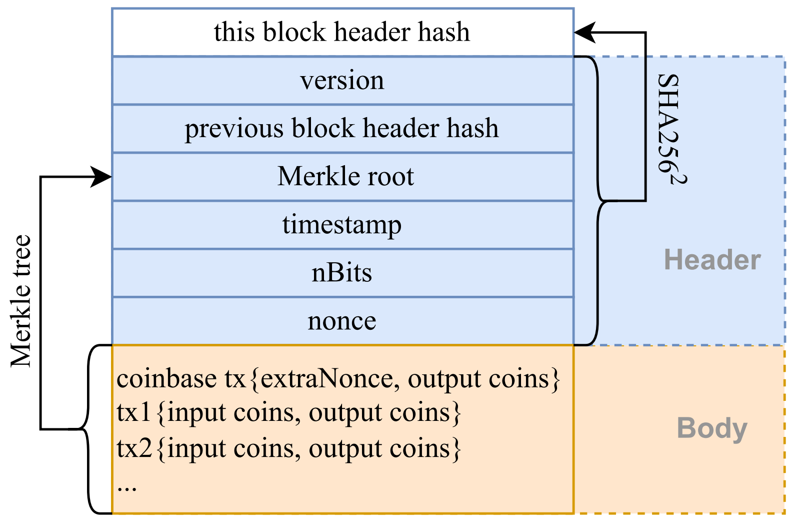
In this chapter, we will first describe conventional blockchain architectures, Proof-of-Work, and the tradeoff between consensus and performance. Then we will introduce blockchain sharding, which lays the basis for our work in Chapters 5 and 6.

2.1 Blockchain Architecture

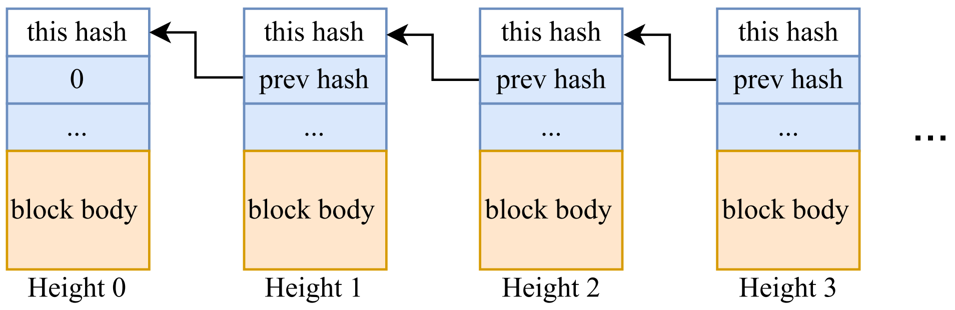
2.1.1 Block and Blockchain

A block consists of a header and a body, which is a list of transactions. A block header contains metadata about this block, particularly the previous block header hash, so that the blocks can be ordered and form a blockchain. Figure 2.1a illustrates the block structure of Bitcoin. Transactions are hashed into a Merkle tree, whose root is part of the block header. A block header hash refers to the double SHA256 hash of the block header. Assuming SHA256 is collision-resistant[40], this block structure ensures that any tampering with a header field or a transaction always results in a different block header hash.

A blockchain is a singly linked list of blocks whose *previous block header hash* fields point to their parent blocks as shown in Figure 2.1b. The height of a block refers to the distance between this block and the genesis block, whose block height is zero. A blockchain without block bodies is called a header chain.


this block header hash
version
previous block header hash
Merkle root
timestamp
nBits
nonce
coinbase tx{extraNonce, output coins}
tx1{input coins, output coins}
tx2{input coins, output coins}
...
Merkle tree
SHA256²
Header
Body

(a) Block structure



(b) Blockchain structure

Figure 2.1: Block and blockchain

2.1.2 Unspent Transaction Output (UTXO) Model

As block bodies are made up of transactions, a blockchain is essentially a ledger recording transaction history. Every node in the network executes the transactions in their apprearance order on the ledger and ends with the same system state. Unlike conventional banking systems, cryptocurrencies like Bitcoin use a UTXO model to express their system states instead of the account-balance model. Accordingly, a transaction spends input UTXOs and generates output UTXOs. Figure 2.2 demonstrates the execution of a Bitcoin transaction. Bob sends 1.5 BTC to Alice by creating a transaction that spends his 2-BTC UTXO and generates a 1.5-BTC UTXO for Alice as well as a 0.5-BTC UTXO for himself.

Once the transaction is executed, the 2-BTC UTXO (i.e., *UTXO B* in Figure 2.2) does not exist anymore. Every transaction consumes some input UTXO(s), except for coinbase transactions, which spend nothing and credit output UTXOs to nodes creating blocks.

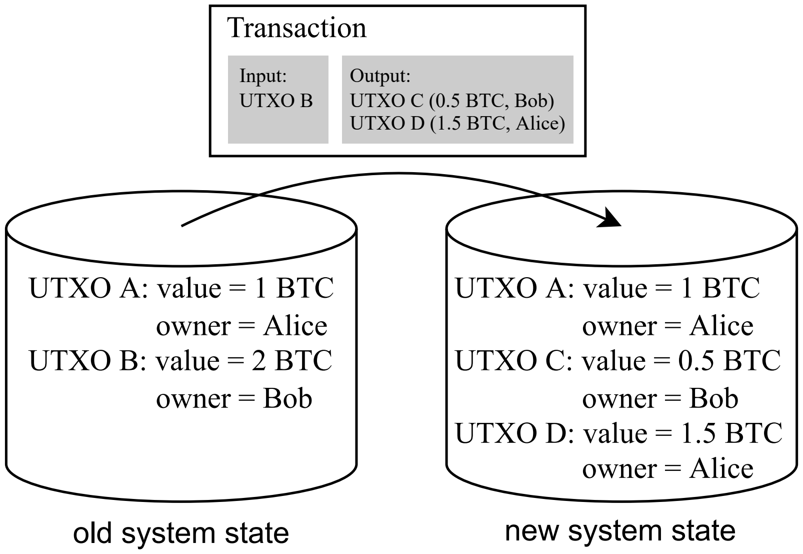


Figure 2.2: Transaction execution in the UTXO model

2.1.3 Proof-of-Work (PoW)

To prevent Sybil attacks [33], Proof-of-Work (PoW) has been invented[78]. It guarantees that the number of blocks a node can find is proportional to the computing power under its control, so forging identities does not increase the block reward received by a node. Specifically, a block is valid only if its header hash is lower than a target value:

$$\text{SHA256}^2(version||H_{prev}||Merkle\,root||timestamp||nBits||nonce) < T \tag{2.1}$$

where $\text{SHA256}^2(\cdot)$ stands for performing the SHA256 hash calculation twice; "$||$" represents the concatenation operation; H_{prev} is the previous block header hash; the other fields correspond to the block header structure in Figure 2.1a. The target value T is stored in the *nBits* field, a 32-bit scientific-notation-like representation of a 256-bit unsigned integer. For example, the current Bitcoin target value starts with 76 bits of zeros[1], so, on average,

[1]The `nBits` (labeled as `Bits` in [13]) value of Block 732770 (mined on April 20, 2022) is 386,529,497, which translates to a 256-bit hash value with 76 leading zeros.

one out of 2^{76} hash values satisfies Equation 2.1. Because hashing is irreversible, nodes have to adjust some fields and calculate the block header hash repetitively to find a valid block header hash. The three adjustable fields in a block are the *nonce* and *timestamp* in its header and the *extraNonce* in its coinbase transaction. An *extraNonce* is, in fact, a signature script. Its value can vary because a coinbase transaction does not need a valid script to redeem previous transaction output [10].

The process of appending blocks to the blockchain is referred to as *mining*, and nodes that dedicate themselves to appending blocks are called *miners*. Mining is extremely computationally intensive due to PoW [104]. Statistically speaking, the more computational resource a miner can control, the more blocks it will find. Details about the statistical analysis can be found in our previous work [88]. Miners are incentivized to mine blocks because they can put themselves as the receivers of the coinbase transactions' output UTXOs. Each block contains only one coinbase transaction, i.e., the first transaction in the block body, as shown in Figure 2.1a.

Proof-of-Stake (PoS)

One problem with PoW is that it wastes electricity and computing resources on useless hash calculations. Researchers are searching for energy-efficient alternatives [18]. Proof-of-Stake (PoS) is a possible substitute. PoS selects block proposers based on the stakes that miners hold in the system [5] [58]. While PoW is secure under the assumption that not sufficient computing power holders will collude in an attack, PoS assumes that not sufficient stakeholders wish to collude in an attack. The latter is sensible because an attack devalues the cryptocurrency and thus reduces the wealth of stakeholders, especially those with a large number of stakes. If this assumption is subverted, then there is no need for the cryptocurrency to exist because it is controlled and mostly owned by attackers. One issue with PoS is initial mining, i.e., how to mine the first block while nobody holds any stakes. PPCoin solves this issue using PoW [60]. In addition, PoS is known to be vulnerable to the nothing-at-stake problem [106], which may lead to double-spending [25] whenever there is a fork in the blockchain.

2.1.4 Peer-to-Peer Network and Fork

Each node connects to a few other nodes referred to as the peers of this node. After a miner solves the PoW puzzle, it gossips the block to all of its peers, which in turn forward the block to their peers. Malicious nodes may tamper with the block (i.e., put themselves as the output UTXO receiver in the coinbase transaction), but this will invalidate the PoW, causing others to detect the tampering. Once a node receives a valid block, it appends the block to its local copy of the blockchain.

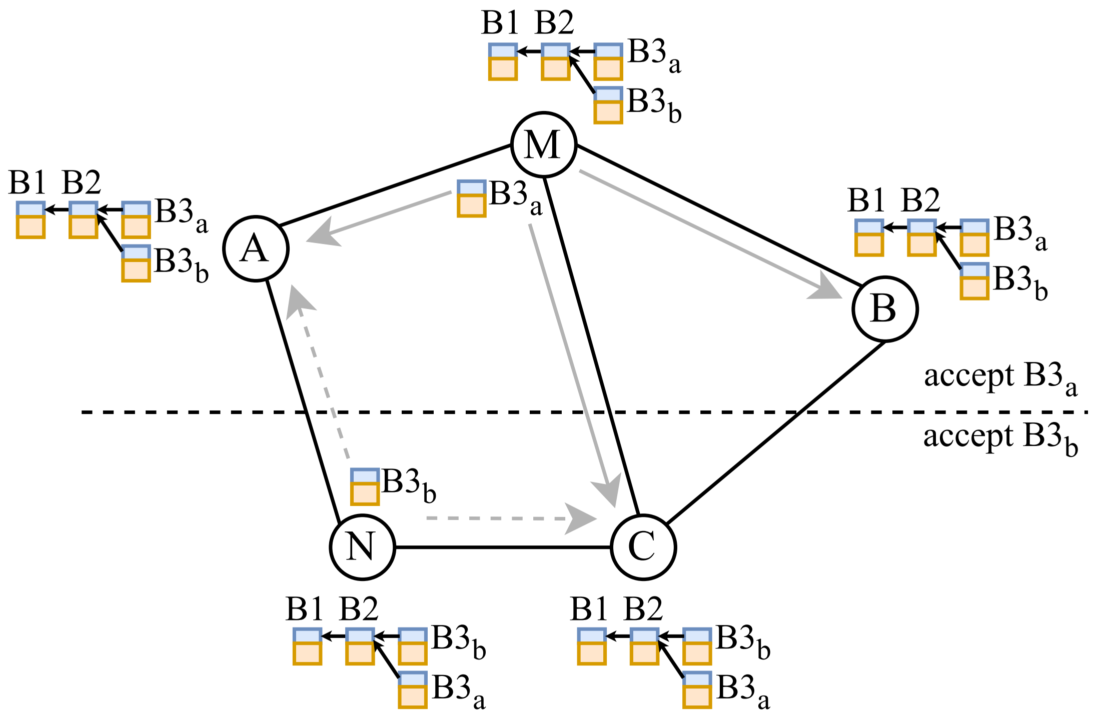


Figure 2.3: Blockchain fork

However, due to the probabilistic nature of PoW and network delay [31], there might be more than one miner finding valid blocks of the same height. Figure 2.3 illustrates such a scenario. Suppose miner M mines block $B3_a$, and at roughly the same time, miner N mines block $B3_b$. Because of the network delay, neither of the two blocks reaches all nodes before the other: block $B3_a$ reaches node A and node B earlier, but block $B3_b$ arrives at node C earlier. After the two blocks arrive at all nodes, each node receives two blocks of the same height. From the perspective of an individual node, the blockchain *fork*s off. In such cases, A node temporarily accepts the first received block. Future blocks may extend either of the two branches. If some nodes execute transactions on one of the branches while the other nodes execute those on the other branch, the states of nodes will diverge. The longest-chain rule eliminates this risk by requiring nodes to accept blocks on the longest branch. In the above example, as long as one of the two branches outgrows the other, every node will deem it the blockchain, hence a consensus.

Other reasons for blockchain forks include malicious attacks and software upgrades. An adversary can intentionally fork the blockchain to switch the longest chain to a branch benefiting himself. A software upgrade may also introduce a fork if upgraded peers reject blocks from non-upgraded peers, or vice versa. If all peers eventually agree on the same branch, the fork is called a *soft fork* [69]. Otherwise, it is called a *hard fork*.

2.1.5 Tradeoff between Consensus and Performance

In Bitcoin, the PoW target value is automatically adjusted so that, on average, one block appears per 10 minutes. Because the block size is limited to 1MB, and an average transaction takes 250 bytes, Bitcoin can achieve at most 7 transactions per second. However, the 1MB block size and the 10-minute block interval are chosen with consensus and security in consideration. We explain the tradeoff in this section and show that improving performance by adjusting the block size and the block interval has an upper bound far from satisfactory.

The two most straightforward approaches to improve throughput are 1) increasing block size and 2) reducing block interval. The former has already been taken by BitcoinCash, which increases the block size to 8MB [8]. However, if a node has not learned all blocks on the longest chain by the time it finds a valid block, it could extend an alternative branch or even create a new fork. Consequently, larger block size and faster block creation increase fork rate and attackers' chance to sabotage the blockchain under the longest-chain rule because more honest nodes waste their computational power on forking branches. In other words, increases in the block size or decreases in the block interval do not translate to linear increases in throughput as fork rate also increases, and transactions enclosed by off-chain blocks are not considered to be in the ledger.

Decker et al. established a model for the Bitcoin fork rate and proved that network propagation delay is the primary cause for blockchain forks [31]. They also verified that connecting one node to all the other nodes in the network can reduce the fork rate by 53.41%. Based on this research, Croman et al. [29] observed that there is a throughput limit of scaling blockchain systems by tuning the blockchain parameters—the block size and interval must satisfy the following inequality:

$$\frac{\textit{block size}}{X\% \;\; \textit{effective throughput}} < \textit{block interval} \tag{2.2}$$

where the metric "X% effective throughput" is defined as (block size)/(time taken for X% of the nodes to receive a full block) [29]. In 2016, 90% effective throughput of Bitcoin corresponds to merely 55 Kbps (27.5 tx/sec). The number is subject to network size and connectivity. X% effective throughput drops if new nodes join the network but connect to only a few peers. Therefore, it is a widely held view that significant throughput improvement demands changes in the fundamental consensus mechanism.

2.2 Blockchain Sharding

Figure 2.4 illustrates the architecture of sharded blockchains. The key idea of sharding is to partition nodes into shards and distribute transaction processing work among shards. Thus each shard maintains a blockchain recording its subset of transactions. Since the per-node workload drops, the throughput of the system increases. Usually, the system state is also partitioned to reduce storage pressure. A blockchain sharding protocol mainly comprises three key components: 1) a secure node partitioning algorithm, 2) an intra-shard consensus protocol, and 3) an atomic commit protocol for cross-shard transaction processing. These components will be elaborated in sections 2.2.1, 2.2.2, and 2.2.3.

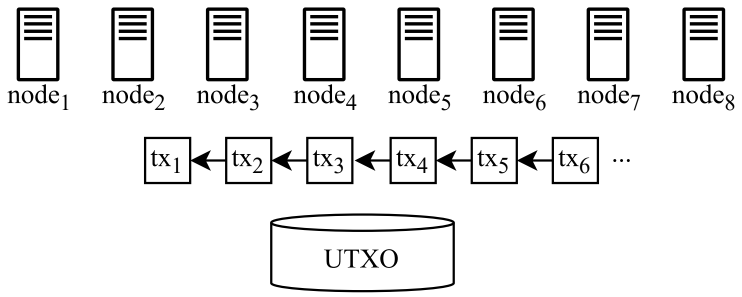
2.2.1 Node Partitioning

There are two challenges regarding node partitioning: establishing node identities in a trustless environment, and dispatching nodes to shards in a bias-resistant way (i.e., nodes should not be able to select shards depending on their preferences). The rest of this section describes the node partitioning methods used in various blockchain sharding protocols.

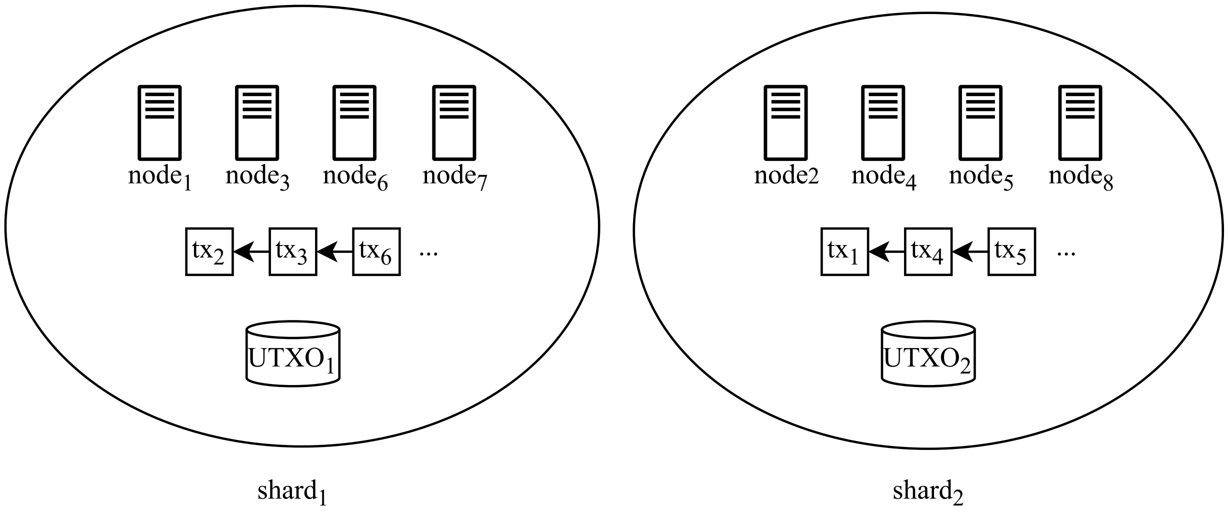
Elastico [71] is the first sharding protocol for blockchains. In Elastico, nodes first establish their identities in a Sybil-attack-resistant way, and then form shards based on randomness. In addition, shard membership is reconfigured periodically (i.e., at the start of every epoch) so that slowly-adaptive attackers do not have enough time to corrupt a whole shard [83]. Specifically, a node locally chooses its identity (`IP`, `PK`), which is an IP-address-public-key pair for authenticated communication. Since there is no PKI to trust, a node must prove to other nodes that its identity is not a Sybil in other approaches, one of which is to show that the identity is backed by some computational power (i.e., PoW). Elastico takes the PoW approach and requires each node to search for a `nonce` that satisfys the following PoW puzzle:

$$\mathrm{H}(\texttt{epochRandomness||IP||PK||nonce}) \leq \texttt{target} \tag{2.3}$$

where $\mathrm{H}(\cdot)$ stands for hashing, and `epochRandomness` is a random string generated at the end of the previous epoch to prevent `nonce` from being precomputed.



(a) Before sharding



(b) After sharding

Figure 2.4: The architecture of sharded blockchain systems

Once identities are established, nodes locally determine shard membership based on the matching between shard IDs and the last few bits of the hash values in Equation 2.3. Because of the diffusion property of hash functions, a node belongs to different shards in different epochs with high probability, so attackers cannot predict shard membership. Proof-of-Stake and Proof-of-Space [4][34] can serve as alternatives to PoW in identity establishment.

OmniLedger [62] is a later blockchain sharding protocol and addresses several challenges that Elastico leaves unsolved. One of such challenges is that Elastico's shard formation is not strongly bias-resistant, because nodes can selectively discard valid `nonce`s in Equation 2.3 in order to search for a hash value that will map them to desired shards. To address this issue, OmniLedger employs a distributed-randomness generation protocol called RandHound [98]. A random number produced by RandHound guarantees to include the contribution of at least one honest participant, so the random number is unbiasable. The output of RandHound is used for mapping nodes to shards, hence unbiasable node-to-shard assignment, which ensures that the ratio between malicious and honest nodes in any given shard approximates to the ratio across all nodes with high probability.

RapidChain [107] appears after OmniLedger. It adopts the paradigm of PoW-based identity establishment and distributed-randomness-based node-to-shard assignment. However, to reduce the communication overhead, RapidChain samples some nodes to participate in the distributed randomness generation protocol.

AHL extends sharding to permissioned blockchains. Unlike permissionless blockchains (e.g., Bitcoin [78] and Ethereum [105]), permissioned blockchains (e.g., IBM HyperLeder [20]) can control who participate in the system—nodes typically have established identities when joining the system. Thus establishing node identities is irrelevant to permissioned blockchains, but mapping nodes to shards is still a challenge. AHL leverages Trusted Execution Environments (TEEs) (e.g., Intel SGX [74]) to generate unbiased random numbers. One property of TEE is that it provides verifiable execution results for code that it protects, so nodes' behaviour cannot deviate from the protected code without being detected.

2.2.2 Intra-Shard Transaction Processing

Once shard membership is established, each shard receives and processes transactions issued by clients. All nodes in the same shard as supposed to maintain the same blockchain locally and be in the same system state, so they are essentially replicas of each other. Since a shard may contain malicious nodes that fail arbitrarily (e.g., unresponsive or equivocal), a Byzantine-fault-tolerant (BFT) consensus protocol is indispensable for a consensus among honest nodes about the blockchain content. Although PoW is able to tolerate Byzantine failures, it is not energy-efficient. More importantly, PoW does not allow blocks of different heights to be mined in parallel because the PoW puzzle of a block is only available after the previous block is mined. This serialized approach impairs the blockchain growth speed and subsequently the throughput of individual shards. On the other hand, conventional BFT protocols cannot tolerate Sybil attacks, but the node identity establishment mechanism in section 2.2.1 has already ruled out Sybil attacks, so conventional BFT protocols are applicable within shards. Most blockchain sharding protocols employ the Practical Byzantine Fault Tolerance (PBFT) [22] protocol for intra-shard consensus, so we describe PBFT followed by various optimizations for scaling it in the rest of this section.

Proposed in 1999, PBFT is the first efficient solution to the Byzantine fault tolerance in a weakly synchronous environment, e.g., the Internet. To tolerate up to f malicious nodes, PBFT requires $3f + 1$ nodes. Thus blockchain sharding protocols usually assume that the malicious node ratio in a shard is less than or equal to $\lfloor \frac{f}{3f+1} \rfloor$. In PBFT, one node is distinguished as the *primary*, and the others are *backup*s. The primary collects transactions from clients, batch them into blocks, and starts a three-phase (i.e., *pre-prepare* phase, *prepare* phase, and *commit* phase) PBFT instance for each block. Figure 2.5 illustrates the PBFT protocol in a 4-node environment (i.e., $f = 1$) where the last backup fails. Note that all messages are signed to ensure message integrity.

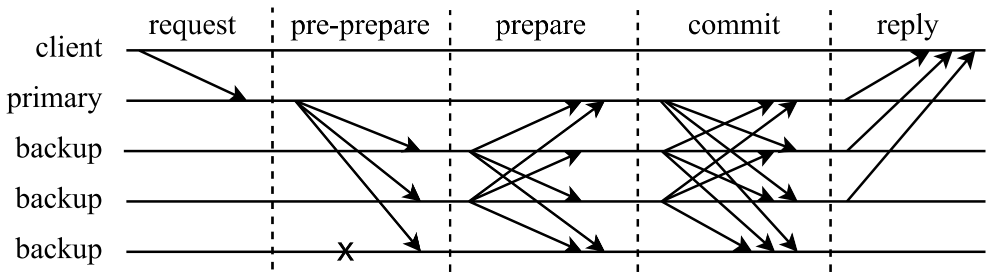


Figure 2.5: PBFT

In the pre-prepare phase, the primary multicasts a `PRE-PREPARE` message (possibly carrying a block) to backups to inform them about a proposal that binds a block to a specific height. Then in the prepare phase, backups multicast the information they received to the other nodes so that every node can detect how many other nodes have received the same information. Both the primary and the backups may equivocate, so a node must receive sufficient consistent `PREPARE` messages from distinct nodes before proceeding further. Specifically, a node keeps collecting `PREPARE` messages until the messages can compose a *prepared certificate*, which comprises one `PRE-PREPARE` message and $2f$ matching `PREPARE` messages (i.e., `PREPARE` messages must include the same proposal as in the `PRE-PREPARE` message). A prepared certificate is proof that $2f + 1$ nodes (one primary plus $2f$ backups) have learned about the proposal. Two conflicting proposals that bind two different blocks to the same height cannot both have prepared certificates because that requires at least $(2f + 1) + (2f + 1) - (3f + 1) = f + 1$ nodes to equivocate, which violates the assumption that at most f nodes are malicious. Once a node holds the prepared certificate of a proposal, the node proceeds to the commit phase of the proposal.

The goal of the commit phase is to make sure that $2f + 1$ nodes hold the prepared certificates of a proposal, which is a crucial condition for the proposal to survive primary node changes. To achieve the goal, every node multicasts a `COMMIT` message to inform other nodes that it has a prepared certificate for the proposal. Once a node has collected $2f + 1$ `COMMIT` messages, it deems the proposal as committed, which means the block is permanently bound to the block height. In other words, the block's order in the blockchain has been determined, and future blocks will not be ordered to the same height. It is worth mentioning that the PBFT instances of proposals with different block heights may proceed in parallel.

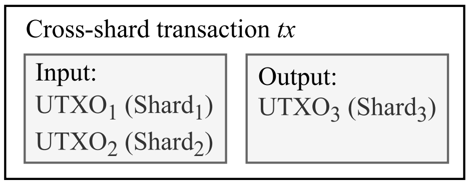
By binding blocks to distinct heights, PBFT establishes a linear order of blocks. Each node can execute transactions in the linear order and reach the same system states as other nodes. After executing a transaction, a node sends the execution results to the corresponding clients, as shown in the *reply* phase in Figure 2.5. Since malicious node may reply dishonestly, a client must waits for $f + 1$ consistent replies before trusting the execution results. The number $f + 1$ ensures that at least one honest node has executed the transaction and generated the result.

Due to the all-to-all multicast in the commit phase, the message complexity of PBFT is $\mathcal{O}(n^2)$, where n is the number of nodes. Thus PBFT is very expensive in large consensus groups [102]. Instead of applying PBFT directly like Elastico [71], other sharding protocols have proposed various optimizations to reduce the overhead of PBFT. AHL [30] utilizes TEEs to simplify PBFT. TEEs prevent nodes from equivocating, so PBFT can tolerate f malicious nodes with only $2f+1$ nodes instead of $3f+1$ nodes, hence a lower communication cost. Byzcoin [61] employs collective signing [99] and communication trees [21][101] to reduce both the signature verification overhead and communication overhead of PBFT.

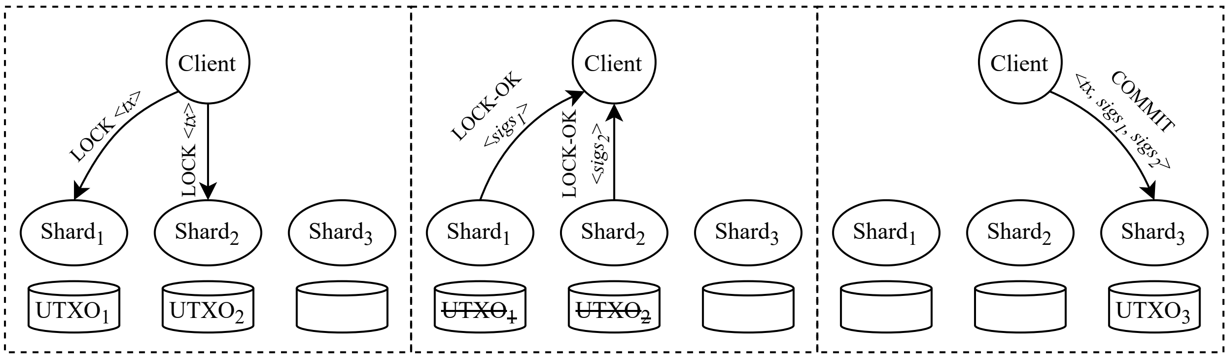
2.2.3 Cross-Shard Transaction Processing

State partitioning incurs cross-shard transactions, which modifies the state of two or more shards. A shard storing at least one input UTXO of a cross-shard transaction is called an *input shard* of the transaction. Similarly, the shard processing the transaction and storing the output UTXOs is called the *output shard* of the transaction. A cross-shard transaction has only one output shard because the output UTXOs are likely to be consumed together in the future and thus should be stored in the same shard.

To ensure state consistency between shards, cross-shard transactions rely on atomic commit protocols to be either unanimously committed or unanimously aborted. The problem of guaranteeing transaction atomicity dates back to the late 1970s [64][51]. Among various protocols, the two-phase commit (2PC) protocol [64] is the most widely used [53][92]. OmniLedger [62] adopts the "two-phase" concept and invents *Atomix* for sharded blockchains. In trustless environments, it is challenging to find an atomic commit protocol coordinator, whose misbehavior may lead to forever-locked UTXOs. OmniLedger utilizes clients as the coordinators of their own transactions so that coordinators are incentivized to conform to the protocol.



(a) A cross-shard transaction



(b) Atomix (successful scenario)

Figure 2.6: An example of OmniLedger's Atomix protocol

The basic idea of Atomix is illustrated in Figure 2.6. A client requests the input shards to lock the input UTXOs (i.e., mark the UTXOs as spent), and the input shards respond with signed lock results. If all input shards have successfully locked their respective input UTXOs, the client sends the output shard a `COMMIT` request along with signatures from the input shards as proof of successful locking. Upon receiving the `COMMIT` request, the output shard creates the output UTXO(s) in its system state, provided that all signatures of the input shards are valid. If any input shard fails to lock an input UTXO, the client requests the other input shards to unlock their respective input UTXOs with the signed response from the fail shard as proof of unsuccessful execution.

Other blockchain sharding protocols proposed various atomic commit protocols but generally follow the "two-phase" paradigm. RapidChain [107] processes a cross-shard transaction by splitting it into multiple sub-transactions, each of which spends UTXOs that reside in one shard (i.e., every sub-transaction has only one input shard). These sub-transactions first move the input UTXOs to the output shard and then complete the original transaction in the output shard. AHL [30] incorporate 2PC and leverage an entire Byzantine fault-tolerant committee as the coordinator. SharPer [2] claims that AHL cannot process cross-shard transactions in parallel due to the single coordinator committee, so it utilizes individual nodes as coordinators instead. Specifically, every node serves as its own coordinator by exchanging messages with the other nodes in all involved shards and deriving the commit decisions locally. Such decentralized approaches have also been explored by CERBERUS [54] (a series of cross-shard transaction processing protocols) and Byshard [55] (a framework for the study of sharded resilient systems). However, decentralized coordination usually incurs high message complexity. Elastico [71] is fundamentally different from other sharding protocols in that it does not partition the system state, i.e., every Elastico peer still maintains the whole system state. As a result, there are no cross-shard transactions in Elastico. Table 2.1 summarizes the above sharding protocols.

Table 2.1: Blockchain sharding protocols

Protocol	Intra-shard Consensus	Atomic commit protocol	Coordinator
Elastico	PBFT	N/A	N/A
OmniLedger	PBFT	Atomix	Client
RapidChain	Synchronous BFT	Transaction splitting	Output shard leader[1]
AHL	PBFT	2PC	Dedicated committee
SharPer[2]	PBFT	Decentralized flattened protocol	Individual nodes

[1] The PBFT primary node in the output shard of the cross-shard transaction.
[2] SharPer supports networks consisting of either crash-only or Byzantine nodes. Here we consider SharPer only in Byzantine-faulty networks, since all other sharding protocols operate under such environments.

2.3 Chapter Summary

In this chapter, we have described the architectures of conventional blockchains and sharded blockchains, both of which are crucial in understanding the rest of this thesis. The core of conventional blockchains are consensus protocols such as PoW, but they also pose a limit for performance improvement. Sharded blockchains are proposed to overcome this limit. Generally, nodes establish identities using PoW or other Sybil-attack-resistant approaches, and partition themselves based on unbiased randomness. Within a shard, nodes run a Byzantine-fault-tolerant consensus protocol such as PBFT to order transactions. To ensure state consistency between shards, various atomic commit protocols were proposed, but they are mostly derived from 2PC.

Chapter 3

Related Work

Related work on new peer bootstrapping, transaction verification result sharing, and transaction placement will be described in this chapter. The key differences between the related work and our work will be summarized.

3.1 Fast Bootstrapping

Ethereum[105], CoinPrune[73], OmniLedger [62], and Vault [66] employ snapshots to reduce the bootstrapping overhead, and so does our work. However, our design still involves different features.

Ethereum fast synchronization [48] allows new nodes to synchronize with the Ethereum network fast by downloading a recent snapshot of the system state. Vitalik Buterin foresaw the need of fast synchronization while designing Ethereum, so a Ethereum block header includes a *state root* field which is the root hash of a *merkle-patricia-tries* that stores the entire state of the system [42]. Therefore, new nodes can verify the integrity of a downloaded snapshot using the corresponding state root. However, early deployed blockchains such as Bitcoin, Litecoin [49], and Dogecoin [47] do not include a state root field in their block header structures. Our work SnapshotSave can equip such blockchains with fast bootstrapping without resorting to hard forks because we carefully place snapshot hashes in coinbase transactions, which includes a field that can hold arbitrary data. The second difference is that Ethereum creates a snapshot for every block, whereas we creates one snapshot per snapshot period (e.g., every 1000 blocks), hence less computational overhead. Last but not least, Ethereum stores multiple recent snapshots (i.e., snapshots corresponding to the latest 128 blocks [42][17]), though unchanged data are referenced instead of copied. On the other hand, our approach stores only the latest snapshot and takes advantage of the system state database to reduce storage consumption.

Our work shares the most similarities with CoinPrune [73], but CoinPrune stores the full recent snapshot, whereas we make use of the existent system state database and propose a "copy-on-write" snapshot storage strategy to avoid wasting resources on duplicated data. Furthermore, SnapshotSave downloads less data if a new peer's neighbors collectively offer a corrupted snapshot. Because CoinPrune downloads the whole snapshot before the header chain, the new peer cannot determine that a snapshot does not match the snapshot hash stored on the header chain until it downloads the whole snapshot. In contrast, SnapshotSave downloads the header chain first and checks the chunk hashes against the on-chain snapshot hash. By the time the peer started downloading chunks, it is guaranteed to have the correct chunk hashes. If a corrupted chunk is received, the new peer can detect it immediately and re-download only the problematic chunk. Lastly, in CoinPrune, after a snapshot block (referred to as *pulse block* in CoinPrune) is created, subsequent blocks within a time window have to reconfirm the snapshot to prevent adversaries from submitting invalid snapshots to the blockchain. On the other hand, SnapshotSave exploits blockchains' intrinsic tie-breaking rules (i.e., longest-chain rule) to secure snapshots.

OmniLedger [62] is a sharding protocol aiming at improving blockchain performance. In the bootstrapping part, it also employs snapshots. However, OmniLedger commits all UTXOs to the blockchain when creating a snapshot, whereas SnapshotSave commits only a snapshot hash. Like CoinPrune, OmniLedger does not utilize the system state database to avoid storing duplicate data either. Moreover, because OmniLedger is incompatible with existing blockchains, the authors did not consider backward compatibility when designing the bootstrapping strategy. Besides, the PBFT-based consensus [23][61] protects OmniLedger from the forking issue, whereas SnapshotSave is compatible with existing blockchains and incorporates fork-tolerant mechanisms.

Vault [65] [66] minimizes the bootstrapping cost for nodes in Algorand [46], which is a permissionless account-balance blockchain system. To achieve scalability, Algorand selects a committee from the total set of nodes to participate in the consensus protocol instance of a block. However, this design makes bootstrapping new nodes particularly challenging. Before trusting the system state information inside the latest block, a new node must be convinced that the peers whose signatures appear in the block's certificate are indeed the committee members chosen by cryptographic sortition. Checking committee membership of a block requires a seed from the previous block, and the seed is valid if and only if the previous block is valid. Consequently, a new node must verify the committee membership and committee signatures of all blocks before downloading the latest system state. Vault changes the seeding mechanism such that a new node only needs to very a subset of blocks. This approach is quite specific to Algorand, thus inapplicable to many other blockchains.

3.2 Transaction Verification Result Sharing

Both *sharded verification* from Red Belly Blockchain [28] and *signature verification sharding* (SVS) from Mir-BFT [95] have the same high-level idea as our Collaborative Transaction Verification (CTV)—a transaction only needs to be verified by $f + 1$ peers in the fault-free scenario. Sharded verification maps each transaction to $f + 1$ primary verifiers and f secondary verifiers. Secondary verifiers only verify the transaction when primary verifiers cannot get a unanimous verification result. In their experiments, transaction dependencies are handled on the client-side, i.e., clients wait for a parent transaction to execute before they send a child transaction. This experimental setting ensures that peers never receive a transaction whose parent transactions are pending. Therefore, sharded verification does not include a transaction dependency-tracking feature. Mir-BFT does not track transaction dependencies either.

The main contribution of both Red Belly Blockchain and Mir-BFT is on designing scalable BFT protocols, with sharded verification or SVS as an optimization. While scalable-BFT-based blockchains may be more secure than sharded blockchains, the former cannot achieve linear performance growth as peers increase. Also, compared with a whole blockchain network, individual shards are significantly smaller, so the network overhead from exchanging verification results is much more manageable. Therefore, we choose to improve sharded blockchains.

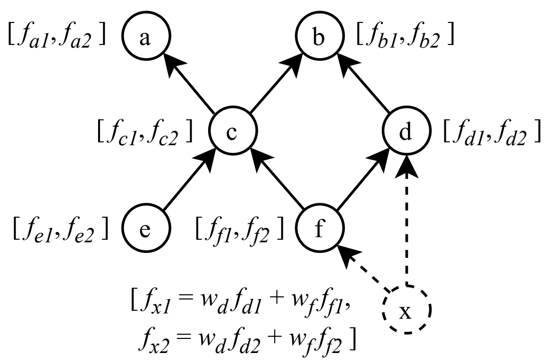
Dividing a shard further into multiple smaller sub-shards can also boost system performance, but at the cost of jeopardizing fault tolerance, especially when the original shard is already small. In contrast, CTV does not compromise fault tolerance—peers will rely on the slow path if the fast path fails due to faulty peers, which will be detailed in Section 5.2.

3.3 Cross-shard Transaction Reduction

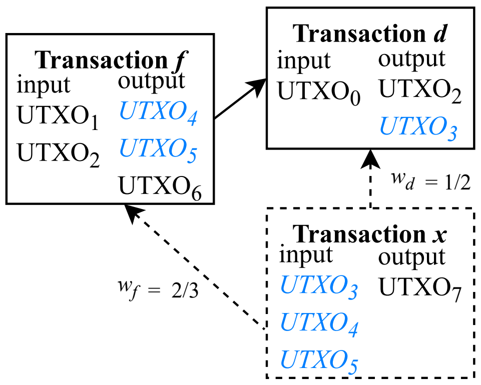
Cross-shard transaction reduction is respectively achieved through transaction placement in UTXO-based blockchains, and through account placement in account-balance blockchains. In this section, we review the previous works in both scenarios.

Transaction Placement in UTXO-Based Blockchains

The hashing placement algorithm places transactions to shards based on prefix matching between transaction IDs and shard IDs. Because transaction IDs are hash values (e.g., SHA256 outputs in Bitcoin [78]), which uniformly distribute over the output range of the corresponding hash function [96], hashing placement is equivalent to placing transactions randomly. In contrast, our Rooted Graph Placement (RGP) algorithm takes transaction dependencies into account.



(a) Transaction graph



(b) Weights of parent transactions

Figure 3.1: Principle of OptChainV2. f_{ij} is the fitness score between transaction i and the j-th shard. The fitness-score array of transaction x is an element-wise weighted sum of the fitness-score arrays of x's parents (i.e., transaction d and f). The weights (e.g., w_d) depend on what fraction of input UTXOs are from the parent transactions as shown in (b).

OptChainV2[1] [80] is a client-side transaction placement algorithm for UTXO-based sharded blockchains. To reduce cross-shard transactions, OptChainV2 builds a graph with transactions as vertices and transaction dependencies as edges. OptChainV2 associates every transaction with a fitness-score array, each element of which reflects the fitness between the transaction and the corresponding shard. Based on PageRank [82], OptChainV2 computes a child transaction's fitness-score array as an element-wise weighted sum of its parents' fitness-score arrays, as shown in Figure 3.1. To account for load balance, OptChainV2 divides fitness scores by the corresponding transaction partition sizes and requires clients to frequently sample shards for communication latency and transaction queue

[1]The original algorithm is called OptChain [79]. We have identified a shortcoming of it, leading to the authors updating their paper accordingly. We have summarized the difference between OptChain and OptChainV2 in [87].

length. There are three main differences between OptChainV2 and our RGP algorithm. First, OptChainV2 utilizes the information of all ancestor transactions since fitness scores are calculated in a top-down approach. By contrast, RGP only uses the recent ancestors but can also reduce cross-shard transaction effectively as we will see in Section 6.2.4. Second, RGP intentionally avoids shard sampling because peers are byzantine faulty and may not respond honestly. Third, OptChainV2 requires clients to honestly share the fitness score arrays of their transactions, which introduces additional trust points.

Account Placement in Account-Balance Blockchains

Hashing placement also applies to account-balance blockchains and creates many cross-shard transactions [37][59]. Generally, in an account-balance blockchain, cross-shard transactions are reduced by placing accounts that frequently transact with each other in the same shard [43][76]. To identify such accounts, Fynn et al. [43] model Ethereum transactions as a graph with accounts as vertices and transactions as edges, and evaluated multiple graph partitioning algorithms against the graph, including METIS [57] (a well-regarded offline graph partitioning algorithm) and its variants. Since account behaviour may change over time, some of the algorithms re-partition the graph periodically. Fynn et al. concludes that METIS produces the fewest cross-shard transactions but the worst load balance, whereas hashing placement is at the other extreme. Consequently, neither METIS nor hashing placement helps the system achieve the best performance. This conclusion agrees with the OptChainV2 paper, which employs METIS and hashing placement for comparison.

Chapter 4

Fast and Low-Storage-Demand Bootstrapping

In this chapter, we present SnapshotSave, a snapshot-based blockchain bootstrapping protocol that allows peers to only store a partial snapshot on the disk.

4.1 Design Overview

The high-level idea of SnapshotSave is letting a new peer download a verifiable consistent snapshot of the system state from other peers to massively reduce the number of blocks that the new peer has to replay during bootstrapping.

This idea is also inspired by the observation that the system state database of Bitcoin has a far smaller size than the ledger database. Bitcoin stores its blocks (i.e., the ledger) and the current system state (i.e., all UTXOs) in the folders named *blocks* and *chainstate* respectively, and performs data queries with the help of LevelDB [45]. The *blocks* folder is constantly more than one order of magnitude larger than the *chainstate* folder as shown in Figure 4.1, so synchronizing the system state instead of the ledger should result in significantly less data transferred. For example, at the latest block when this research is conducted (i.e., the 600k block height), the *blocks* folder is 289GB, whereas the *chainstate* folder is only 6.2GB. Moreover, the snapshot synchronization averts expensive ECDSA signature verifications [56], which mandates the original block synchronization to be usually a CPU-bound process.

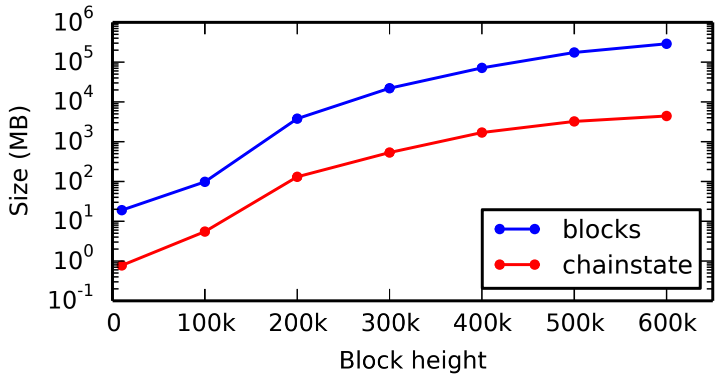


Figure 4.1: The *blocks* database size versus the *chainstate* database size

4.2 Architecture and Protocol

4.2.1 Snapshot Structure

To enable a new peer to rebuild the system state at a block height, all UTXOs existing at the block height must be transferred to the new peer. In addition, essential information about the chain state at the same height (e.g., the block header, total number of transactions up to the block height, timestamps of the last ten blocks, etc.) must also be transferred, otherwise the new peer would not be able to link future blocks to the chain. These UTXOs together with the essential chain information constitute a *snapshot*. In addition, a snapshot must be verifiable, otherwise a malicious old peer can fool a new peer by sending a corrupted snapshot to the new peer. Although placing the whole system state on the blockchain produces verifiable snapshots, this approach could easily blow up the blockchain, not to mention exceeding the block size limit in some existing blockchains. In our design, a Merkle tree[75] is built with the hash of 50k concatenated UTXOs (i.e., a chunk) as a leaf node, and the Merkle tree root is concatenated with the essential chain information hash before being hashed into the *snapshot hash*, which is committed to the blockchain. Therefore, any tampering with the snapshot always results in a different snapshot hash. Figure 4.2 illustrates the structure.

SnapshotSave sorts UTXOs before assembling chunks so that a new peer can request different chunks from different old peers without worrying that honest old peers may send inconsistent chunks. The chunk size is set to 50k UTXOs, yielding a message size of approximately 3.5 MB, which is well below the 4MB message size limit imposed by the Bitcoin Core [50]. In other words, a chunk can be transferred within one message. Moreover, using chunks instead of single UTXOs as Merkle tree leaves also reduces the leaf number, hence

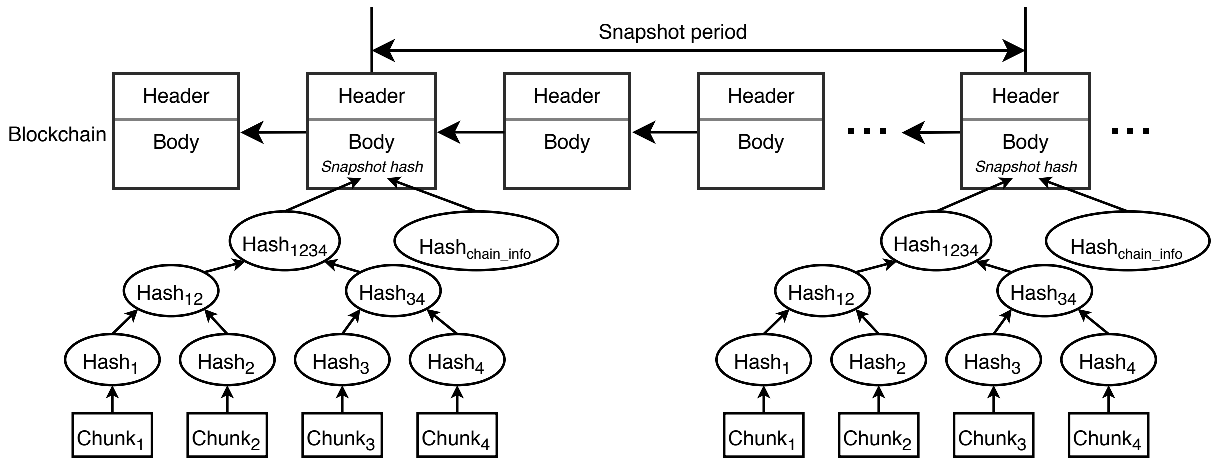


Figure 4.2: A snapshot hash is calculated based on the UTXO Merkle tree root and the essential chain information hash.

faster Merkle tree construction.

4.2.2 Snapshot Storage

Every peer in the blockchain network maintains a copy of the latest snapshot. However, SnapshotSave makes use of the *chainstate* database to avoid unnecessary storage consumption. The high-level idea is that both the UTXOs in a snapshot and the *chainstate* database can be abstracted as sets of UTXOs, and the intersection of the two sets are UTXOs that are never spent since the snapshot creation, as shown in Figure 4.3. Based on this observation, UTXOs are divided into three sets: S_{spent} (UTXOs existing in the latest snapshot but spent by later transactions), $S_{unspent}$ (UTXOs existing in the latest snapshot and not spent yet), and S_{added} (UTXOs created by later transactions and hence not in the latest snapshot). The UTXOs in $S_{unspent}$ already have their information stored in the *chainstate* database, so SnapshotSave does not bother to store them again for a snapshot. On the other hand, UTXOs in S_{spent} require extra storage because they have been deleted from the *chainstate* database.

In Bitcoin, a UTXO is identified by a ($txid$, n) pair in the way that the UTXO is the n-th output of transaction $txid$. This pair constitutes an *Outpoint* object in Bitcoin Core. The *chainstate* database is a key-value database with *Outpoint* objects as the keys and *Coin* objects as the values. A *Coin* object consists of a coinbase flag, a block height,

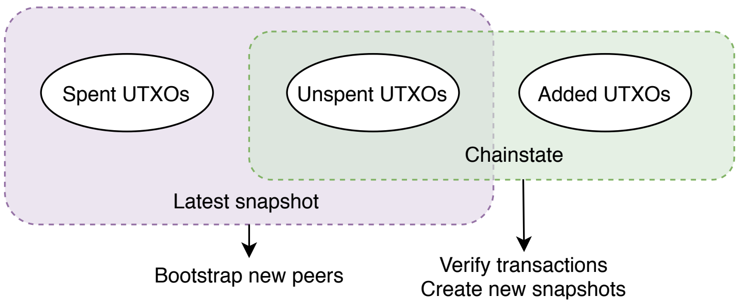


Figure 4.3: The set intersection of the latest snapshot and the *chainstate* database is the set of UTXOs never spent since the snapshot creation. The latest snapshot is used to bootstrap new peers, and the *chainstate* database is used to verify transactions and create new snapshots.

a value, and a script recording the owner (e.g., scriptPubKey). Therefore, a UTXO is comprised of a pair of *Outpoint* and *Coin* as shown in Figure 4.4.

To reconstruct a snapshot, three pieces of data are necessary and should be stored: the *Outpoint* components of UTXOs in $S_{unspent}$, the full UTXOs in S_{spent}, and the essential chain information. The *Outpoint* parts of the UTXOs in $S_{unspent}$ have to be stored because they serve as keys when fetching the *Coin* components from the *chainstate* database. Compared with storing the whole snapshot, this storage mechanism arguably demands less storage space.

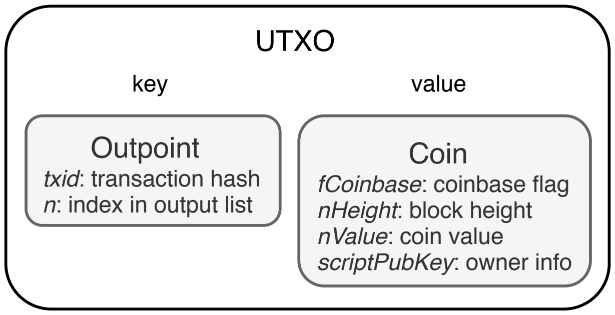


Figure 4.4: UTXO structure

Peers in the network discard old snapshots as soon as a new one matures (i.e., buried deep enough on the blockchain such that a forking is not likely to overwrite it), because bootstrapping new peers from the latest snapshot is always faster than from a previous snapshot. Fork tolerance will be discussed in Sections 4.2.5 and 4.3. Some new peers might fail to download residual chunks of the second latest snapshot due to the emergence of the next snapshot, but they can always restart synchronization by downloading the new latest snapshot provided that the interval between two snapshots is long enough for snapshot downloading.

4.2.3 Snapshot Creation

Snapshots are created periodically, and the creation is triggered by block heights. We denote the snapshot period by N_{period} (e.g., $N_{period} = 1000$ blocks), as shown in Figure 4.2, and refer to the blocks whose heights are multiples of N_{period} as *snapshot blocks*. When the next block is a snapshot block, miners create a snapshot of the system state and put the snapshot hash in the block before solving the PoW puzzle. To create a snapshot, a miner sorts all the UTXOs in the *chainstate* database, cuts them into 50k-UTXO chunks, and builds the Merkle tree. Other peers verify the snapshot hash by calculating a local snapshot hash with their local system states and comparing the local snapshot hash with the one in the received block. If the two snapshot hashes are equal, the peers append the received block to their local copies of the blockchain. Otherwise, the peers reject the block.

When a snapshot is created, S_{spent} and S_{added} are initialized as empty sets, whereas $S_{unspent}$ is initialized as all the UTXOs in the *chainstate* database. During transaction execution before the next snapshot block, S_{spent}, $S_{unspent}$, and S_{added} are constantly updated. Output UTXOs are inserted into S_{added}; input UTXOs are moved from $S_{unspent}$ to S_{spent} or just removed from S_{added}, depending on whether they exist in $S_{unspent}$ or S_{added}.

4.2.4 Snapshot Synchronization

From a related work [73], we borrow the definition of *tail blocks*—the blocks subsequent to the latest snapshot block. Transactions in tail blocks must be replayed in order to reach the latest system state. Thus bootstrapping with SnapshotSave consists of four steps: snapshot hash retrieval, chunk hash and essential chain information download, chunk download, and tail block download.

A new peer can retrieve a snapshot hash either pessimistically or optimistically. In the pessimistic approach, the new peer does not believe that the old peers that it connects to would collectively provide the correct latest snapshot hash, so it downloads the entire header chain and requests the latest snapshot block body, which contains the snapshot hash. This way, the new peer can check whether a snapshot hash is indeed on the best chain, thus avoiding being cheated by malicious peers. The new peer can identify the latest snapshot block because of the pre-defined N_{period}. In the optimistic approach, the new peer queries several old peers for the latest snapshot hash as well as the current chain length, which enables the peer to tolerate forks (see details in Section 4.2.5). As long as a majority replies consistently, the peer believes the values and continues to the next step.

In the second step, the new peer requests the snapshot metadata (i.e., chunk hashes and the essential chain information) by sending a `GetMetadata` message to one old peer. Once receiving the `GetMetadata` message, the old peer replies with a `Metadata` message enclosing the snapshot metadata. The new peer checks whether the metadata produces a hash value matching the snapshot hash in the first step. If so, the new peer continues to request all chunks in parallel with `GetChunk` messages. Otherwise, the new peer notices that the old peer has lied about the metadata and repeats step two but with a different old peer. For optimistic new peers, the first two steps can be achieved with one message by asking the old peer to piggyback the snapshot hash and the current chain length on the `Metadata` message.

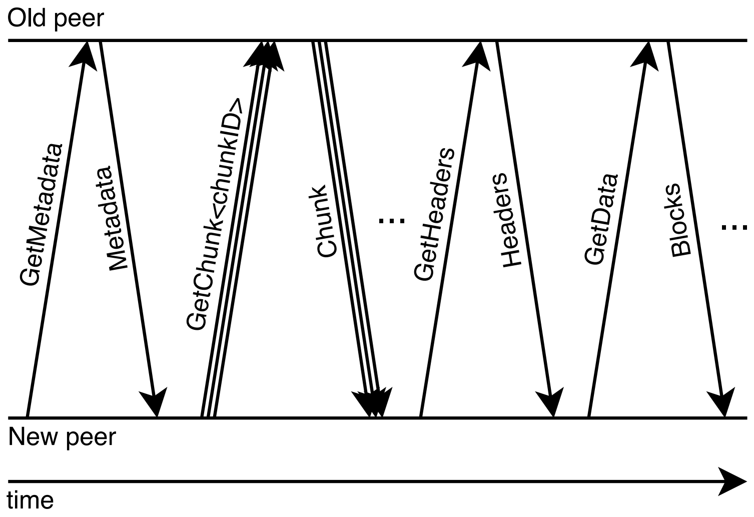


Figure 4.5: A new peer requests snapshot metadata, chunks, and tail blocks from an old peer.

Old peers respond to a `GetChunk` message with a `Chunk` message enclosing the requested chunk specified by the `chunkID` argument in the `GetChunk` message. To efficiently assemble the chunk given a `chunkID`, an old peer keeps a sorted array of *Outpoints* in the latest snapshot, but this array is a memory-only data structure since it can be easily reconstructed from the S_{spent} and $S_{unspent}$ sets. For every UTXO in the chunk, the old peer fetches the

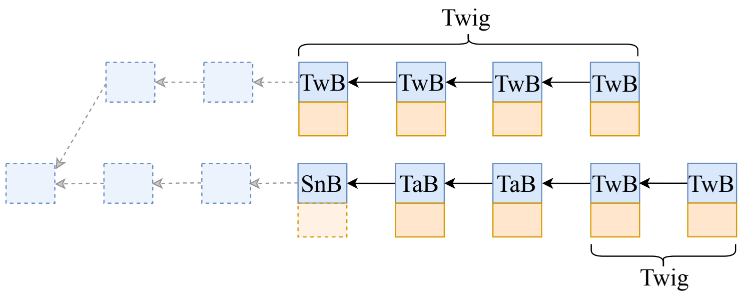
Coin component from the *chainstate* database if the UTXO is in $S_{unspent}$ or from the snapshot storage otherwise. The new peer may request chunks from multiple old peers to expedite the synchronization process. The advantage of obtaining chunk hashes before downloading chunks is that the new peer can detect corrupted chunks and only re-download the problematic chunks instead of all chunks. The new peer applies a correct chunk by adding the enclosed UTXOs to its *chainstate* database. By the time all chunks are applied, the new peer becomes ready to connect tail blocks.

A new peer must download and replay all the transactions in tail blocks so that it can have the up-to-date system state to verify future transactions. This can be achieved by performing a block synchronization starting at the snapshot block, i.e., through the `GetHeaders`, `Headers`, `GetData`, and `Block` messages of the Bitcoin protocol. Figure 4.5 summarizes the message exchanges between a new peer and an old peer from step two to step four.

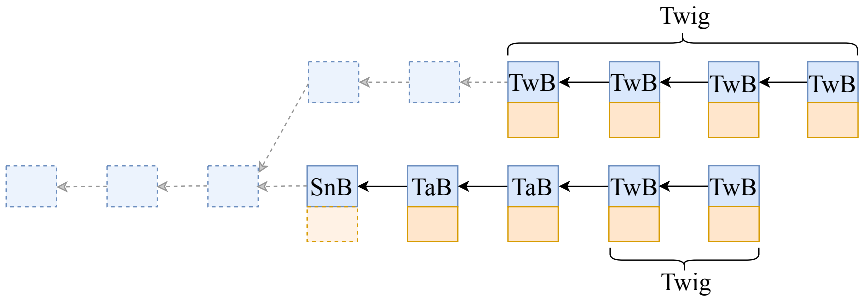
4.2.5 Fork Tolerance of Optimistic SnapshotSave

Figure 4.6 illustrates a scenario where an optimistic peer lacks the information to determine whether another branch has become the new longest chain. The peer bootstrapped itself with snapshot block on the lower branch, but later received blocks belonging to the upper branch. Because it misses the dotted headers and arrows, it cannot determine at which height the fork occurs, and whether the upper branch is longer than the lower branch.

To solve this problem, an optimistic peer keeps all blocks received since the beginning of its bootstrap, both the on-chain and off-chain blocks. Some off-chain blocks may have links between them and form a partial branch with an unknown forking point. We refer to this kind of partial branch as *twig*s. The twig of the original longest branch consists of all the blocks after the last tail block, as illustrated in Figure 4.6. Note that the peer is certain about the last tail block height because it is given the chain length at the first bootstrapping step. If a twig grows longer than the twig of the original longest branch, the peer has to download the entire header chain to assess the lengths of both branches. It is worth mentioning that this twig-length monitoring mechanism may raise false alarms. For example, the case shown in Figure 4.6a triggers header chain downloading since the upper twig outgrows the lower one by two blocks, but the lower branch remains the longest. However, this mechanism guarantees that the optimistic peer never misses a true longest-chain switch. The proof will be given in Section 4.3. In addition, the mechanism incurs no overhead because the current Bitcoin peers also keep the off-chain blocks and link them if possible. Otherwise, peers would not be aware of the lengths of other branches.



(a) a fork not altering the longest chain



(b) a fork altering the longest chain

Figure 4.6: An optimistic peer cannot differentiate the above two cases. SnB, TaB, and TwB represent a snapshot block, a tail block, and a twig block respectively.

4.2.6 Backward Compatibility

Software upgrades in permissionless blockchains are challenging, because non-upgraded and upgraded nodes may fail to reach consensus on a global longest chain, in which case the upgrade would introduce an undesirable hard fork. To avoid such a situation, SnapshotSave stores snapshot hashes in the *extraNonce* fields of snapshot blocks' coinbase transactions. This field can hold up to 96 bytes of arbitrary data because the coinbase transactions have no input UTXOs and hence no need to include a valid script redeeming any previous transaction outputs [11].

Figure 4.7 illustrates why SnapshotSave will not cause a hard fork in Bitcoin as long as upgraded peers hold more than 50% of the global hash power. When it comes to snapshot blocks, upgraded peers will reject blocks from non-upgraded peers due to the lack of valid snapshot hashes. However, non-upgraded peers will accept blocks from upgraded peers because they do not care about the values of *extraNonce* fields, where snapshot hashes exist. Therefore, as long as the branch containing snapshot hashes (the lower branch in

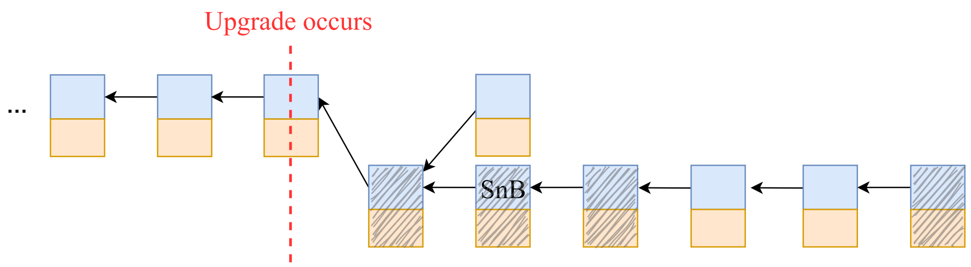


Figure 4.7: Upgrade through a soft fork. Hatched blocks are mined by upgraded peers, and other blocks are mined by non-upgraded peers.

Figure 4.7) stays longer than any other branch (the upper branch in Figure 4.7), both the upgraded and non-upgraded peers will accept it as the longest chain. In the worst case, all non-upgraded peers extend the branch without snapshot hashes, so the upgraded peers must collectively control more hash power than their non-upgraded counterparts in order to ensure that the branch containing snapshot hashes is longer, hence the 50% threshold. Once the non-upgraded peers accept the branch containing snapshot hashes, they also extend this branch, so the lower branch in Figure 4.7 includes blocks from non-upgraded peers as well.

4.3 Security

In this section, we prove that SnapshotSave is secure. Specifically, pessimistic SnapshotSave can tolerate malicious peers offering tampered chunks and blockchain forks. We make the following assumptions when deducing the security properties:

1. Hash functions are collision-resistant[40], i.e., different plaintexts always yield different ciphertexts.
2. PoW guarantees the integrity of the header chain.
3. Parent blocks reach every peer before the child blocks.

We justify the second assumption as follows. Suppose a malicious peer p_m changes a block header H to H_m and tries to mislead peer p into accepting H_m instead of H as part of the header chain. The successor of H on the header chain is denoted as H_s. According to Assumption (1), the header hash of H_m must be different than H. Given the recent PoW difficulty [13], H_m has no more than a $1/2^{76}$ probability to meet the PoW requirement. As a result, p_m has to adjust the *nonce* field of block header H_m. On average, this process takes 2^{76} double SHA256 calculation. Even if p_m makes H_m PoW-valid, the header hash of H_m must be different than H according to Assumption (1), so there is no link between H_m and H_s. Peer p_m can change the *prev hash* field of H_s to rebuild the link, but this destroys H_s's PoW validity with high probability. Therefore, replacing a block header requires solving the PoW puzzles for all its descendants. The huge computational demand prevents a peer from tampering with the header chain.

Property 1. If a pessimistic peer P_p accepts a snapshot Snp, then Snp is the authentic latest snapshot.

Proof. We prove this property by contradiction. Suppose P_p accepts Snp, and Snp is not the authentic latest snapshot. We denote the authentic snapshot as Snp_a.

Because P_p has downloaded the header chain before downloading Snp, and the header chain is not corrupted according to Assumption (2), P_p is aware of the latest snapshot block height and the correct block header. The correct block header ensures that the snapshot block body used by P_p is correct, so the block body contains the snapshot hash of Snp_a. Since P_p accepts Snp, the snapshot hash of Snp equals the snapshot hash of Snp_a.

Since Snp is not Snp_a, there are one or more differences in their UTXOs, essential chain information, or both. If their UTXOs are different, their Merkle tree leaves in Figure 4.2 would be different according to Assumption (1). The difference of leaves propagates through the Merkle tree to the root also according to Assumption (1), so the Merkle tree roots of Snp and Snp_a are different. Regardless of whether Snp and Snp_a have the same essential chain information hash, the snapshot hash of Snp is different than that of Snp_a according to Assumption (1). Similarly, if Snp and Snp_a differ in essential chain information, their essential chain information hash ($Hash_{chain\_info}$ in Figure 4.2) would be different again according to Assumption (1). Thus, no matter whether the UTXO Merkle tree root of Snp is the same as that of Snp_a, the snapshot hash of Snp and Snp_a are different according to Assumption (1). As a result, the snapshot hash of Snp does not equal the snapshot hash of Snp_a in spite of how Snp differs from Snp_a.

However, previously we deduced that the snapshot hash of Snp equals the snapshot hash of Snp_a. Thus the snapshot hash of Snp both equals and does not equal the snapshot hash of Snp_a, a contradiction.

$\square$

Property 2. If an optimistic peer P_o obtains the correct snapshot hash from its neighbors, Property 1 is also true for P_o.

Proof. We prove this property by contradiction. Suppose P_o obtains the correct snapshot hash $Hash_c$ and accepts a snapshot Snp, and Snp is not the authentic latest snapshot. We denote the authentic snapshot as Snp_a.

Since $Hash_c$ is correct, $Hash_c$ equals the snapshot hash of Snp_a. Since P_o accepts Snp, the snapshot hash of Snp equals $Hash_c$. Thus, the snapshot hash of Snp equals the snapshot hash of Snp_a.

Since Snp is not Snp_a, we have the snapshot hash of Snp not equal to the snapshot hash of Snp_a based on the proof of Property 1. Thus, the snapshot hash of Snp both equals and does not equal the snapshot hash of Snp_a, a contradiction. □

Property 3. If a peer P (either pessimistic or optimistic) bootstraps itself with a snapshot block B_{snp}, and a fork occurs at B_{snp} or a **descendant** block B_{des} of B_{snp}, then P is able to tell whether the forking branch grows longer than the originally longest branch, and, if so, change its system state to the tip of the forking branch.

Proof. Since B_{des} or B_{snp} is the forking point, the blocks on both branches are descendants of B_{snp}. According to Assumption (3), these blocks reach P after B_{snp}. Because P (no matter pessimistic or optimistic) keeps all blocks received since the beginning of the bootstrapping process, P has all blocks on both branches. By following links between these blocks starting from the tip blocks, P constructs the two branches as shown in Figure 4.8. Thus P can tell whether the forking branch is longer by counting the blocks on both branches.

Suppose the forking branch does become longer as Figure 4.8 illustrates, P has two options for updating its system state to the tip of the forking branch: 1) undo the transactions in the blocks on the original longest branch, and then execute the transactions in the blocks on the forking branch, or 2) wait for a mature snapshot block to appear on the forking branch and run the SnapshotSave protocol again. □

Property 4. If a **pessimistic** peer P_p bootstraps itself with a snapshot block B_{snp}, and a fork occurs at an **ancestor** block B_{anc} of B_{snp}, then P_p can tell whether the forking branch grows longer than the originally longest branch, and, if so, change its system state to the tip of the forking branch.

Proof. Since B_{anc} is the forking point, some blocks on the forking branch may exist before P_p bootstrapped itself. Because P_p obtains the entire header chain during bootstrapping, it has the headers of these blocks. Because P_p keeps all the blocks received since the

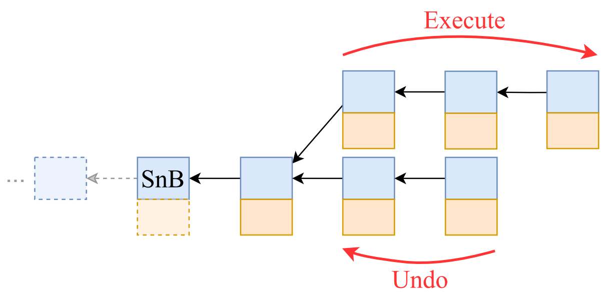


Figure 4.8: A fork occurs at the first descendant block of the snapshot block (SnB) which an optimistic peer bootstrapped itself with. The upper branch is the forking branch.

beginning of the bootstrapping process, P_p has full blocks for the remaining part of the forking branch. Thus, P_p has the block headers of all the blocks on the forking branch. Similarly, P_p has the block headers of all the blocks on the original longest branch. By following the links between these block headers starting from the tip block headers, P constructs the two header branches. Thus, P can tell whether the forking branch is longer by counting the block headers on both branches.

Suppose the forking branch does become longer, the peer has only one option to update its system state to the tip of the forking branch—wait for a mature snapshot block to appear on the forking branch, and run the SnapshotSave protocol again. The transaction-undo option in the proof of Property 3 is not available, because P_p lacks the information about the input UTXOs of transactions in blocks between B_{anc} and B_{snp}. Note that downloading the block bodies won't help because transactions only specify the *Outpoint* parts of input UTXOs, but to restore these input UTXOs, the *Coin* parts are also necessary.

□

Property 5. If (1) an **optimistic** peer P_o bootstraps itself with a snapshot block B_{snp}, and (2) a fork occurs at an **ancestor** block B_{anc} of B_{snp}, and (3) P_o obtains the correct snapshot hash and longest chain length, and (4) the forking branch grows longer than the originally longest branch, then P_o can detect that the forking branch is longer and change its system state to the tip of the forking branch.

Proof. Because of the fork-tolerant mechanism depicted in Section 4.2.5, a branch can be split into the *twig* and the *trunk* as illustrated in Figure 4.9. The two parts are separated by the moment when the peers initiate synchronization. Without loss of generality, suppose two branches br_1 and br_2 exist at the moment peer P_o initiates synchronization, and P_o

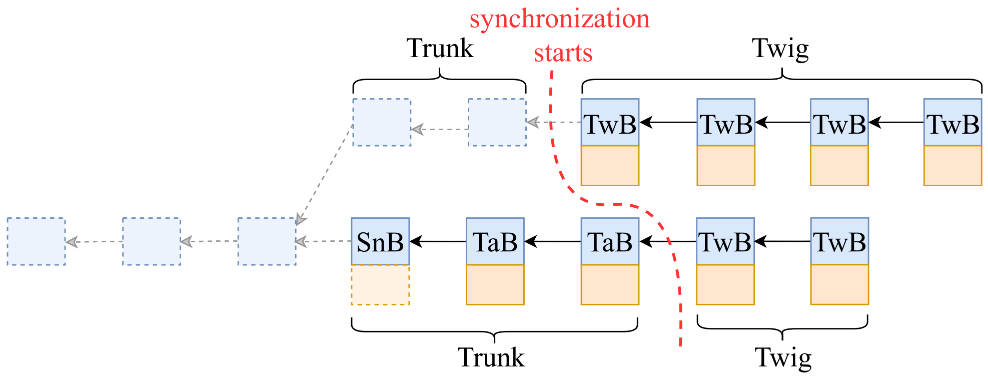


Figure 4.9: A fork occurs at an ancestor block of the snapshot block (SnB) where an optimistic peer bootstrapped itself. The start time of snapshot synchronization cuts a branch into a *twig* part and a *trunk* part.

obtains the latest snapshot hash and the length of br_1. Because of the third hypothesis, br_1 must be no shorter than br_2 at that moment. In other words, $len(tr_1) \geq len(tr_2)$, where tr_1 and tr_2 are the trunk parts of br_1 and br_2 respectively, and $len(\cdot)$ represents the length of a branch or a partial branch. The fourth hypothesis means $len(br_1) < len(br_2)$, i.e., $len(tr_1) + len(tw_1) < len(tr_2) + len(tw_2)$, where tw_1 and tw_2 are the twig parts of br_1 and br_2, respectively. Consequently, there must be $len(tw_1) < len(tw_2)$. Thus, header chain downloading is triggered due to the fork-tolerant mechanism described in Section 4.2.5. Then, P_o can judge the branch lengths and update its system state as a pessimistic peer does in the proof of Property 4. $\square$

4.4 Evaluation

A prototype of SnapshotSave is implemented based on the source code of the Bitcoin Core. We compared SnapshotSave with both Bitcoin Core's block synchronization and a related work CoinPrune[73], which is also a snapshot-based bootstrapping protocol.

4.4.1 Testbed and Experiment Design

The testing environment is created on a local cluster with Intel Xeon E5-2620v3 CPUs. We observed that the block synchronization of Bitcoin Core is a rather CPU-intensive process for a new peer. If the new peer is equipped with only one CPU, synchronizing 600k blocks would take several days. Thus, we created two 8-CPU, 32GB-memory VMs on different physical machines—one as the old peer and the other as the new peer. The network bandwidth between the two VMs is 1 Gbps.

We used the original Bitcoin Core code to download the blocks from the public P2P network. To avoid copying hundreds of gigabytes repeatedly, the old peer ran directly on the directory where the downloaded blocks were stored. During tests, neither the old nor the new peer connected to the public P2P network. They connected only to each other to ensure all resources were dedicated to our tests. For the Bitcoin Core's block synchronization test, both the old peer and the new peer write performance measurement data into the log file at the desired block heights, allowing us to collect the data from 10k to 600k block heights uninterruptedly. In contrast, for the SnapshotSave and CoinPrune tests, 10k blocks were first downloaded from the public network, and the old peer ran on the same directory to create a snapshot. We set the tail block count to 10 because the performance difference between SnapshotSave and CoinPrune would be concealed if the tail block processing dominates the synchronization time. Thus, the snapshot should be created at the 9990th block. The old peer achieved this by temporarily invalidating the 9991st block. The system state then reverts to the historic state where transactions in block 9990 have been executed. Next, the old peer creates a snapshot followed by re-considering the 9991st block. Finally, the new peer connects to the old peer and performs the synchronization. We repeats this block-downloading, snapshot-creating, and test-running process for other block heights sequentially.

4.4.2 SnapshotSave vs. Bitcoin Core

Figure 4.10 shows that pessimistic SnapshotSave is more than two orders of magnitude faster than Bitcoin Core. It significantly shortens the initial synchronization time, especially at high block heights, e.g., from 7.97 hours to 2.59 minutes (99.46% less) at the 600k block height. Furthermore, the synchronization time of SnapshotSave grows slower than that of Bitcoin Core as the block height increases. This matches the trends of database sizes displayed in Figure 4.1. Optimistic SnapshotSave achieves extremely low synchronization time at low block height because many early-stage Bitcoin blocks enclose few transactions [79], resulting in a few UTXOs. Consequently, the header chain downloading dominates the synchronization time of pessimistic SnapshotSave. On the other hand, optimistic SnapshotSave involves no header chain downloading and thus saves most synchronization time of pessimistic SnapshotSave. Nonetheless, optimistic SnapshotSave has fewer advantages at high block heights, where the header chain downloading time becomes insignificant compared with chunk downloading time and state updating time. More analysis regarding the breakdown of the synchronization time will be provided in Section 4.4.3.

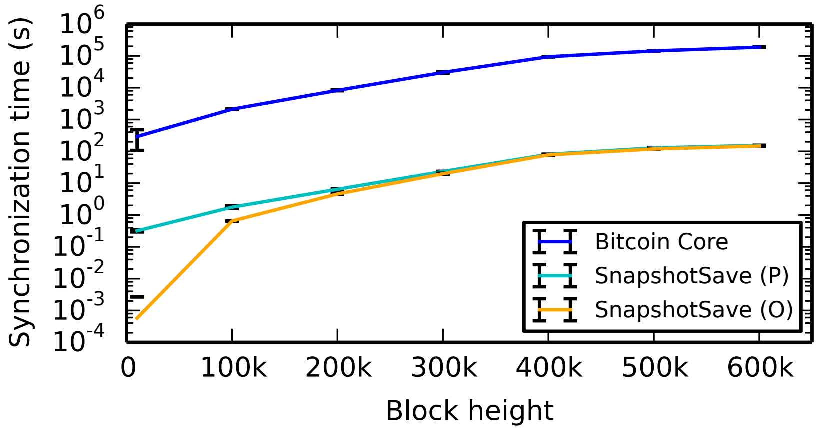
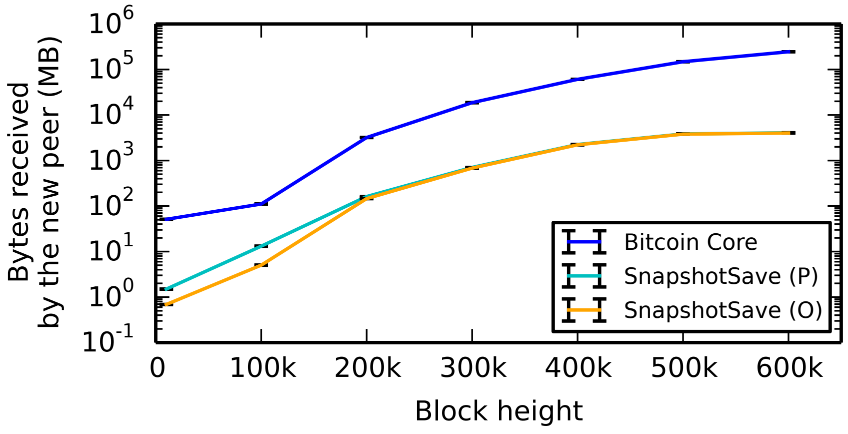
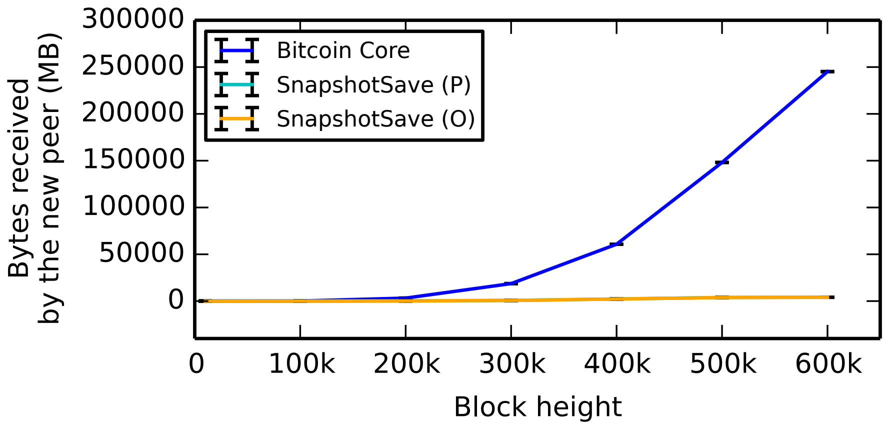


Figure 4.10: Synchronization time. SnapshotSave (P) and SnapshotSave (O) represent pessimistic SnapshotSave and optimistic SnapshotSave, respectively.

As expected, SnapshotSave also transferred significantly fewer bytes between the old and the new peers. Bytes received by the new peer are comparable in size to the *blocks* database during the Bitcoin Core synchronization, and to the *chainstate* database during the SnapshotSave synchronization. As well, the bytes received by the new peer during the SnapshotSave synchronization increase more slowly than those during the Bitcoin Core synchronization with the growth of the block height. The contrast becomes more overwhelming in Figure 4.10b, where the y-axis is on a linear scale.



(a) Logarithmic scale



(b) Linear scale

Figure 4.10: Bytes received by the new peer

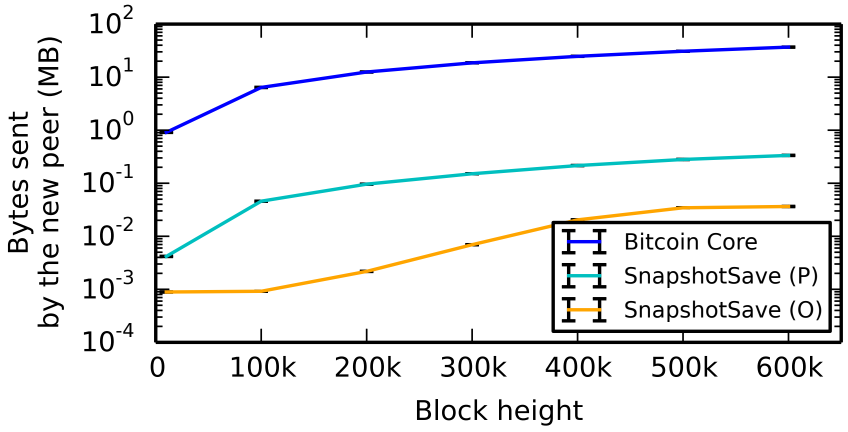


Figure 4.11: Bytes sent by the new peer

SnapshotSave also massively reduces bytes sent by the new peer. Optimistic SnapshotSave even manages to reduce the number by three orders of magnitude, as shown in Figure 4.11. This is because Bitcoin Core requires the new peer to send one `GetHeaders` message to fetch up to 2000 block headers and one `GetData` message to request one block. Though SnapshotSave also requires the new peer to send a `GetChunk` message for every chunk, the snapshot is far smaller than the full blockchain, and the old peer responds with a roughly

3.5MB chunk as opposed to a 1MB block in Bitcoin Core. The large chunk size further reduces round trips between the old peers and the new peer.

4.4.3 SnapshotSave vs. CoinPrune

CoinPrune is the closest related work to our SnapshotSave because it also employs snapshot synchronization and targets UTXO-based blockchains, as mentioned in Section 3.1. Therefore, in this section, we compare the performance and storage usage of SnapshotSave with those of CoinPrune.

Performance

Figure 4.12 shows that pessimistic SnapshotSave is constantly faster than CoinPrune, and slightly slower than optimistic SnapshotSave. At block height 500k, pessimistic SnapshotSave and optimistic SnapshotSave takes on average 10.1% and 16.8% less time compared with CoinPrune. These percentages rise to 17.2% and 21.4% at the block height of 600k.

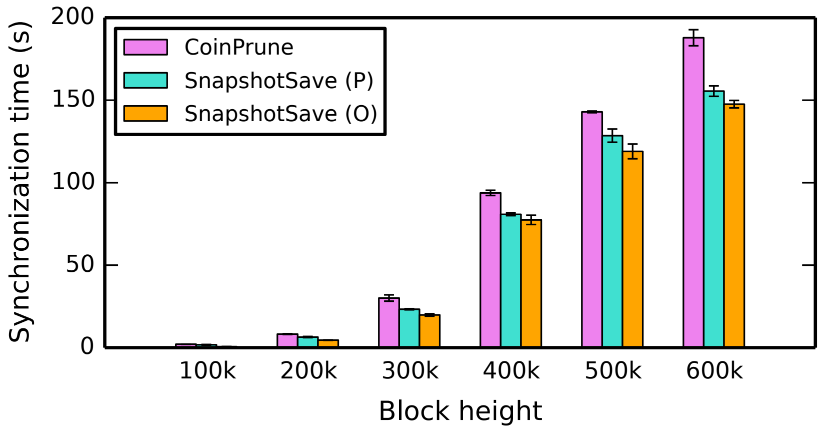


Figure 4.12: Synchronization time

Figure 4.13 gives a breakdown of the synchronization time at block height 600k. The discrepancy between CoinPrune and pessimistic SnapshotSave results from the difference in snapshot downloading time. CoinPrune caps the chunk size to 1 MB, whereas a SnapshotSave chunk contains up to 50k UTXOs (roughly 3.5 MB) as mentioned in Section 4.2.1. Unsurprisingly, the larger chunk size in SnapshotSave yields fewer chunks (Figure

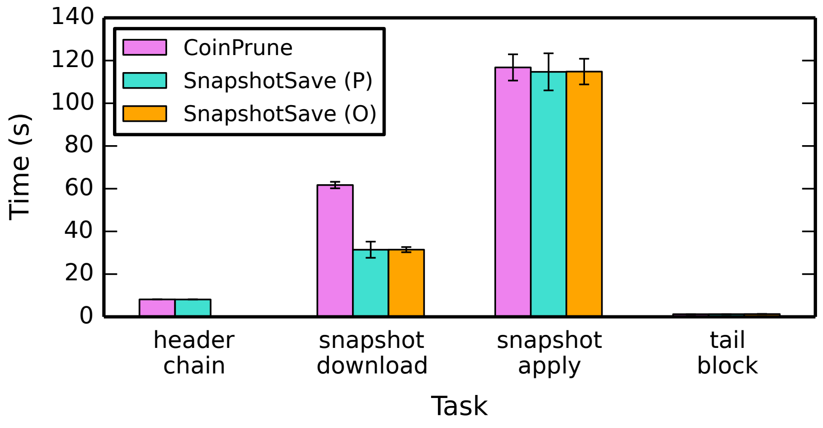


Figure 4.13: The breakdown of synchronization time

4.14) and faster snapshot downloading. On the other hand, the reason why optimistic SnapshotSave is even faster is that it skips the header chain downloading. We observed that the snapshot-applying process (i.e., adding all UTXOs in the snapshot to the new peer's *chainstate* database) becomes dominant since the block height of 300k, where the number of UTXOs starts increasing rapidly as shown in Figure 4.15.

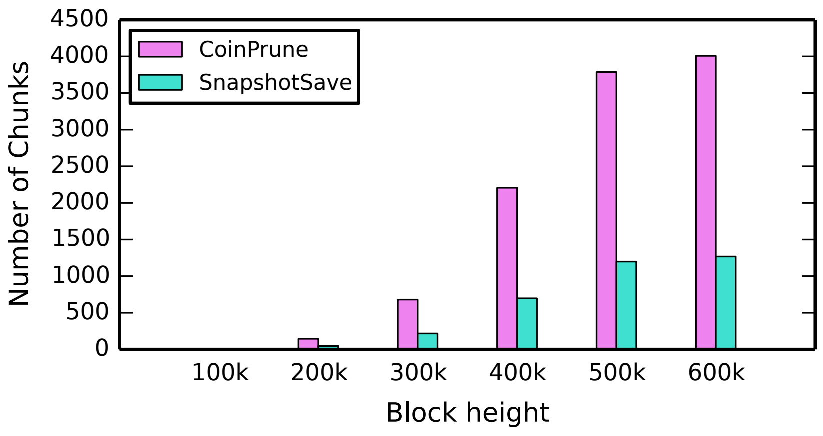


Figure 4.14: Number of Chunks

Storage Saving

Figure 4.16 demonstrates that SnapshotSave requires approximately 43% less storage space than CoinPrune. This means an old peer can dedicate less disk space for bootstrapping

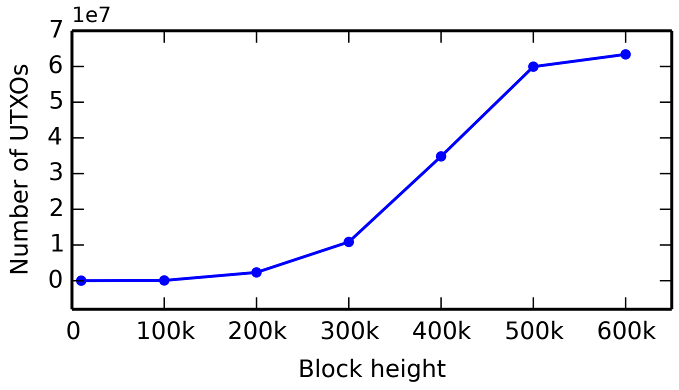


Figure 4.15: Number of UTXOs at different block heights

new peers. Because old peers store the *Outpoint* components of UTXOs in $S_{unspent}$ and the full UTXOs in S_{spent}, the storage space size will grow when UTXOs are moved from $S_{unspent}$ to S_{spent} during transaction execution. Thus we also evaluate the increase rate of the storage space size.

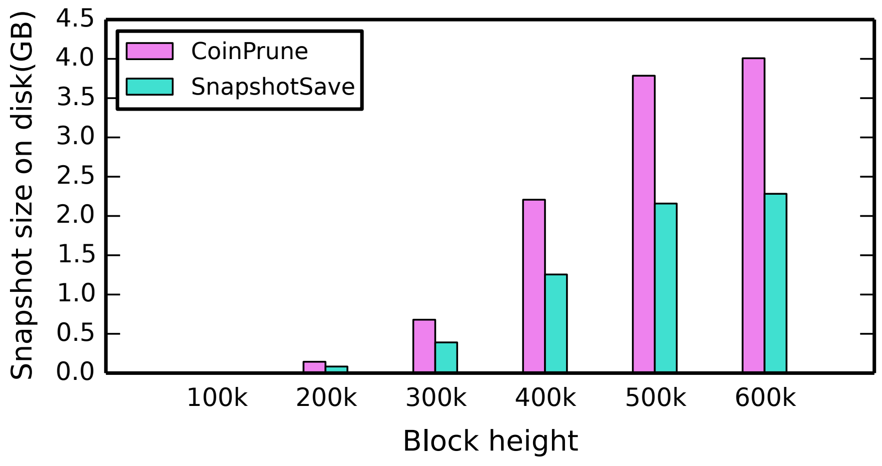


Figure 4.16: Snapshot storage demands

For the snapshot created at the 600k-th block, Figure 4.17 illustrates the variations in the storage space size, the number of UTXOs in $S_{unspent}$, and the number of UTXOs in S_{spent}. At the snapshot creation, all the 63.39 million UTXOs are in $S_{unspent}$. Transactions in the first 1k subsequent blocks consume 1.8% (1.14 million) of those UTXOs. Since these UTXOs are moved to S_{spent}, UTXO counts in the two sets always add up to 63.39 million.

It is worth mentioning that the transactions in the 1k blocks probably spend more than 1.14 million UTXOs, but some UTXOs are generated between block height 600k and 601k, and thus simply removed from S_{added}. As described in Section 4.2.2, UTXOs in S_{added} do not affect snapshot storage. Surprisingly, S_{spent} grows quite slowly. Transactions in the subsequent 10k blocks (from 600k to 610k) consume only 10% of the UTXOs in the snapshot. As a result, the snapshot storage size increased by a mere 7.6% over the 10k blocks. In addition, the increase rate of the storage size drops as the block height increases, which suggests that transactions tend to spend recently created UTXOs.

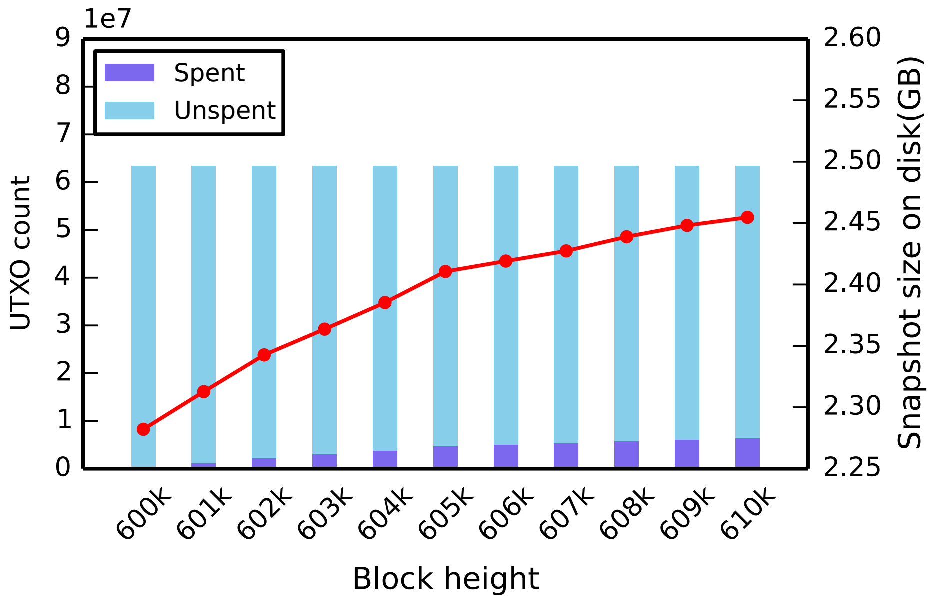


Figure 4.17: Variations in $S_{unspent}$ size and S_{spent} size for the snapshot created at the 600k block height

Intuitively, the shorter the snapshot period, the more storage space our design can save, because the storage space size increases between two snapshot blocks. For instance, if the snapshot period N_{period} is extremely long, all UTXOs in $S_{unspent}$ would eventually be moved to S_{spent}, in which case our design stores the entire snapshot and saves no storage space. However, SnapshotSave could save considerable storage space with reasonable N_{period} values. Based on the experimental results for the snapshot created at height 600k, we speculate that SnapshotSave will save approximately 42% space if one snapshot is created per week (i.e., $N_{period} = 1008$) in Bitcoin, and about 40% space if one snapshot is created per month (i.e., $N_{period} = 4320$).

4.5 Chapter Summary

Bootstrapping new peers by replaying all transactions on the blockchain is time-consuming and will be more so in the future. In this chapter, a snapshot synchronization approach is introduced for the fast bootstrapping purpose. It takes advantage of the existing system state storage to reduce the snapshot storage overhead. The evaluation results of our prototype has demonstrated that SnapshotSave can save more than 99% of the time compared to the block synchronization approach, and requires about 40% less space than storing the entire latest snapshot. The bootstrapping protocol is carefully designed to ensure security and backward compatibility. The relatively slow increase in UTXOs as opposed to blocks indicates that SnapshotSave can potentially benefit blockchains more in the future.

Chapter 5

Collaborative Transaction Verification

Blockchain sharding protocols (e.g., OmniLedger [62], RapidChain [107], and AHL[30]) usually require peers to independently verify every transaction in its shard. Therefore, the performance of a shard is bounded by the computational capacity of individual peers.

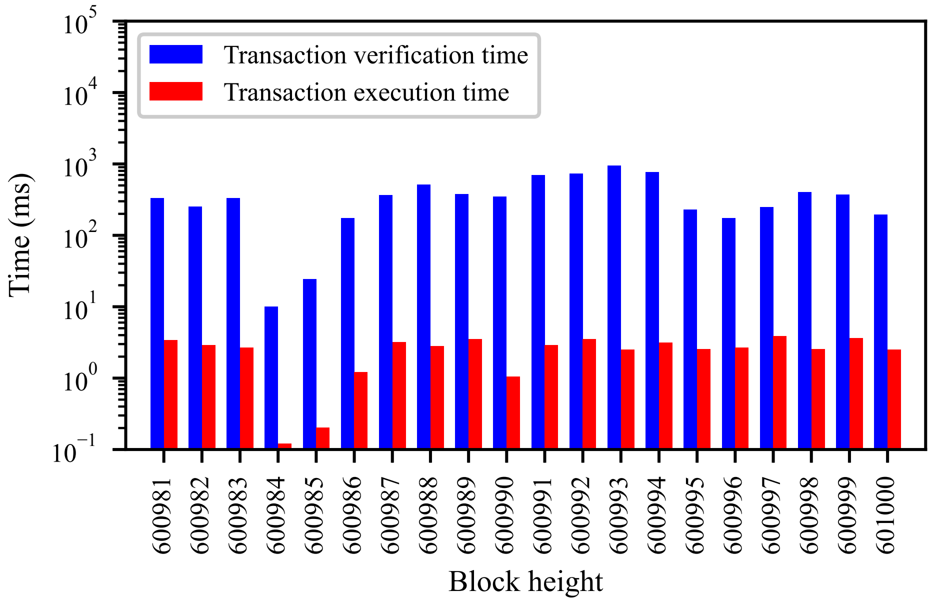


Figure 5.1: Transaction verification versus execution time

The high cost of transaction verification has been mentioned in [95][72], but no specific figures are provided to quantify the cost. We ran a Bitcoin Core node and quantitatively analyzed the processing time of 20 fairly recent Bitcoin blocks (42,582 transactions in total). Figure 5.1 shows the results. The ratio of transaction verification time to transaction execution time ranges from 64.5 to 380.6 with an average value of 150.9. In other words, transaction verification is, on average, two orders of magnitude more expensive than transaction execution. This suggests that reducing the per-peer transaction verification work will effectively lower the block processing cost. Collaborative Transaction Verification (CTV) is designed to fulfill this task and will be presented in this chapter.

5.1 Assumptions

Assumption 1. Peers verify transactions right before executing them, rather than before voting for committing or aborting the enclosing blocks. In other words, transaction verification occurs after block ordering. This model allows us to completely decouple transaction verification from the consensus protocol, and ensures that honest peers will produce the same verification result for the same transaction. On the other hand, verifying transactions in the middle of the consensus protocol, such as letting peers only multicast PBFT `PREPARE` messages if all transactions in the block are valid, may cause honest peers to disagree on the validity of the block due to dependent transactions. This is because the goal of the consensus protocol is only to produce a global order of blocks across all honest peers, but peers execute transactions at their own pace. It is possible that some honest peers have executed all parent transactions of a block and deem the block valid, while other honest peers haven't executed the parent transactions and deem the block invalid.

It is worth mentioning that, although the blockchain may include invalid transactions in the above model, peers can still reach the same state because they can all detect and discard invalid transactions during transaction verification. We believe that invalid transactions from normal clients are rare. In case malicious clients launch Denial-of-Service (DoS) attacks, the PBFT leader switches to a verify-then-propose mode when it finds a large number of invalid transactions in recent blocks.

Assumption 2. Byzantine-Fault-Tolerant (BFT) protocols, e.g., PBFT [23], HQ Replication [27], Zyzzyva [63], BFT-SMART [6], generally tolerate up to f faulty replicas ($f \geq 1$) with $3f+1$ replicas. Therefore, in blockchain sharding protocols that employ BFT for intra-shard consensus, it is a common assumption that the number of faulty peers in the shard is less than or equal to $\lfloor \frac{n-1}{3} \rfloor$, where n is the shard size (i.e., the number of peers in the shard).

5.2 Overview of CTV

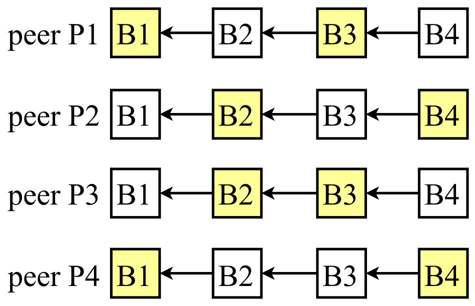
CTV makes use of the second assumption to optimize transaction verification. Specifically, the assumption guarantees that if $f + 1$ peers produce consistent verification results for a transaction, the results must be reliable, because at least one honest has verified the transaction. This is also the reason why the client in Figure 2.5 waits for only $f + 1$ consistent execution results before trusting the results. Note that the word "consistent" implies that the client may collect more than $f + 1$ results in total because inconsistent results from faulty peers are not counted.

Based on the above reasoning, we design CTV as follows. For each block, CTV optimistically selects $f + 1$ peers as the *verification committee (VC)* that are responsible for verifying all transactions in the block. Once a peer in the VC finishes verifying the transactions, it informs peers not in the VC about the verification results so that those peers can determine which transactions to execute without local verification. For simplicity, we refer to verifying all transactions in a block as verifying the block. In the fault-free scenario, a peer is expected to verify $\frac{f+1}{3f+1}$ of the blocks. We will discuss the scenario where VCs include faulty peers in the last two paragraphs of this section.

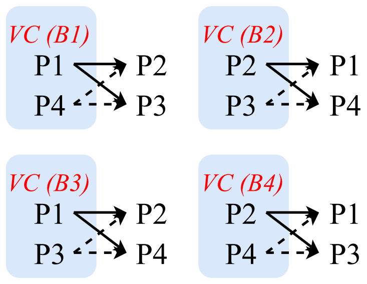
Figure 5.2 illustrates the high-level idea of CTV with an example of a four-peer shard. Blocks have been already ordered linearly by the blockchain, and every peer maintains a copy of the blockchain locally. The VC of block Bi is denoted by $VC(Bi)$. In Figure 5.2a, block Bi is highlighted at peers that are members of $VC(Bi)$. For instance, $VC(B1)$ consists of peers $P1$ and $P4$, and $P1$ is in both $VC(B1)$ and $VC(B3)$. While peers $P1$ and $P4$ verify block $B1$, $P2$ and $P3$ verify $B2$ simultaneously. Once a peer finishes verifying a block, it sends the results to peers outside the VC . For example, $P1$ sends the verification result of $B1$ to $P2$ and $P3$ (see Figure 5.2b). VC membership is calculated using the mechanism that will be described in Section 5.3.1. As a result, every peer verifies only two blocks but obtains sufficient information to execute transactions in all four blocks, so peers complete the same number of transactions faster than under independent transaction verification (ITV), as demonstrated in Figure 5.2c. Although the message exchange seems expensive, we batch verification results of transactions in a block into one message. Compared with verifying hundreds or thousands of ECDSA signatures, sending and receiving $f + 1$ messages are arguably cheaper. Moreover, message propagation is off the critical path since it happens asynchronously in parallel with the processing of subsequent blocks.

However, transaction dependencies hinder parallel transaction verification. In the example in Fig. 5.2, if a transaction tx_2 in $B2$ consumes a UTXO produced by a transaction

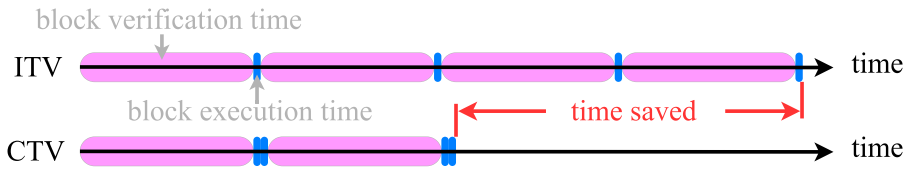
tx_1 in $B1$ or conflicts with tx_1 (i.e., $tx1$ and $tx2$ claim common UTXO(s)), then $VC(B2)$ cannot verify tx_2 until $VC(B1)$ sends over the verification result of tx_1. In both cases, tx_1 is a *parent transaction* of tx_2. A transaction can only be executed if both the following two conditions are met: 1) it is locally verified as valid, or $f+1$ peers have deemed it valid; 2) none of its parent transactions is pending. We observed that a blockchain establishes a total order of transactions, but transaction dependencies exhibit only a partial order. Therefore, peers can verify and execute transactions in distinct dependency-respecting orders and still produce the same verification results and ultimately the same system state.



(a) Peers maintain the same blockchain but are in the VCs of different blocks (highlighted).



(b) Peers in the $VC(Bi)$ send the verification result to peers outside $VC(Bi)$.



(c) A peer processes blocks faster under CTV.

Figure 5.2: Verification results sharing boosts performance.

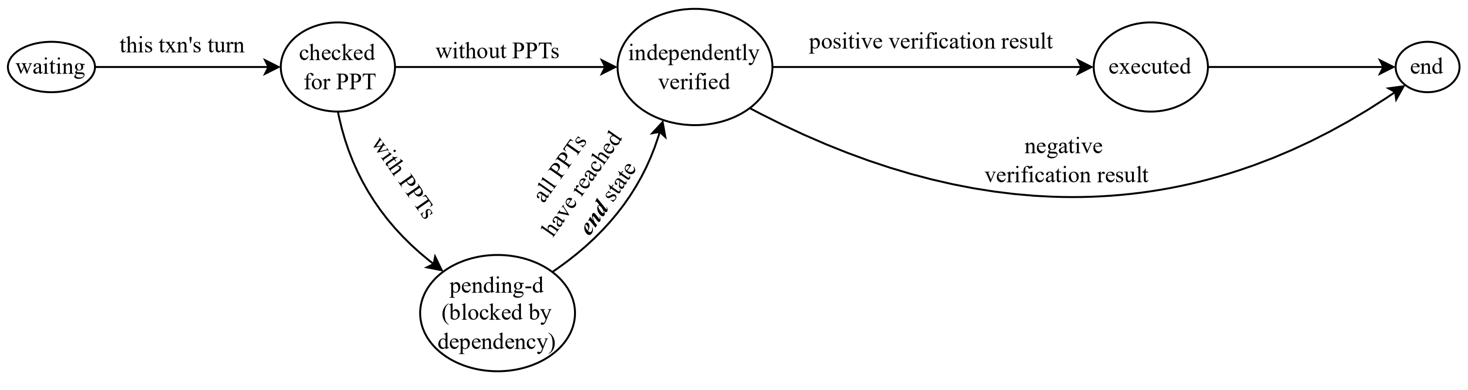
Based on the above observations, we design the high-level CTV algorithm as shown in Algorithm 1 to allow dependencies-aware verification result sharing. For better readability, “transaction” is abbreviated to “txn” in figures and algorithms in the rest of this chapter. A peer processes transactions in the order they appear on the blockchain and maintains a dependency graph to buffer temporarily unexecutable transactions. Specifically, when encountering a transaction whose VC includes the peer, the peer attempts to verify the transaction independently. However, if the transactions have *pending parent transactions* (*PPTs*), the peer has to set the transaction aside by adding it to the dependency graph. This may happen when one parent transaction is being verified by another VC. On the other hand, when encountering a transaction whose VC excludes the peer, the peer attempts to utilize the verification results from other peers to execute or discard the transaction directly. Nevertheless, two circumstances render the transaction temporarily unexecutable: 1) it has not accumulated enough verification results from other peers, or 2) it has PPTs. Under either of these two circumstances, the transaction will be added to the dependency graph for later revisiting. We refer to transactions whose processing is blocked by dependencies (i.e., blocked by PPTs) as *pending-d* transactions, and transactions whose processing is blocked by other peers (i.e., missing verification results) as *pending-p* transactions.

Algorithm 1: High-level CTV Algorithm

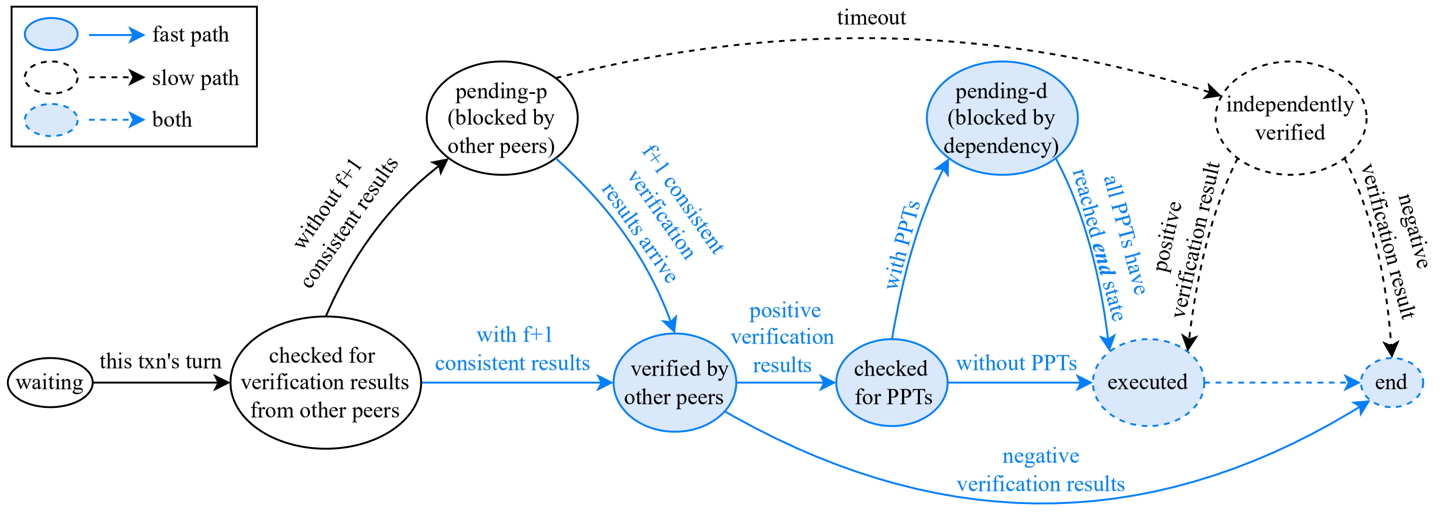
```
  Input: a stream of ordered blocks
1 while true do
2     if the next block B is available then
3         sequentially process txns in B. If pending-p or pending-d txns are met, add
            them to the dependency graph                                  // Algorithm 2
4     revisit pending-p txns that have accumulated enough verification results and
        their pending-d descendants                                       // Algorithm 3
5     revisit timeout pending-p txns and their pending-d descendants      // Algorithm 4
6     send verification results to corresponding peers
```

A *pending-p* transaction is revisited when enough verification results have arrived. If the transaction becomes eligible for execution (i.e., valid and without PPTs), it is removed from the dependency graph. The removal of a transaction will trigger the removal of its child transaction if the transaction is the last PPT of the child, so the descendant transactions are revisited as well. A *pending-p* transaction is also revisited when the peer times out waiting for the verification results. In other words, the *pending-p* transaction has been in the dependency graph for a long time. This might occur if the VC of the transaction contains faulty peers.

To assist readers to understand Algorithm 1, we provide transaction life cycles in Figure 5.3, which depicts the algorithm from the perspective of a transaction, denoted by x. In both life cycles, transaction x starts with a *waiting* state as peers process transactions sequentially. $VC(B)$ is also denoted by $VC(x)$ if block B encloses transaction x. From the perspective of x, peers can be divided into two categories: $\{P \mid P \in VC(x)\}$ and $\{P \mid P \notin VC(x)\}$. At a peer in the first category, x follows the life cycle in Figure 5.3a, where x is checked for PPTs before being verified and will enter the *pending-d* state to wait for being revisited later if it is temporarily unexecutable due to PPTs. Details about PPT detection will be given in Section 5.3.2.


waiting
this txn's turn
checked for PPT
without PPTs
with PPTs
independently verified
pending-d (blocked by dependency)
all PPTs have reached *end* state
positive verification result
executed
end
negative verification result

(a) The life cycle of a transaction at peers in its VC


fast path
slow path
both
waiting
this txn's turn
checked for verification results from other peers
without f+1 consistent results
with f+1 consistent results
pending-p (blocked by other peers)
timeout
f+1 consistent verification results arrive
verified by other peers
positive verification results
checked for PPTs
with PPTs
pending-d (blocked by dependency)
all PPTs have reached *end* state
without PPTs
executed
independently verified
positive verification result
negative verification result
end
negative verification results

(b) The life cycle of a transaction at peers outside its VC

Figure 5.3: Transaction life cycles. PPTs stand for Pending Parent Transactions.

In contrast, at a peer outside of $VC(x)$, transaction x follows the life cycle in Figure 5.3b, where it is checked for not only PPTs but also $f + 1$ consistent verification results. Verification results are checked first because there is no need to perform PPT checking if the verification results show that x is invalid. This design has two merits: 1) peers do not waste computation resources on checking PPTs for invalid transactions, and 2) the parent transactions of x have more time to be executed, so x has a lower chance to enter the *pending-d* state. The definitions of the *fast path* and the *slow path* are as follows: **fast path**—a series of state transitions that leads a transaction to the *end* state based on verification results from other peers; **slow path**—a series of state transitions that leads a transaction to the *end* state based on independent verification. If x falls into the *pending-p* state, it waits for more verification results to arrive for the next T_{slow} seconds. If enough results have arrived before T_{slow} seconds elapse, x exits the *pending-p* state and is still on the fast path. Otherwise, x takes the slow path to move toward the *end* state. Note that the fast path and the slow path only exist in the life cycle in Figure 5.3b.

It is worth mentioning that the slow path does not involve PPT checking but still respects transaction dependencies. This is because the *pending-p* state fulfills "first in, first expire". For example, suppose transaction w is ordered before x on the blockchain, and w also falls into the *pending-p* state. Because the peer processes w before x, w enters the *pending-p* state earlier than x. Since both w and x are given a time window of T_{slow} to wait for verification results, w must expire earlier than x. Therefore, by the time x expires, no transaction ordered before x is in the *pending-p* state. Section 5.3.5 will elaborate on why no transaction ordered before x can be in the *pending-d* state either. Consequently, when x expires, all transactions ordered before it must have reached the *end* state, so x is guaranteed to be free of PPTs.

Now that the algorithm is described from both a peer's perspective and a transaction's perspective, we summarize the key logic of CTV:

1. Every peer attempts to process transactions sequentially. Transactions without blocking factors (e.g., PPTs) reach the *end* state of their life cycles when they are sequential processed; whereas the other transactions are "tossed" into the dependency graph of pending transactions. A peer removes a transaction from the dependency graph and resumes processing it when the blocking conditions no longer hold.

2. A transaction can remain in the *pending-p* state for at most T_{slow} seconds. This time window allows the peer to be flexible about the verification result arrival time so that the success of the fast path is not hampered by, for example, different transaction verification orders, slow peers, or network latency. The threshold T_{slow} prevents the peer from waiting indefinitely.

3. In the fault-free scenario, if the value of T_{slow} is sufficiently high, the slow path will never be invoked. In other words, a transaction is verified only by $f+1$ peers in the fault-free scenario. The proof will be given in Section 5.4.

Lastly, we describe a design that optimizes the performance under faults. In line 6 of Algorithm 1, all independent verification results, including those for transactions taking the slow path, are sent to other peers, although the peer is outside the VC of those slow-path-taking transactions. This enables the transaction to still take the fast path at some peers. For example, suppose the VC of a transaction x includes one faulty peer and thus can only provide f consistent results to peers outside $VC(x)$. Then the first timing-out peer verifies x and sends the results to the remaining $2f-1$ peers. Thus each of the $2f-1$ peers receives $f+1$ verification results and avoids verifying x independently. Generally, when a VC can only provide r consistent verification results in time, the first $f+1-r$ timing-out peers act as substitutes for the $f+1-r$ problematic peers to share verification results. The other peers still receive $f+1$ results as if the VC consists of non-faulty peers. Since r is in the range of $[1, f+1]$, the number of timing-out peers is in the range of $[0, f]$ provided that no two peers time out simultaneously. Thus even in the worst case where a VC comprises f faulty peers, CTV is still expected to have better performance than ITV because the verification work is duplicated $2f+1$ times under CTV but $3f+1$ times under ITV.

We'd like to emphasize that, as long as Assumption 2 holds, the number of faulty peers in a VC does not affect security but only the duplication factor of the transaction verification work, so an attacker cannot trick the shard into executing an invalid transaction. If Assumption 2 is violated (i.e., the shard contains more than f peers), the BFT protocol will break down, so guaranteeing the security of CTV in this scenario is meaningless. Algorithm 2 in Section 5.3.4 will detail Line 3 of Algorithm 1; Algorithms 3 and 4 in Section 5.3.5 will detail Lines 4 and 5 of Algorithm 1, respectively.

5.3 Algorithms

This section provides detailed algorithms to complete the high-level algorithm (i.e., Algorithm 1). First, a verification committee formation algorithm and a PPT detection mechanism are introduced as the building blocks of the detailed algorithms. Then we give a concrete example to illustrate the workflow of CTV at the transaction level. Lastly, the detailed algorithms are presented with the help of the concrete example.

5.3.1 Verification Committee Formation

Shard membership is determined by the sharding protocol with all peers agreeing on it, as described in Section 2.2.1. The shard membership reconfiguration is done in a Sybil-attack-proof way (e.g., PoW-based [62] or TEE-based [30]) and does not occur frequently. With known shard members, CTV forms the VC of a block in the following approach:

1. for each peer ID, concatenate it with the block hash, then calculate the hash of the concatenated value.
2. sort the hash values.
3. peers with the lowest $f + 1$ hash values consist of the VC of the block.

The above approach has three merits. First, because hash values are uniformly distributed over the output range of the hash function[96], this approach is equivalent to randomly selecting $f + 1$ peers out of $3f + 1$ peers for each block. Thus the overall verification work is evenly distributed among peers. Second, given a block, every node derives the same VC membership since the approach is deterministic. This ensures that every node can independently determine which blocks it should verify and which peers it should share the results with. Third, the VC membership of a block is not revealed until the block becomes available, so attackers cannot predict the VC members and target their attacks at peers in the VC to slow down the verification of the block.

5.3.2 Pending Parent Transaction Detection

There are two types of parent transactions (i.e., producing or consuming the input UTXO(s) of child transactions), so we design two PPT detection mechanisms to cover both of them. A peer uses both the two mechanisms to detect all the PPTs of a transaction.

For the first type of parent transactions, identifying PPTs requires two steps: (1) identifying parent transactions and (2) determining which parent transactions are pending. The first step can be achieved by examining child transactions' input UTXO IDs, each of which is a pair of $(txid, j)$ that points to the $(j+1)$-th output UTXO of the transaction whose ID is $txid$. The first elements of the input UTXO IDs identify the parent transactions. Next, we searching the dependency graph for the parent transactions to determine whether they are pending or not. As the graph consists of pending transactions, a parent transaction is pending if and only if it exists in the graph. Although the searching seems expensive, a binary search tree (BST) can be utilized to lower the time complexity to $\mathcal{O}(n'log(n))$, where n' is the parent transaction count, and n is the pending transaction count. We expect low a n' for the vast majority of transactions because 93% of Bitcoin transactions have less than 3 parents [79].

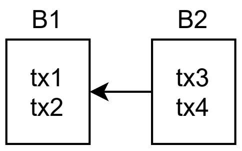
For the second type of parent transactions, we compare the input UTXO IDs of the given transaction to those of the pending transactions. Pending transactions that share common input UTXO(s) with the transaction are also PPTs.

Our detection mechanism prevents dummy transactions, which consume nonexistent UTXO(s), from lingering in the dependency graph. Suppose there is a dummy transaction tx_d. A peer in $VC(tx_d)$ judges that tx_d is free of PPTs since tx_d's parent transaction does not exist in the dependency graph, so the peer verifies tx_d immediately. The verification result must be negative because the UTXO database does not contain tx_d's input UTXO. As a result, at peers in $VC(tx_d)$, tx_d does not even manage to join the dependency graph. At a peer outside $VC(tx_d)$, tx_d may temporarily exist in the dependency graph, but will be removed as soon as the peer receives $f+1$ verification results invalidating tx_d or verifies tx_d locally in the slow path.

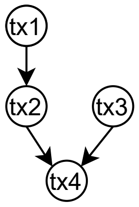
5.3.3 A Concrete Example

For better clarity, we provide a concrete example to demonstrate how CTV works before describing the detailed algorithms. In this example, a shard is given two ordered blocks, each of which comprises two transactions, as shown in Figure 5.4a. Transaction dependencies are illustrated in Figure 5.4b, where the directions of edges are from parent transactions to child transactions. We use one peer to represent one VC in this example: peer $P1$ for the $VC(B1)$ and peer $P2$ for $VC(B2)$. This allows a peer to be sure about the validity of a transaction once it receives the corresponding verification result from the other peer, hence a clear communication pattern. We denote the dependency graphs of pending transactions at $P1$ and $P2$ by $G1$ and $G2$ respectively. Messages are labeled with circled letters for easy reference.

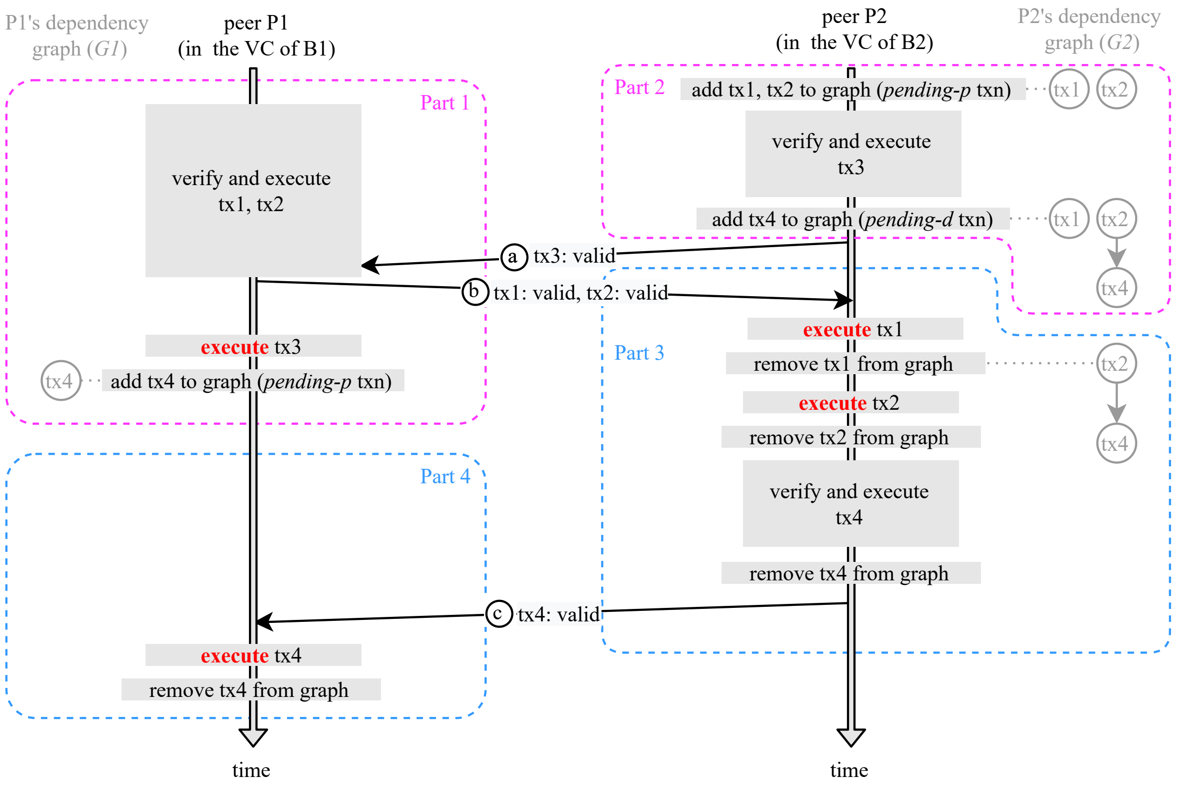
Figure 5.4c demonstrates one possible transaction processing sequence of $P1$ and $P2$. For better readability, we break Figure 5.4c into four parts. *Part 1* and *Part 2* show how the peers sequentially process transactions in the two blocks. Whenever a peer adds a transaction to its local dependency graph, the updated graph state is given next to modifying step. *Part 3* and *Part 4* depict how the peers revisit transactions in their local dependency graph and drive the transactions to the *end* states in Figure 5.3.



(a) Blocks and transactions



(b) Transactions dependencies



(c) A possible transaction processing sequence under CTV. Red "execute"s are execution without local verification.

Figure 5.4: A concrete example of transaction processing under CTV

In *Part 1*, peer $P1$ starts with verifying the two transactions in block $B1$. After sending message ⓑ to inform peer $P2$ about the verification results, $P1$ executes $tx3$ directly because message ⓐ states that $tx3$ is valid. Since $P1$ is not responsible for verifying $B2$, and there is no message indicating whether $tx4$ is valid or not, $P1$ pushes $tx4$ into $G1$. At the end of *Part 1*, $P1$ has sequentially processed all four transactions: three of them have reached the *end* state, whereas $tx4$ has paused in the *pending-p* state. $P1$ intentionally suspends the processing of $tx4$ in the hope that $tx4$'s verification result will arrive soon so that it can avoid verifying $tx4$ by itself.

In *Part 2*, $P2$ adds $tx1$ and $tx2$ into $G2$ as two *pending-p* transactions and then starts to process $B2$. Without any parent transactions, $tx3$ can be verified immediately. However, $tx4$ has two parent transactions, namely $tx2$ and $tx3$. By searching $G2$, $P2$ detects that $tx2$ is still pending, so it must postpone verifying $tx4$ to after $tx2$ finishes. Therefore, $tx4$ as well as an edge from $tx2$ to $tx4$ is added to $G2$. At the end of *Part 2*, $P2$ has sequentially processed all transactions, although only $tx3$ has reached the *end* state. $tx1$ and $tx2$ have paused in the *pending-p* state, and $tx2$ causes $tx4$ to pause in the *pending-d* state.

In *Part 3*, message ⓑ triggers the execution of $tx1$ and $tx2$ at $P2$. Removing $tx2$ from $G2$ eliminates the only PPT of $tx4$, so $P2$ subsequently verifies and executes $tx4$. In *Part 4*, $P1$ idles while waiting for the verification result of $tx4$ since there are no more blocks to process. As soon as message ⓒ arrives, $P1$ executes $tx4$.

So far, both $P1$ and $P2$ have executed all transactions but in different orders: $tx1 \rightarrow tx2 \rightarrow tx3 \rightarrow tx4$ for $P1$, and $tx3 \rightarrow tx1 \rightarrow tx2 \rightarrow tx4$ for $P2$. However, both orders respect the dependencies given by Figure 5.4b and result in the same system state. Although $P1$ idles for a while in *Part 4* in this simple example, peers will process subsequent blocks instead of idling in practice.

5.3.4 Sequential Processing Algorithm

With the above example, this section elaborates on Algorithm 2, and the next section will elaborate on Algorithm 3 and 4. Each of these algorithms fulfills a specific task in Algorithm 1. Algorithm 2 sequentially processes transactions in a block. As some transactions fall into the *pending-p* or *pending-d* state, the dependency graph gradually expands during the sequentially processing. Transactions are processed according to either the first life cycle (Line 2∼8) or the second life cycle (Line 10∼20) depending on whether the peer is a member of the VC of the block. To help Algorithm 4 detect long-pending transactions, blocks with pending transactions are added to a set (Line 22).

Algorithm 2: Sequential Transaction Processing

```
Input: a dependency graph of pending txns $G$, a block $B$, a set of txns deemed
       valid by $f+1$ distinct peers $S_{CTV\_valid\_txn}$, a set of txns deemed invalid by
       $f+1$ distinct peers $S_{CTV\_invalid\_txn}$, a set of blocks that include pending
       txns $S_{pblock}$
1  if this peer is in $VC(B)$ then
2      for $tx \in B$ do
3          if $tx$ has pending parent txns then
4              add $tx$ to $G$ as a pending-d txn
5          else
6              $isValid \leftarrow$ verify $tx$
7              if $isValid$ then
8                  execute $tx$
9  else
10     for $tx \in B$ do
11         if $tx \in S_{CTV\_valid\_txn}$ then
12             if $tx$ has pending parent txns then
13                 add $tx$ to $G$ as a pending-d txn
14             else
15                 execute $tx$
16             delete $tx$ in $S_{CTV\_valid\_txn}$
17         else if $tx \in S_{CTV\_invalid\_txn}$ then
18             delete $tx$ in $S_{CTV\_invalid\_txn}$
19         else
20             add $tx$ to $G$ as a pending-p txn
21     if at least one txn of $B$ is in $G$ then
22         add $B$ to $S_{pblock}$ with a timestamp
```

When being pushed into the dependency graph, *pending-p* and *pending-d* transactions are treated differently. A *pending-d* transaction' dependencies on other pending transactions are added as its incoming edges, whereas a *pending-p* transaction' dependencies are ignored. For instance, in the concrete example, when $tx2$ is added to $G2$ as a *pending-p* transaction, its dependency on $tx1$ is not added to the graph. Therefore, although *pending-p* transactions may become parents of future transactions, they are never child transactions. This design not only reduces the number of edges in the dependency graph but also saves some computational work. Suppose an edge from $tx1$ to $tx2$ was added to $G2$, and ⓑ did **not** include the "tx2: valid" part. Then when $tx1$ is removed, the peer would follow the edge and decrement $tx2$'s PPT counter. However, this operation would be meaningless because $tx2$'s validity remains unresolved and thus could not be executed regardless of its PPT count. With our design, dependency graphs at all the peers have the following property:

Property 1. Source vertices are *pending-p* transactions, and other vertices are *pending-d* transactions.

5.3.5 Revisiting Algorithms

Algorithms 3 and 4 revisit pending transactions in the dependency graph and evict eligible ones. Algorithm 3 revisits *pending-p* transactions that have accumulated enough verification results as well as their descendant transactions. As these transactions progress toward the *end* state, they may get removed from the dependency graph, so the graph gradually shrinks during the revisiting. All the revisited *pending-p* transactions take the fast path after exiting the *pending-p* state. However, *pending-p* transactions may fall into the *pending-d* state at PPT checking, in which case they are "upgraded" to *pending-d* transactions: edges connecting them with their PPTs are added to the graph (Line 3). Line 1∼3 of Algorithm 3 and Line 11∼13 of Algorithm 2 jointly provide the dependency graph at a peer with the following property:

Property 2. Child transactions whose VCs exclude the peer must be valid because they are in the *pending-d* state of the second life cycle.

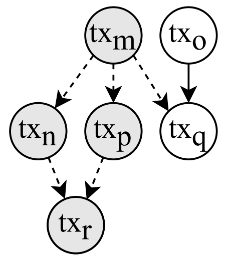
On the other hand, PPT-free valid *pending-p* transactions are executed immediately at Line 5. This execution may render some descendant transactions free of PPTs, so the function `remove_descendants` is invoked to examine the descendant transactions (Line 6). The function takes a transaction as the argument and performs breadth-first search (BFS) to traverse the subgraph rooted at the transaction. Whenever a free-of-PPT transaction is encountered, the peer processes and removes the transaction from the graph. Because such descendant transactions are *pending-d* transactions as per Property 1, the peer resumes processing them from the *pending-d* states in the two life cycles. Specifically, descendant transactions whose VCs include the peer are verified; descendant transactions whose VCs exclude the peer are executed directly due to Property 2.

Suppose the dependency graph contains six transactions as shown in Fig. 5.5a, and tx_m is passed to the `remove_descendants` function as *root*. The function first follows the

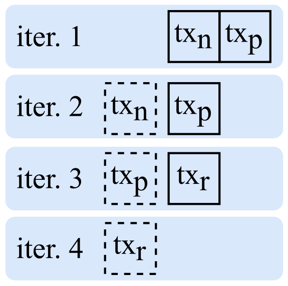
Algorithm 3: Revisiting *Pending-p* Txns with Enough Verification Results

```
Input: $G$, $S_{CTV\_valid\_txn}$, and $S_{CTV\_invalid\_txn}$ as in the input list of Algorithm 2
1  for $tx \in S_{CTV\_valid\_txn} \cap G$ do
2      if tx has pending parent txns then
3          upgrade $tx$ to a pending-d txn
4      else
5          execute $tx$
6          remove_descendants($tx$)
7      delete $tx$ in $S_{CTV\_valid\_txn}$
8  for $tx \in S_{CTV\_invalid\_txn} \cap G$ do
9      remove_descendants($tx$)
10     delete $tx$ in $S_{CTV\_invalid\_txn}$
   /* traverse the subgraph rooted at root with BFS */
11 Function remove_descendants($root$):
12     $Q \leftarrow$ an empty queue
13     for $tx_c \in$ the child transactions of root do
14         if $tx_c$ has no other pending parent txn then
15             enqueue $tx_c$ to $Q$
16     delete $root$ in $G$
17     while $Q$ is not empty do
18         $curTx \leftarrow$ dequeue the first element of $Q$
19         $B \leftarrow$ the block enclosing $curTx$
20         if this peer is in $VC(B)$ then
21             $isValid \leftarrow$ verify $curTx$
22             if $isValid$ then
23                 execute $curTx$
24         else
25             execute $curTx$
26         for $tx_c \in$ the child transactions of $curTx$ do
27             if $tx_c$ has no other pending parent txn then
28                 enqueue $tx_c$ to $Q$
29         delete $curTx$ in $G$
```

outgoing edges of tx_m and finds three children, namely tx_n, tx_p, and tx_q, but enqueues only the first two children because the last one has another parent tx_o. Figure 5.5b reflects the state of the BFS queue with each row corresponding to one iteration. In each of the subsequent iterations, one transaction is popped out of the queue, and its free-of-PPT children are pushed in. The popped-out transaction resumes following its life cycle to the *end* state and are removed from the graph. In the second iteration, tx_n is popped out, but its only child tx_r is not pushed in because the other parent of tx_r is still pending. In the next iteration, tx_p is popped out, and tx_r is finally enqueued. Eventually, all grey vertices and dotted edges in Figure 5.5a are removed.



(a) dependency graph



(b) BFS queue states

Figure 5.5: Removing tx_m triggers the removal of tx_n, tx_p and tx_q. Dotted transactions in (b) are dequeued.

Algorithm 4 also revisits pending transactions and reduces the dependency graph size but only deals with long-standing *pending-p* transactions as well as their descendant transactions. All *pending-p* transactions processed by this algorithm take the slow path. To detect long-pending transactions, the peer relies on the set of blocks containing pending transactions (i.e., S_{pblock}), which is gradually built by Algorithm 2. Because blocks are processed in order, the lowest-height block in S_{pblock} must contain the longest-pending transactions. Therefore, if T_{slow} time has elapsed since this block is added to S_{pblock}, all pending transactions in the block are verified in order at Line 3∼7. PPT checking is not needed because there are no pending transactions in previous blocks. For this very reason, the first pending transaction in the block must be in the *pending-p* state. Algorithm 4 also calls the `remove_descendants` function to examine descendant transactions (Line 7).

Algorithm 4: Revisiting Timeout *Pending-p* Txns

```
Input: $G$ and $S_{pblock}$ as in the input list of Algorithm 2, slow path triggering
       timeout $T_{slow}$
1 $B_{1st} \leftarrow$ the lowest-height block in $S_{pblock}$
2 if $B_{1st}$ has been in $S_{pblock}$ for at least $T_{slow}$ time then
3     for $tx \in B_{1st} \cap G$ do
4         $isValid \leftarrow$ verify $tx$
5         if $isValid$ then
6             execute $tx$
7         remove_descendants($tx$)
8     remove $B_{1st}$ from $S_{pblock}$
```

5.4 Correctness of Slow-Path-Free CTV

The slow path is a countermeasure against faulty peers and unreliable networks. In the fault-free scenario, even without the slow path, CTV is still able to complete all transactions. We formalize this property of CTV as the theorem below, which facilitates the proof.

Theorem: Given a certain number of ordered blocks, if (1) all peers in the shard are honest, (2) messages are eventually delivered, and (3) the slow path is disabled, CTV can still complete all transactions, i.e., no transaction is left forever in a local dependency graph.

Proof. We prove this theorem by contradiction. Suppose some transactions remain forever in the local dependency graphs at some peers. We denote peer i's local dependency graph at time $t = \infty$ by $G_i = (V_i, E_i)$, where V_i is the set of pending transactions and E_i is the set of directed edges. By combining all the local vertex sets, one can build the global vertex set $V = \cup_{i=1}^{3f+1} V_i$. Undoubtedly, V comprises all transactions that linger in at least one local graph. The ordered blocks establish a total order of the transactions in V: $x < y$ if transaction x is in a block ordered before the block that includes y, or x and y are in the same block and x is ordered before y. Consequently, there must be a minimal element x_{min} in V such that $\forall_{x \in V}(x_{min} \leq x)$. We will first prove that peers in $VC(x_{min})$ cannot have x_{min} in their local dependency graphs and then prove that the existence of x_{min} leads to a contradiction.

Back when peer i where $i \in VC(x_{min})$ checked x_{min} for PPTs, its local dependency graph consisted of only transactions ordered before x_{min}. We denote the graph at that moment as $G_i' = (V_i', E_i')$. Then there exists $\forall_{x' \in V_i'}(x' < x_{min})$. If x_{min} did not depend on any transactions in V_i', it could not be added to the dependency graph and thus cannot exist in V_i. On the other hand, if x_{min} depended on some transaction(s) in V_i', it would be added to the graph but removed later. This is because from $\forall_{x \in V}(x_{min} \leq x)$ and $V_i \subseteq V$, we can derive that $\forall_{x \in V_i}(x_{min} \leq x)$, so there must be $V_i' \cap V_i = \varnothing$. In other words, V_i' and V_i share no common transactions because the former comprises transactions ordered before x_{min} but the latter comprises transactions ordered after x_{min} and possibly x_{min} itself. The empty intersection means V_i includes no parent transactions of x_{min}, so x_{min} must have been removed together with its last parent transaction in the graph. Consequently, in either case, there exists $\forall_{i \in VC(x_{min})}(x_{min} \notin V_i)$.

From $x_{min} \in V$ and $\forall_{i \in VC(x_{min})}(x_{min} \notin V_i)$, we can derive that $\exists_{i \notin VC(x_{min})}(x_{min} \in V_i)$. We denote one of such peers, whose local dependency graph includes x_{min}, by k. According to hypothesis (1), the $f+1$ peers that are members of $VC(x_{min})$ must have produced consistent verification results for x_{min} and sent the results to k. Because of hypothesis (2), k must have received the results. Since x_{min} remains in V_k, the results must have confirmed that x_{min} is valid, and x_{min} has fallen into the *pending-d* state as shown in the fast path of the second life cycle. The *pending-d* state means a parent transaction of x_{min} must exist in V_k.

However, as a valid transaction, x_{min} only consumes UTXO(s) produced by transactions ordered before it. Since $\forall_{x \in V}(x_{min} \leq x)$ and $V_k \subseteq V$, we have $\forall_{x \in V_k}(x_{min} \leq x)$. In other words, no transaction in V_k is ordered before x_{min}, so V_k cannot include a parent transaction of x_{min}. Previously we have deduced that a parent transaction of x_{min} exists in V_k, so V_k both includes and does not include a parent transaction of x_{min}, a contradiction. □

5.5 Evaluation

We implemented CTV as well as an OmniLedger-like sharding protocol based on Bitcoin Core [50]. Inspired by [100] and [70], we also implemented chain replication of blocks to overcome the bandwidth bottleneck of PBFT leaders. Since CTV is targeted at improving the performance of individual shards, we evaluated it with a single shard. Experiments are done on a local cluster. Peers run on the machines with dual Xeon E5-2620 at 2.1 GHz (12 cores) and 64GB RAM, and one client runs on a machine with dual Xeon E5-2630 at 2.6 GHz (12 cores, 2 hyperthreads per core) and 256GB RAM. The client reads historical transactions from the Bitcoin blockchain and sends them to the leader node at a rate of 10k tps. The transactions in 500 recent Bitcoin blocks (block height 601000 to 601499)

Table 5.1: The default setting

Round-trip delay between peers	40 ms
Network bandwidth	200 Mbps
Block size	500 txns
Number of peers	16
Slow path triggering timeout (T_{slow})	100 s

are replayed, totaling 1,120,294 transactions. Each experiment is repeated five times. To emulate a geo-distributed environment, network delays are injected between peers using Linux `NetEm`. If unspecified, default parameter values listed in Table 5.1 are used. The impacts of these parameters will be evaluated in Section 5.5.2.

5.5.1 Importance of Transaction Dependency Awareness

Sharded verification from the Red Belly Blockchain [28] also lets peers share transaction verification results to reduce the per-peer computational workload. As their experiments are designed such that clients never send transactions consuming UTXOs of pending transactions, sharded verification does not incorporate dependency tracking. However, it is not uncommon for transactions to consume UTXOs of pending transactions. For example, 299 out of the 2399 transactions in Bitcoin block 601500 depend on some preceding transaction(s) in the same block. Therefore, we do not adjust the transaction-sending order in our experiments.

Figure 5.6 compares CTV with sharded verification and ITV. The throughput of sharded verification drops to 0 tps quickly after the client finishes sending transactions at 110s. As a result, it completes merely 34% of the transactions. By not accounting for transaction dependencies, peers verify some child transactions before parent transactions and thus mistakenly determined that the child transactions are invalid. Then, all the descendant transactions of such mistakenly invalidated transactions become invalid due to missing input UTXOs. Some readers might think that sharded verification should be able to complete more transactions because most Bitcoin users seem to transact infrequently. However, we find that 84% of transactions in blocks of height [601000, 601500) consume UTXO(s) produced in the same block height range. In other words, with this workload, shard verification only guarantees to complete 16% of the transactions; the other 84% are vulnerable to dependency violation. By contrast, CTV and ITV complete all the transactions, with CTV 2.2x faster. The red line representing CTV in Figure 5.6 stops at around 210s because CTV completes processing all the transactions.

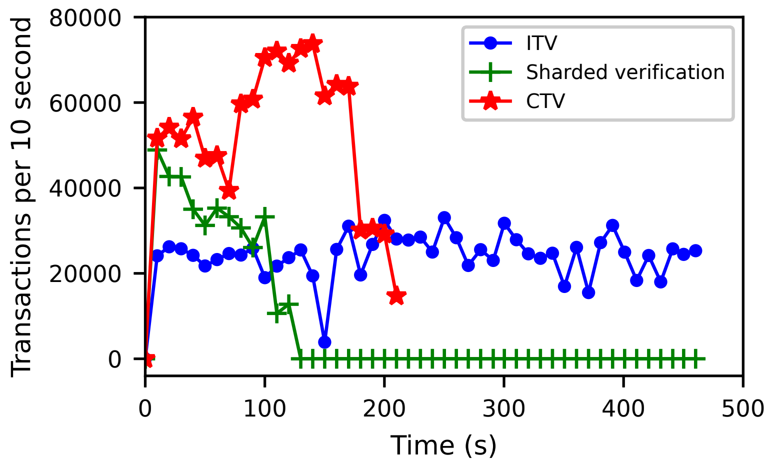


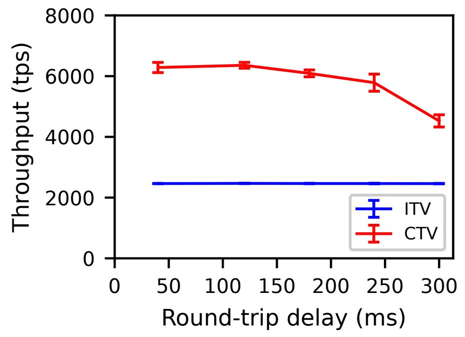
Figure 5.6: Comparison of throughput between ITV, sharded verification, and CTV

The plummeting throughput of the ITV at 150s results from transactions with unusual input UTXO counts. These transactions map to Bitcoin blocks 601145 to 601147. For example, starting from the 102nd transaction in block 601147, each of the 34 subsequent transactions spends 200 UTXOs [16]. These transactions also cause the low throughput of CTV at around the 70s.

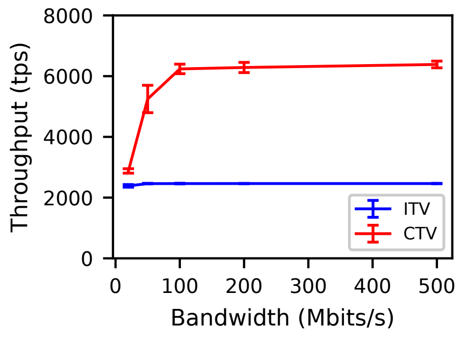
5.5.2 Fault-Free Performance

As we saw in the last section, sharded verification cannot complete all the transactions due to transaction dependency violation, so it is meaningless to measure its performance in the absence of correctness. Thus in this section, we only compare the performance of our approach with that of independent transaction verification.

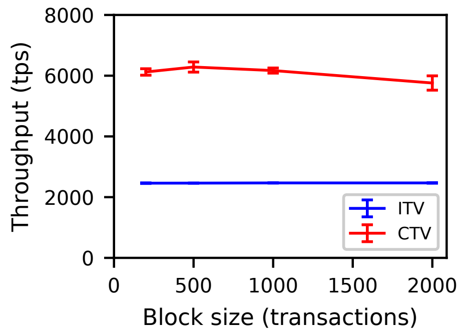
With the default setting, CTV achieves 2.6x higher throughput than ITV. This throughput gain is approaching the ideal value for a 16-node shard, which is $\frac{3f+1}{f+1}|_{f=5} = 2.7$x. To evaluate the performance of CTV under various configurations, we varied the parameters in Table 5.1 and summarized the results in Figure 5.7. Since we sacrifice some level of network usage in exchange for a lighter computational load, we first evaluate how network condition affects CTV. Equal network delays are injected between each pair of peers. Figure 5.7a demonstrates that long network delays negatively influence the performance of



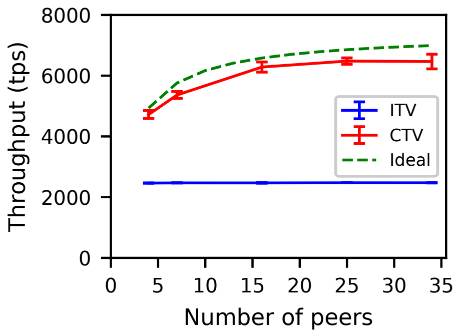
(a) Throughput v.s. network delay



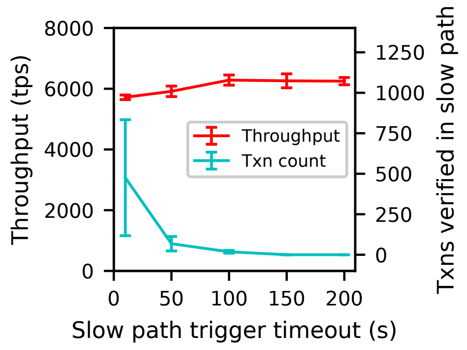
(b) Throughput v.s. bandwidth



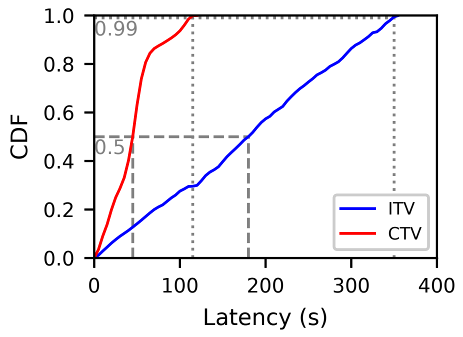
(c) Throughput v.s. block size



(d) Throughput v.s. shard size



(e) Throughput v.s. T_{slow}



(f) Execution Latency

Figure 5.7: Performance under various configurations

CTV. This is because each *pending-p* transaction remains in the dependency graph for a longer time and tends to hamper more transactions. However, even under the extremely long delay of 300ms (higher than the latency between any two of the 14 AWS regions measured by [28] except for São Paulo), CTV still yields 84% throughput improvement. In addition, Figure 5.7b shows that CTV can reach its full capacity when peers are connected by a network with minimally 100 Mbps bandwidth. When bandwidth is lower than 100 Mbps, blocks are not disseminated as fast as peers can process them. At 20 Mbps, even the performance of ITV is slightly attenuated.

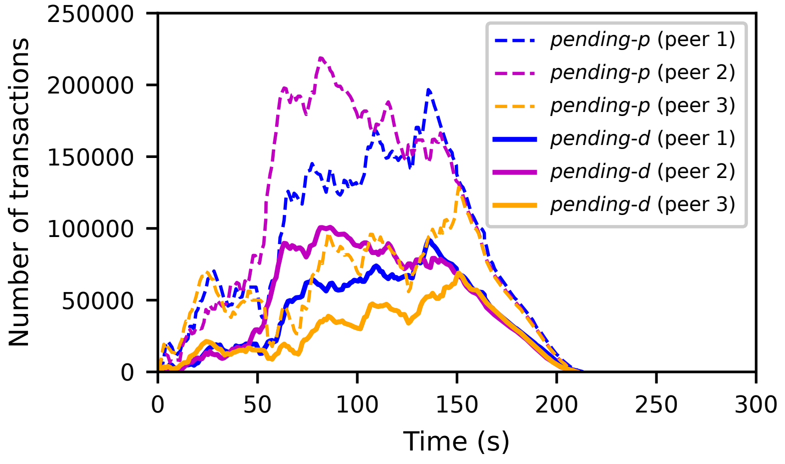
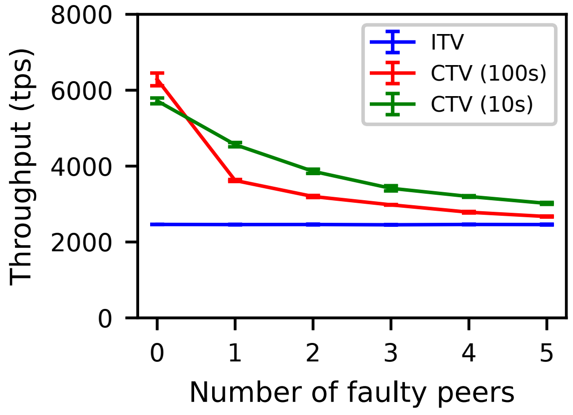


Figure 5.8: Dependency graph size

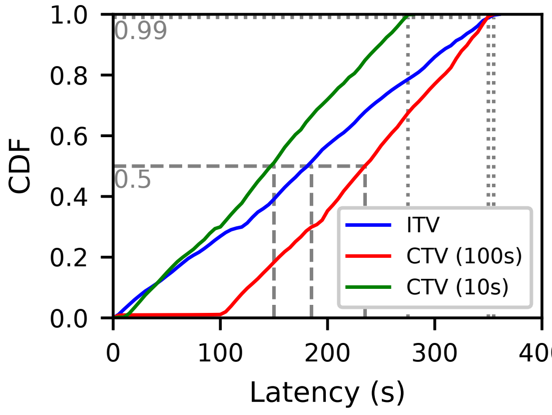
Large block sizes diminish the performance of CTV as illustrated in Figure 5.7c. This arises out of the design decision that CTV only checks newly received verification results in between the processing of two blocks (see Algorithm 1). Small block sizes allow peers to perform the checking frequently and evict transactions from the dependency graph timely, but excessively small block sizes incur high consensus overhead and should be avoided since one consensus instance is invoked per block. Next, we varied the number of peers in the shard. From Figure 5.7d, one can see that CTV consistently attain over 92% of the ideal throughput values, which equal the throughput of ITV multiplied by $\frac{3f+1}{f+1}$. Lastly, we tried different T_{slow} values. With low T_{slow}, peers are "impatient" about waiting for verification results from others, so they verify more transactions in the slow path, shown by the lower curve in Figure 5.7e. Unsurprisingly, the extra verification work impairs the performance. On the other hand, when $T_{slow} \geq 150$s, no transaction takes the slow path, which reinforces our correctness analysis in section 5.4. In addition to throughput improvement, CTV also reduces execution latency by 2/3 due to the overall per-peer workload drop. The 100-second-scale latency comes from the 10k-tps sending rate, which stresses the system since it is higher than the transaction processing speeds and thus causes a long server-side transaction queuing delay. For the same reason, the default T_{slow} value is set to 100s instead of some lower value.

Figure 5.8 shows how the dependency graph sizes change over time at three sampled peers. Although the number of pending transactions varies from peer to peer, like in the concrete example, *pending-p* transactions are significantly more than *pending-d* transactions across all peers, which suggests that avoiding establishing edges for *pending-p* transactions effectively reduces the cost of maintaining the graph. Another observation is that there are no forever-pending transactions, which agrees with the correctness analysis.

5.5.3 Performance Under Faults



(a) Throughput under faults



(b) Latency under five faulty peers

Figure 5.9: Performance under faults. CTV(t) represents CTV with $T_{slow} = t$.

In our experiments, faulty peers do not process transactions but still participate in block dissemination. Even when 5 out of 16 peers are faulty, CTV still outperforms ITV as shown in Figure 5.9a. In addition, lower T_{slow} values help improve the performance of CTV under faults, because peers stop waiting for faulty peers to share verification results earlier. When the number of fault peers equals f, the probability of forming a fault-free

VC is $\binom{2f+1}{f+1}/\binom{3f+1}{f+1}$, which equals only 0.058 in a 16-peer shard. Therefore, in Figure 5.9b, few transactions have less-than-100s latency when $T_{slow} = 100s$, although this value hardly affects the latency in the fault-free scenario due to the rare usage of the slow path. However, the latency can be significantly improved by using a short timeout such as 10s. Because peers also share verification results obtained in the slow path, initially timing-out peers essentially substitute for faulty peers to fulfill verification result sharing. Thus, each peer still verifies fewer transactions than under ITV, so the two CTV curves in Figure 5.9b rise faster than the ITV curve.

5.6 Chapter Summary

As various works optimize consensus and block dissemination, computationally intensive transaction verification has become the newly exposed bottleneck of BFT-based blockchains, including sharded ones. CTV helps accelerate transaction processing by allowing shard members to concurrently verify transactions and exchange the results with each other. Dependency awareness ensures that CTV preserves the state machine replication model despite the fact that peers verify transactions in different orders. Our evaluation results show that CTV can produce 2.6x throughput improvement without compromising fault tolerance or degrading the performance under faults.

CTV can be adapted to account-balance blockchains by changing the parent detection mechanism to an approach based on accessed account analysis. CTV also applies to BFT-based unsharded blockchains with a moderate number of peers. Permissioned blockchains are likely to fall within this category.

Chapter 6

Smart Transaction Placement

Transaction placement in a UTXO-based sharded blockchain means to determine the output shards of transactions. Cross-shard transaction processing must involve communication between the participant shards to ensure consistent states across shards, hence a high processing cost. In this chapter, I will first describe two transaction characteristics to show that reducing cross-shard transactions through transaction placement is feasible and can benefit the performance of sharded blockchains. Then, I will present Rooted Graph Placement, which leverages transaction dependencies to greatly reduce cross-shard transactions, followed by two techniques for speeding up the processing of cross-shard transactions and their dependent transactions.

6.1 Transaction Characteristics

Understanding transaction characteristics allows us to determine whether carefully placing transactions to shards can achieve substantial performance improvement. We are particularly interested in two characteristics: 1) how many transactions can be easily turned into single-shard transactions, and 2) how expensive a cross-shard transaction can be when compared with a single-shard transaction. A transaction placement algorithm can only improve the system performance if cross-shard transactions cost a lot more than their single-shard counterparts and can be greatly reduced by the placement algorithm.

6.1.1 Transaction Dependencies

If transaction $tx2$ spends the UTXO(s) produced by transaction $tx1$, then $tx1$ is referred to as the *parent transaction* and $tx2$ as *the child transaction*. Given a transaction, the number of its parent transactions directly affects its probability of being single-shard. For example, a transaction with only one parent can be a single-shard transaction if it is placed in the same shard as its parent. In contrast, a transaction with 100 parents is very unlikely to be single-shard since that requires all of the 100 parents have been placed in the same shard. Thus, we analyze parent transaction counts for over 1 million transactions in the Bitcoin blocks of height 601000 to 601499 (coinbase transactions are excluded since they have no parents and are guaranteed to be single-shard). Because blocks are usually referred to using block heights, we refer to the block of height i as block i in the rest of this chapter.

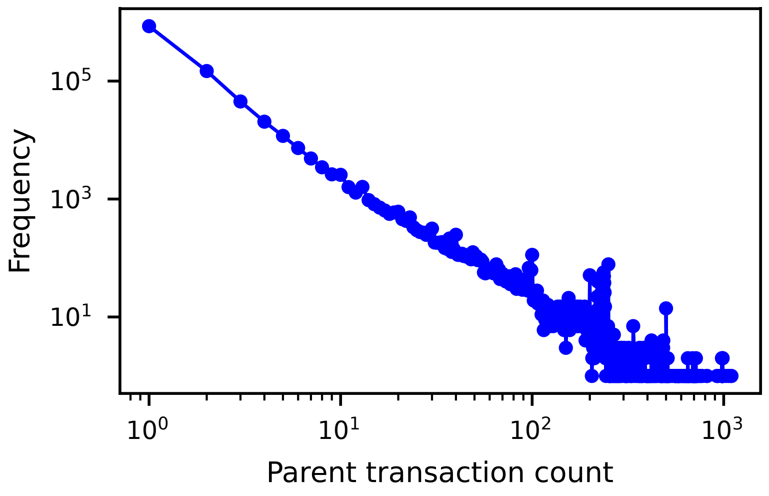


Figure 6.1: Distribution of parent transaction counts

Fig. 6.1 shows that parent transaction counts conform to a power-law distribution, which means that transactions with a few parents occur more frequently than transactions with many parents. We are particularly interested in transactions with one parent because a placement algorithm can easily make such transactions single-shard. By analyzing Bitcoin block 0 to block 680k (mined in April 2021 [12]), we found that one-parent transactions account for a significant proportion throughout the history of Bitcoin and remain approximately 75% since block 420k as shown in Fig. 6.2. The percentage changes a lot for the first 100k blocks because they include only a few transactions at the start of Bitcoin, as shown by the blue bars in Fig 6.2.

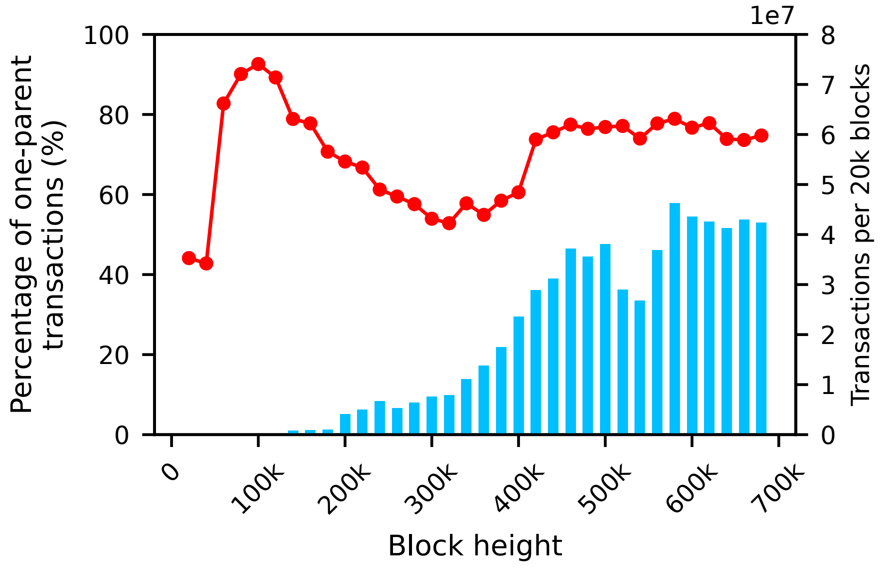
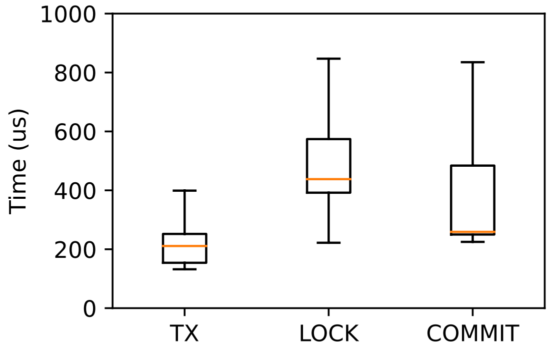


Figure 6.2: The percentage of transactions with one parent

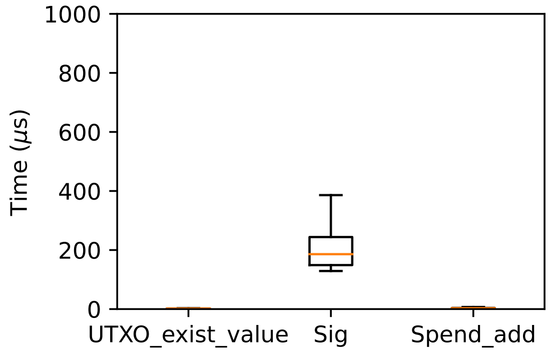
6.1.2 Cost of Cross-Shard Transactions

In this section, we compare the execution time of single-shard transactions and cross-shard transactions by quantitative measurements. An OmniLedger-like sharding protocol is implemented based on Bitcoin Core [50], and Bitcoin transactions are replayed in a 2-shard environment. More details about the implementation and the testbed will be given in Section 6.4. For a single-shard transaction, the client sends a `TX` request containing the transaction directly to the output shard. For a cross-shard transaction, the client sends a `LOCK` request to each input shard and a `COMMIT` request to the output shard, provided all input shards reply with positive locking results (illustrated in Fig. 2.6). The measurement results show that the median processing time for a `TX` request, a `LOCK` request, and a `COMMIT` request are 211 μs, 438 μs, and 259 μs, respectively.

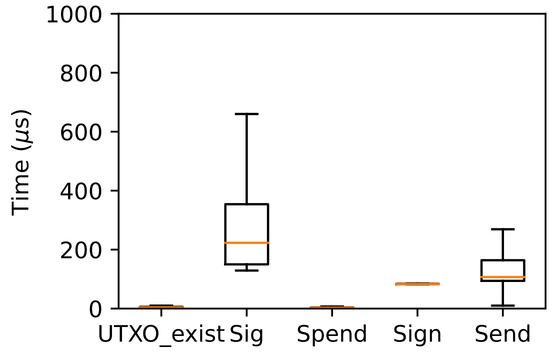
To understand why `LOCK` and `COMMIT` requests take a longer time to process than `TX` requests, we further measure the processing time of each step in request processing. In a UTXO-based blockchain, a node checks for three conditions when verifying a transaction:



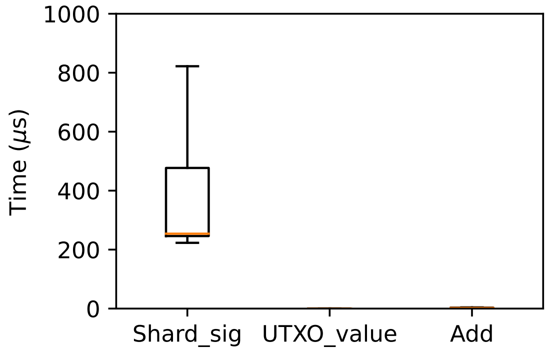
(a) TX, LOCK, and COMMIT



(b) Breakdown of TX



(c) Breakdown of LOCK



(d) Breakdown of COMMIT

Figure 6.3: Request processing time (2 shards)

1) the input UTXOs exist and are unspent, 2) the total input value is not less than the total output value, and 3) the transaction includes the correct signatures from input UTXO owners. Provided the transaction meets all three conditions, the node will execute it by removing the input UTXOs and adding the output UTXOs to the system state. In Fig. 6.3b, the *UTXO_exist_value* label corresponds to the first two transaction verification steps, the *Sig* label to the third verification step, and the *Spend_add* label to the system state update. Fig. 6.3b shows that checking signatures of input UTXOs dominates the processing time of a single-shard transaction. In Fig. 6.3c, the *UTXO_exist* label corresponds to the first transaction verification step. Note that `LOCK` request processing does not include value checking because an input shard is oblivious to input UTXO values of other input

shards. Thus, value checking is done by the output shard during `COMMIT` request processing. Similarly, system state update is separated into two parts: removing the input UTXOs in `LOCK` request processing, and adding the output UTXOs in `COMMIT` request processing. Compared with `TX` request processing, `LOCK` request processing includes two additional steps: signing the lock result (labeled as *Sign*) and sending the result to the client (labeled as *Send*). These two steps are comparable in processing time to input UTXO signature checking. Lastly, `COMMIT` request processing is dominated by verifying the signatures of lock results, which are produced by input shards. The step is labeled as *Shard_sig* in Fig. 6.3d.

As a result, if a one-parent transaction becomes cross-shard as a result of being placed to a different shard than its parent, its processing time would be more than tripled due to the communication and input shard authentication overhead.

6.2 Rooted Graph Placement (RGP)

In this section, we describe our transaction placement algorithm—Rooted Graph Placement (RGP), which reduces cross-shard transactions in UTXO-based blockchains. We observed that, compared with unrelated transactions, transactions with (transitive) dependencies in between are more likely to have their output UTXOs spent together. Figure 6.4 illustrates the intuition behind this observation. Suppose Alice controls two Bitcoin addresses [9] and decides to dedicate $Alice\_addr_2$ for burger buying. In tx_1, Alice transfers 1 BTC from $Alice\_addr_1$ to $Alice\_addr_2$. Then in tx_2, Alice pays for one burger, which costs 0.4 BTC, with the first output UTXO of tx_1 (denoted by $\langle tx_1, 0\rangle$). In tx_3, she buys the second burger. After that, the balance in $Alice\_addr_2$ is lower than the burger price, so Alice combines her two addresses to pay for a burger in tx_4.

This observation suggests that considering non-parent ancestor transactions can help with reducing future cross-shard transactions. In the above example, when tx_3 is to be placed, transactions tx_1 and tx_2 are historical transactions, and tx_4 does not exist yet but is a possibility in the future. If we only consider the parent transaction tx_2, we would place tx_3 to the same shard as tx_2 so that tx_3 is a single-shard transaction. However, tx_4 will be single-shard if we place tx_3 to the same shard as tx_1, a non-parent ancestor of tx_3. Therefore, we design RGP to place a transaction to the shard with most of the ancestor transactions, including non-parent ones.

Besides cross-shard transaction reduction, RGP also considers load balancing. Because sharding protocols usually partition peers based on unbiased random numbers and reconfigure shard membership periodically for security purposes[62][107][30], this work assumes

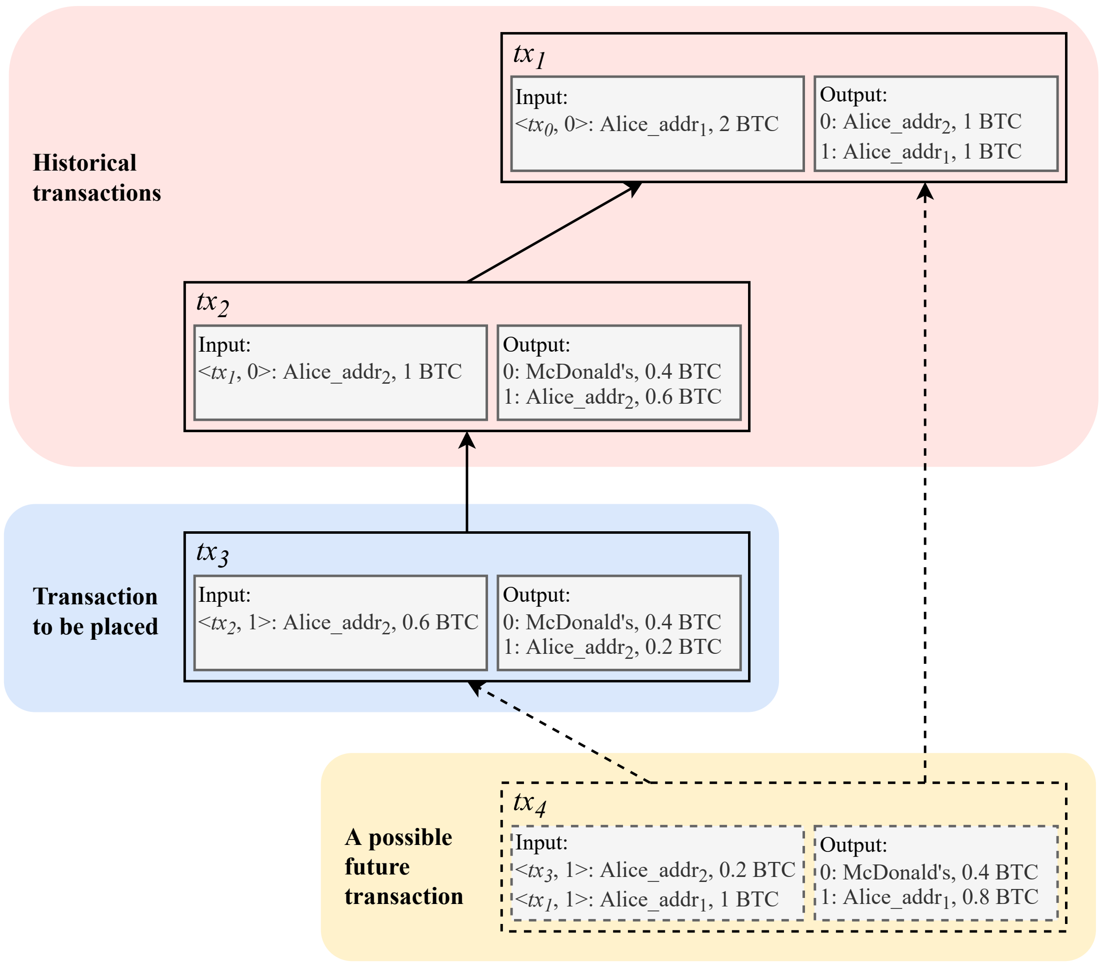

Historical transactions
tx_1
Input:
<tx_0, 0>: Alice_addr$_1$, 2 BTC
Output:
0: Alice_addr$_2$, 1 BTC
1: Alice_addr$_1$, 1 BTC
tx_2
Input:
<tx_1, 0>: Alice_addr$_2$, 1 BTC
Output:
0: McDonald's, 0.4 BTC
1: Alice_addr$_2$, 0.6 BTC
Transaction to be placed
tx_3
Input:
<tx_2, 1>: Alice_addr$_2$, 0.6 BTC
Output:
0: McDonald's, 0.4 BTC
1: Alice_addr$_2$, 0.2 BTC
A possible future transaction
tx_4
Input:
<tx_3, 1>: Alice_addr$_2$, 0.2 BTC
<tx_1, 1>: Alice_addr$_1$, 1 BTC
Output:
0: McDonald's, 0.4 BTC
1: Alice_addr$_1$, 0.8 BTC

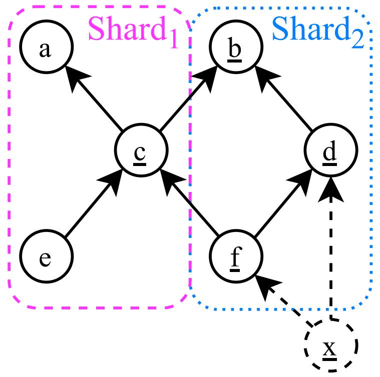
Figure 6.4: An output UTXO of a dependent transaction and an output UTXO of an ancestor transaction are consumed together.

that computational resources are evenly distributed among shards. Thus, RGP attempts to assign an equal number of transactions to all shards.

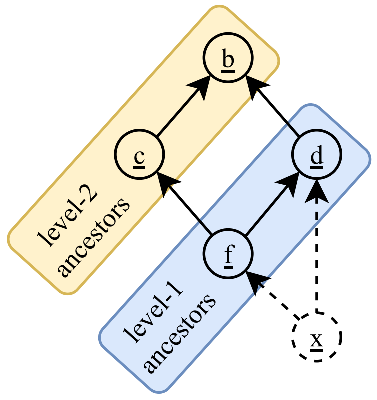
6.2.1 Cross-Shard Transaction Reduction

As RGP aims at placing a transaction to the shard with the most ancestor transactions, it models these transactions and dependencies between them using a rooted directed acyclic graph (DAG). Given a new transaction, RGP builds a graph $G = (V, E)$ rooted at the new transaction. A transaction $u \in V$ only if u is an ancestor transaction of the new transaction (including the new transaction itself). A directed edge $(v, u) \in E$ if and only if $u \in V$, $v \in V$, and v consumes the output UTXO(s) of transaction u. RGP builds the graph starting from the root because child transactions carry information about parent transaction IDs. Finding all ancestor transactions is expensive, so RGP only considers ancestor transactions within a certain distance from the root. We refer to RGP that traces back k levels of ancestor transactions as RGPk. Figure 6.5a illustrates an example of RGP2, where transactions that are part of the rooted graph are underlined. Transaction a is not part of the rooted graph because it is 3 hops away from the root x, while RGP2 only considers ancestors within two hops. Therefore, the rooted graph is essentially a subgraph of the global transaction dependency graph. An ancestor transaction is called a *level-j ancestor* if the shortest path between the root vertex and the ancestor vertex consists of j edges, as shown in Figure 6.5b.

With the rooted graph, RGP counts the number of ancestor transactions in each shard and calculates *cost scores* based on the counting results. A shard's cost score reflects the cost-effectiveness of placing the new transaction in the shard. Generally, RGP attempts to place a new transaction to the shard that has processed most of its ancestors, but we have two special considerations.



(a) Ancestors in two shards



(b) Ancestor levels

Figure 6.5: An example of RGP2. Underlined transactions are vertices of the rooted graph.

First, we distinguish between *totally spent ancestors* and *partially spent ancestors*. A partially spent ancestor has output UTXO(s) that remain unspent after the new transaction is executed, whereas a totally spent ancestor has all its output UTXOs spent. Figure 6.6

demonstrates why totally spent ancestors should be given less weight than partially spent ancestors. Suppose transaction x is a new transaction to be placed, and its two ancestors (i.e., transaction d and f) are in different shards. Transaction d is totally spent since $UTXO_2$ has been spent by transaction f and $UTXO_3$ will be spent by x. On the other hand, transaction f is partially spent since no transaction consumes $UTXO_5$. Thus a future transaction y may consume the output UTXOs of both transaction f and transaction x. To prevent such future transactions from modifying two shards, transaction x should be placed in the same shard as f. Therefore, we use a coefficient $\alpha \in (0, 1)$ to give totally spent ancestors less weight than partially spent ancestors.

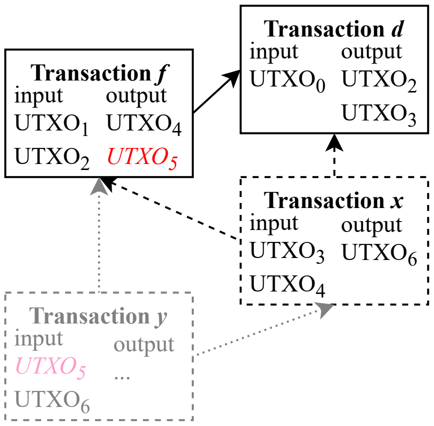


Figure 6.6: Transaction f is a partially spent ancestor of transaction x, and transaction d is a totally spent ancestor of transaction x.

Second, RGP takes level sizes (i.e., the number of ancestors in each level) into consideration so that cost scores are not biased by the largest level. For example, a new transaction may have one level-1 ancestor (in $shard_1$) and four level-2 ancestors (one in $shard_1$ and three in $shard_2$). Without considering level sizes, RGP would tend to assign the new transaction to $shard_2$, since $shard_2$ holds one more ancestor than $shard_1$. In other words, the cost scores would be biased by level-2 ancestors because they outnumber the level-1 ancestor by 4 times. However, the level-1 ancestor is important since the new transaction would be single-shard if placed to the same shard as the level-1 ancestor. Thus, RGP divides ancestor counts by the corresponding level sizes to ensure the equal significance of each level. Incorporating the two special considerations into RGP, we define the cost score as follows:

$$S_{cost}(i) = \sum_{j=1}^{k} \frac{p_{ij} + \alpha t_{ij}}{\sum_{m=1}^{n_s}(p_{mj} + t_{mj})} \tag{6.1}$$

where $S_{cost}(i)$ is the cost score of shard i ($1 \leq i \leq n_s$); n_s is the number of shards; k is the number of ancestor levels in the graph; p_{ij} (or p_{mj}) is the number of partially spent level-j ancestors in shard i (or shard m); t_{ij} (or t_{mj}) is the number of totally spent level-j ancestors in shard i (or shard m); α is the totally spent ancestor weight ($0 < \alpha < 1$). The denominator $\sum_{m=1}^{n_s}(p_{mj} + t_{mj})$ is the sum of level-j ancestors across shards, which represents the size of the j-th level in the rooted graph. $S_{cost}(i)$ is in the range of [0, k]. A high cost score means placing the transaction in the shard is likely to reduce future cross-shard transactions, including the one currently being placed.

6.2.2 Load Balancing

When placing a transaction, RGP also calculates a *load score* for each shard to account for load balancing. A high load score means that the shard is experiencing a relatively light workload, so the transaction will experience a relatively low queuing delay if placed to the shard. We use partition i (denoted by P_i) to refer to the set of transactions that have been placed to shard i. Obviously, small partitions should receive high load scores. Also, we want to limit the maximum partition size difference so that load imbalance is bounded. A piecewise function is designed to satisfy these two requirements:

$$S_{load}(i) = \begin{cases} 0 & \text{if } |P_i| \geq |P_{min}| + \theta \\ 1 - \gamma \frac{|P_i| - |P_{min}|}{|P_{max}| - |P_{min}|} & \text{if } |P_{min}| + \theta > |P_i| > |P_{min}| \\ 1 & \text{if } |P_i| = |P_{min}| \end{cases} \tag{6.2}$$

where $S_{load}(i)$ is the load score of shard i; $|P_i|$ is the size of partition i; $|P_{min}|$ and $|P_{max}|$ are sizes of the smallest partition and largest partition, respectively; $|P_{min}| + \theta$ is the boundary partition size that distinguishes *large partition*s from *medium partition*s (used to limit load imbalance); $\gamma \in (0, 1]$ is a coefficient that determines how heavily a medium partition is penalized. $S_{load}(i)$ gently penalizes medium partitions based on their sizes and aggressively penalizes large partitions. Note that in the second line of Equation 6.2, the denominator $|P_{max}| - |P_{min}|$ is implicitly guaranteed to be greater than zero by the condition $|P_i| > |P_{min}|$. For medium partitions, $S_{load}(i)$ is in range $[1 - \gamma,\ 1)$.

To take into account both cross-shard transaction reduction and load balancing, the final decision should be based on both the cost score and the load score. While adding up the two scores seems to be a reasonable choice, it cannot limit load imbalance because large partitions with non-zero ancestors could have a higher score sum than small partitions without ancestors. Thus, to have bounded load imbalance, we design the overall score as

Algorithm 5: Rooted Graph Placement

```
Input: a new transaction $x$, the number of ancestor levels $k$, the number of shards
       $n_s$, transactions reachable from $x$ within $k$ hops, totally spent ancestor
       weight $\alpha$, partition sizes $|P_1|, |P_2|, \ldots, |P_{n_s}|$, medium partition penalty
       coefficient $\gamma$, imbalance upper bound $\theta$
Output: $x$'s output shard ID $s_{out}(x)$
1  if $x$ is a coinbase transaction then
2      $s_{out}(x) = \text{hash}(x) \mod n_s$
3  else
       /* Build the rooted graph */
4      starting from transaction $x$, using BFS to build a rooted graph with $k$ levels of
        ancestors.
5      for $i \in [1, n_s]$ do
           /* Compute the cost score of shard $i$ */
6          $S_{cost}(i) = \sum_{j=1}^{k} \frac{p_{ij}+\alpha t_{ij}}{\sum_{m=1}^{n_s}(p_{mj}+t_{mj})}$
           /* Compute the load score of shard $i$ */
7          if $|P_i| \geq |P_{min}| + \theta$ then
8              $S_{load}(i) = 0$
9          else if $|P_i| > |P_{min}|$ then
10             $S_{load}(i) = 1 - \gamma \frac{|P_i|-|P_{min}|}{|P_{max}|-|P_{min}|}$
11         else
12             $S_{load}(i) = 1$
           /* Compute the overall score of shard $i$ */
13         $S_i = S_{cost}(i) \cdot S_{load}(i)$
       /* Place $x$ into the shard with the highest overall score */
14     if $max(S_i) > 0$ then
15         $s_{out}(x) = \text{argmax}_i\ (S_i)$
16     else
17         $s_{out}(x) =$ the shard ID of $P_{min}$
```

the multiplication of the two scores:

$$S_i = S_{cost}(i) \cdot S_{load}(i) \tag{6.3}$$

where S_i is the overall score of shard i. As $S_{cost}(i)$ is in range [0, k], and $S_{load}(i)$ is in range [0, 1], S_i must be in range [0, k]. The overall scores of large partitions are always zero because of Equation 6.2. If $S_i = 0$ for all shards, which occurs when ancestors only exist in large partitions, RGP places the new transaction in the smallest partition. In this way, RGP never places transactions to large partitions, so the maximum partition size difference is bounded by θ. Algorithm 5 shows the complete RGP algorithm.

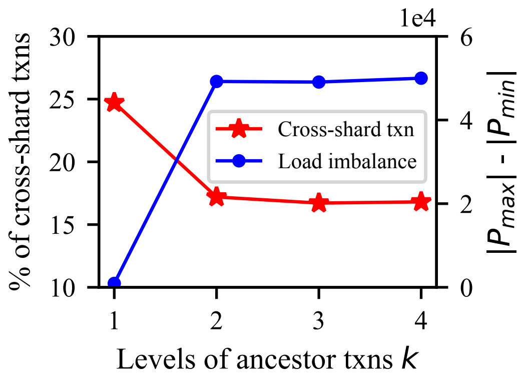
6.2.3 Impacts of Parameters

In this section, we demonstrate how the four parameters of RGP affect its transaction partitioning quality (i.e., the cross-shard transaction number and load balancing) and give the recommended parameter values. Generally, due to the intrinsic tradeoff between cross-shard transaction reduction and load balancing, varying a parameter usually improves one metric but worsens the other. In this section, recommended parameter values are derived using the first 200k Bitcoin blocks. However, we will see that these values are quite robust and transferable to different workloads in Section 6.2.4.

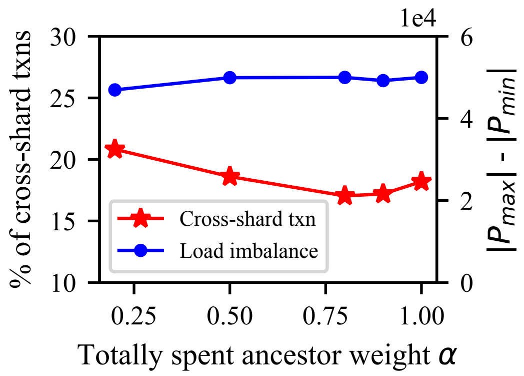
Figure 6.7a demonstrates that RGP1 (i.e., RGP that traces back one level of ancestor transactions) can lower the cross-shard transaction percentage to 25%, in contrast to 93% with hashing placement. RGP2 further reduces cross-shard transactions to 17%. The phenomenon that RGP2 produces fewer cross-shard transactions than RGP1 agrees with the intuition described at the beginning of Section 6.2. Another way of explaining the phenomenon is that RGP2 feeds the RGP algorithm with more information about the global dependency graph than RGP1. It has also been mentioned in [94] that more information yields better partitioning quality. RGP3 and RGP4 can produce even fewer cross-shard transactions, but the improvement is marginal.

In Figure 6.7b, the lowest cross-shard transaction percentage occurs when $\alpha \in [0.8, 0.9]$. This is because when $\alpha = 1$ (i.e., totally spent ancestors are given the same weight as partially spent ancestors), RGP ignores the fact that a partially spent ancestor and the new transaction may be referenced in the same future transaction. Nevertheless, low α values make RGP undervalue totally spent ancestors and thus increase cross-shard transactions as well.

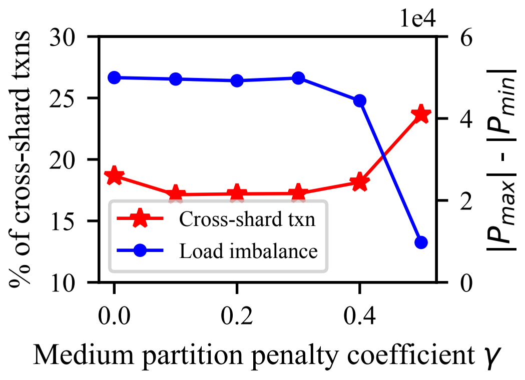
Figure 6.7c illustrates the impact of γ. As expected, with high γ values, medium partitions are penalized heavily, hence better load balancing but more cross-shard transactions. Surprisingly, $\gamma = 0$ also results in a relatively high cross-shard transaction percentage. We believe the reason is that, when $\gamma = 0$, load scores equal either zero or one according to Equation 6.2. As a result, the overall score defined in Equation 6.3 equals either the cost score or zero instead of a comprehensive assessment based on both ancestor transaction distribution and shard loads.



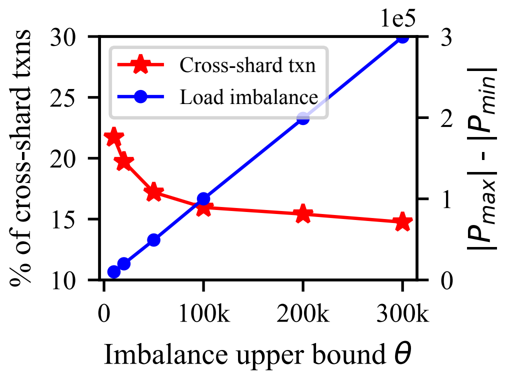
(a) Vary k ($\alpha = 0.9, \gamma = 0.2, \theta = 50\text{k}$)



(b) Vary α ($k = 2, \gamma = 0.2, \theta = 50\text{k}$)



(c) Vary γ ($k = 2, \alpha = 0.9, \theta = 50\text{k}$)



(d) Vary θ ($k = 2, \alpha = 0.9, \gamma = 0.2$)

Figure 6.7: Influence of RGP parameters (16 shards)

Finally, as θ controls the maximum partition size difference, it is natural that load imbalance grows linearly with θ and cross-shard transactions decrease as θ grows. The elbow of the curve in Figure 6.7d suggests that $\theta = 50k$ is a reasonable choice since further increasing θ does not reduce cross-shard transactions much but cause high imbalance. Per the above analysis, we recommend the following parameter values: $k = 2$, $\alpha = 0.9$, $\gamma = 0.2$, $\theta = 50k$. These values are used in the rest of this chapter.

6.2.4 Partitioning Quality Comparison

To show that the recommended parameter values generalize well with other workloads, we employ four Bitcoin transaction sets of similar sizes, as detailed in Table 6.1. The partitioning quality of RGP2 is compared with that of hashing placement and *OptChainV2-T2S*, which is OptChainV2 [80] without transaction latency sampling[1]. OptChainV2 is a state-of-the-art transaction placement algorithm for sharded blockchains, and its details have been given in Section 3.3. OptChainV2-T2S is used instead of OptChainV2 in this section because it can be evaluated analytically without deploying shards, which is also true for hashing placement and RGP2. This is particularly useful when comparing the algorithms under high shard counts, e.g., 128 shards. OptChainV2-T2S produces slightly fewer cross-shard transactions than OptChainV2 because the former misses the sampling feature for fine-grained load balancing. If RGP2 can achieve a similar number of cross-shard transactions as OptChainV2-T2S, it will be at least as effective as OptChainV2 in terms of cross-shard transaction reduction. We will compare RGP2 with sampling-enabled OptChainV2 in Section 6.4.

Table 6.1: Four transaction datasets

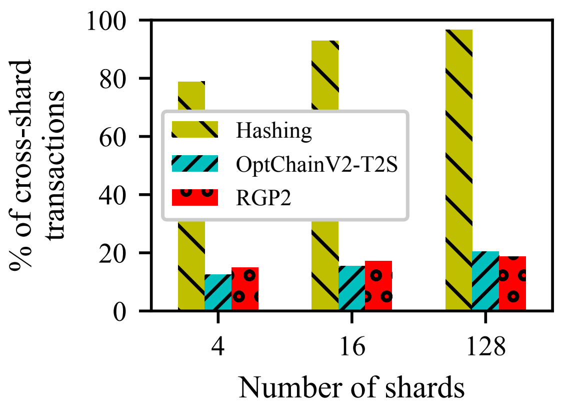
Dataset	Bitcoin block heights	Number of transactions [1]
D_1	[0, 200k)	7,316,308
D_2	[200k, 227k)	7,371,053
D_3	[227k, 252k)	7,316,337
D_4	[252k, 275k)	7,238,332

[1] Coinbase transactions are excluded.

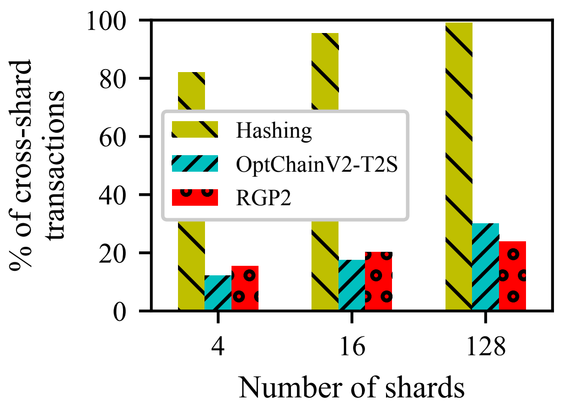
Figure 6.8 compares the cross-shard transaction percentage of the three placement algorithms. For all datasets, regardless of the number of shards, RGP2 and OptChainV2-T2S produce similar numbers of cross-shard transactions, and the number is significantly less than that of hashing placement. These results confirm that despite considering only two levels of ancestors, RGP2 is able to reduce cross-shard transactions as effectively as OptChainV2-T2S.

To learn how shard loads vary over time, we analyze the dynamic shard loads in a 4-shard environment, as illustrated in Figure 6.9. Unsurprisingly, hashing placement balances loads extremely well with every shard constantly receiving about 25% of the transactions.

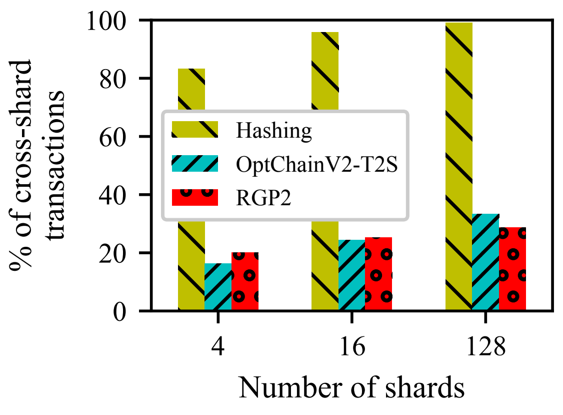
[1]The name *OptChainV2-T2S* comes from the OptChainV2 paper, where the method is referred to as *T2S-based*.



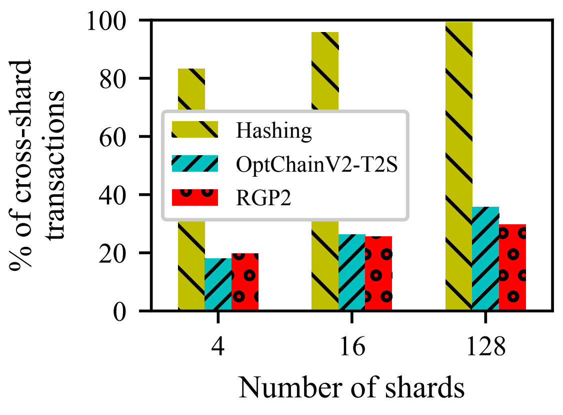
(a) Dataset D_1



(b) Dataset D_2



(c) Dataset D_3

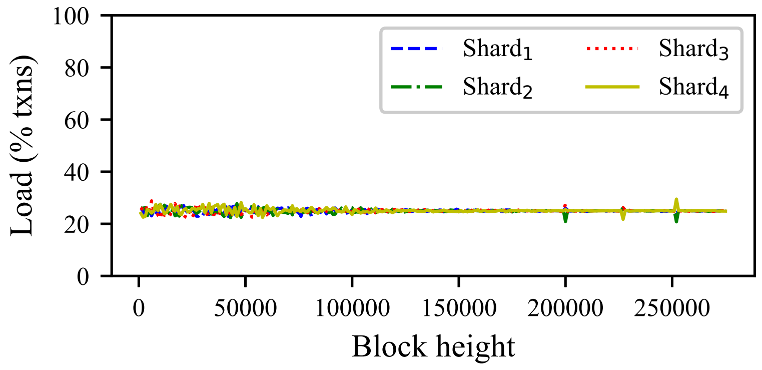


(d) Dataset D_4

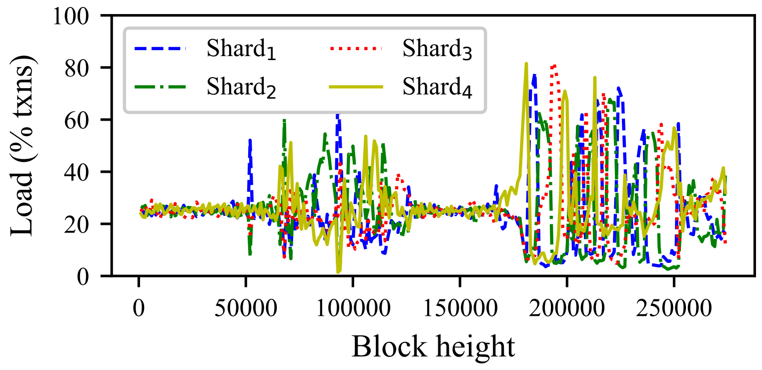
Figure 6.8: Cross-shard transactions

In Figure 6.9a, the three small spikes at block heights 200k, 227k, and 251k correspond to the start heights of datasets $D2 \sim D4$, and the slight fluctuations at low block heights are due to small block sizes.

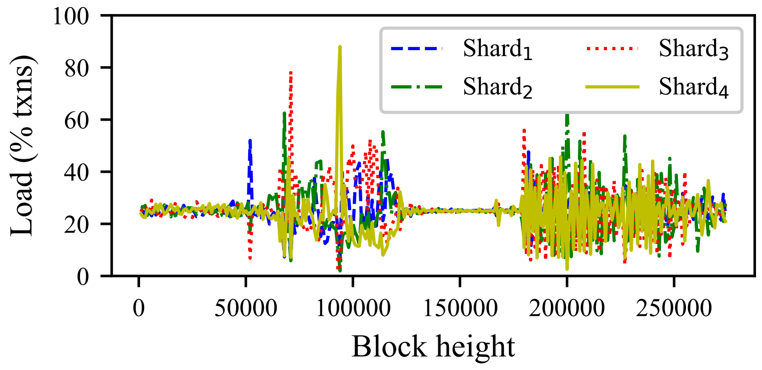
Another observation is that Figure 6.9b and Figure 6.9c exhibit a common pattern: all curves are quite flat in blocks [0, 50k) and [125k, 175k), but fluctuate a lot in the range of [70k, 120k) and [180, 250k). This pattern relates to transactions that consume the output UTXOs of their immediate predecessors on the blockchain. For example, starting from



(a) Hashing placement



(b) OptChainV2-T2S



(c) RGP2

Figure 6.9: Dynamic shard loads (4 shards)

the 326th transaction in Bitcoin block 177253, each of the 267 subsequent transactions spends the UTXOs produced by its immediate predecessor[15]. When faced with such transactions, both OptChainV2-T2S and RGP2 tend to place them to the same shard as their predecessors. Consequently, a sequence of such transactions will cause a shard to temporarily receive more transactions than other shards. However, the load balancing mechanisms prevent a shard from being overloaded for a long time, so shards take turns to receive the most transactions. Table 6.2 shows that transactions depending on their immediate predecessors account for a relatively high percentage whenever the shard load curves fluctuate drastically.

Table 6.2: Bitcoin transactions consuming UTXOs produced by their predecessors

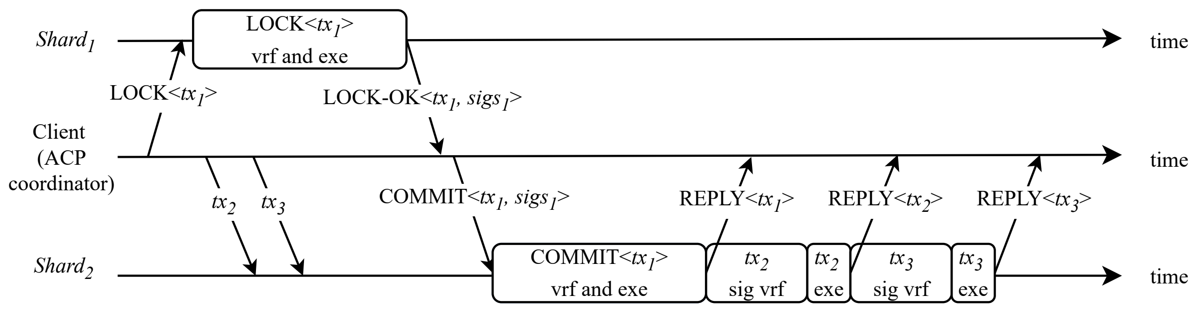
Block height	Transactions depending on their immediate predecessors
[0, 50k)	0.1%
[70k, 120k)	18.6%
[125k, 175k)	6.1%
[180k, 250k)	21.3%
[253k, 275k)	14.6%

6.3 Efficient Cross-Shard Transaction Processing

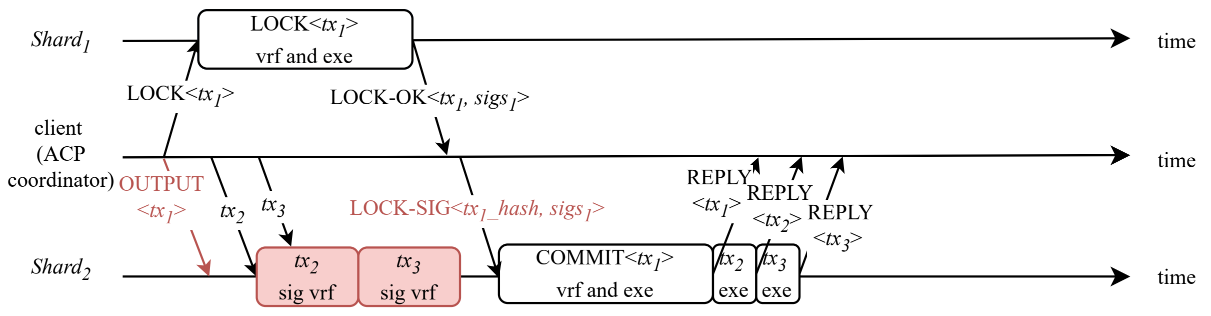
Although RGP reduces the number of cross-shard transactions, it cannot eliminate them. In this section, we propose two techniques to lessen the impact of the remaining cross-shard transactions on system performance. The first technique expedites dependent transaction processing, while the second technique reduces the communication and computation overhead involved in cross-shard transaction processing. Both techniques require modifications to the atomic commit protocol. Since neither technique deals with the intra-shard consensus protocol, we abstract away the consensus process as if client requests are ordered as soon as they reach their destination shards. Our two techniques focus on how transactions are processed (i.e., verified and executed).

6.3.1 Dependent Transaction Pre-verification

Cross-shard transactions usually experience long execution latency due to atomic commit protocols such as Atomix in Figure 2.6. This inevitably delays the processing of their



(a) Plain Atomix



(b) Atomix with dependent transaction pre-verification

Figure 6.10: DPV parallelizes the Atomix lock phase with dependent transaction signature verification.

dependent transactions since transactions must be executed in a dependency-respecting order. Figure 6.10a illustrates the timeline of processing one cross-shard transaction (i.e., tx_1) and two single-shard transactions that depend on it (i.e., tx_2 and tx_3). Suppose the client runs RGP2 and determines that tx_2 and tx_3 should be placed to the output shard of tx_1, which is $Shard_2$. Although the client sends tx_2 and tx_3 soon after sending tx_1, $Shard_2$ delays processing the two dependent transactions until tx_1 is executed in order to respect dependencies. Throughout the process, two steps are notably expensive—the locking phase of the atomic commit protocol and the verification of tx_2 and tx_3—because both steps involve the verification of signatures from input UTXO owners. Nonetheless, the two steps do not have to be carried out in a serialized manner.

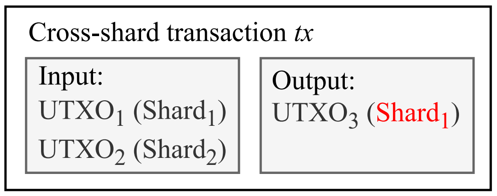
As mentioned in Section 6.1.2, transaction verification comprises input UTXO existence checking, input UTXO value checking, and signature checking. The first item is stateful, whereas the second and the third are stateless. In fact, as long as the input UTXO properties (i.e., owner address, amount, etc.) are available, the second and third conditions can be checked at any time.

To reduce the execution latency of dependent transactions, we propose dependent transaction pre-verification (DPV), which performs dependent transaction signature verification in parallel with the locking phase of the atomic commit protocol. In order to accommodate this idea, the output shard must be informed about the cross-shard transactions' output UTXO properties early. Figure 6.10b illustrates how we achieve this by splitting the `COMMIT` request into two messages, namely `OUTPUT` and `LOCK-SIG`. The `OUTPUT` message carries only the cross-shard transaction and is sent to the output shard as soon as the `LOCK` request is sent to the input shard. $Shard_2$ can start verifying the signatures of tx_2 and tx_3 as soon as it receives the `OUTPUT` message. Meanwhile, the input shard is verifying tx_1's signature(s). On the other hand, the `LOCK-SIG` message carries the input shard signatures as well as a hash of the transaction, which is used to match the `LOCK-SIG` message with the corresponding `OUTPUT` message. The `LOCK-SIG` message serves as proof that the input shard has successfully locked the input UTXOs of tx_1. The `OUTPUT` and `LOCK-SIG` messages are later assembled into one `COMMIT` request so that the original `COMMIT` request processing routine can be reused. DPV also applies to transitive dependencies, e.g., tx_3 may depend on tx_2 instead of tx_1, and cross-shard dependent transactions. In the latter case, the output shard pre-verifies the `LOCK` requests of the cross-shard dependent transactions. DPV pre-verifies dependent transactions in their appearance order on shard ledgers.

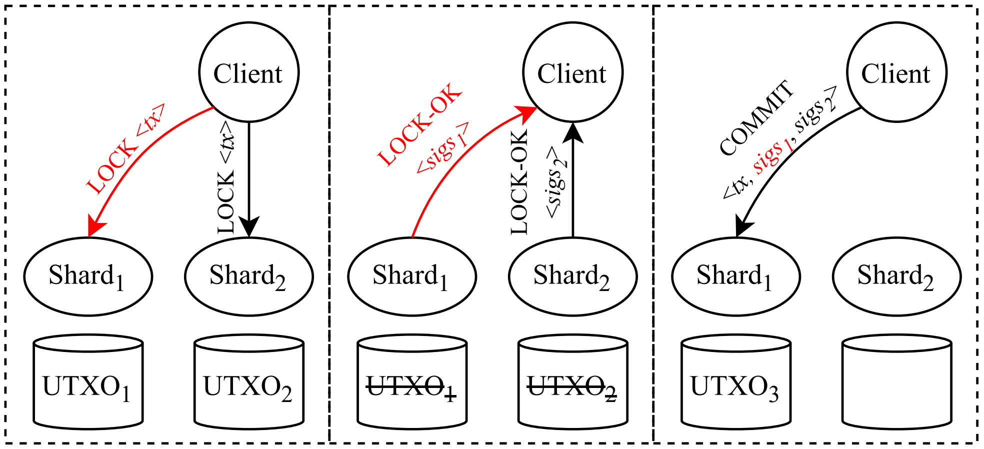
DPV is safe, i.e., invalid transactions will not be mistakenly treated as valid ones. For parent cross-shard transactions, although dishonest clients may send dummy `OUTPUT` messages, the transactions will not be executed without valid signatures from the input shards. In other words, `LOCK-SIG` messages from input shards protect the output shard's system state from being tampered with. For dependent transactions, DPV may verify their signatures but will not execute them until their parent transactions are executed. As a result, a dummy `OUTPUT` message cannot induce the output shard to execute either the cross-shard transaction or the dependent transactions. Nevertheless, the computational work involved in pre-verifying the dependent transactions is wasted, so peers should only pre-verify dependent transactions when CPUs are idle to avoid performance degradation and DoS attacks caused by dummy `OUTPUT` messages.

6.3.2 Atomic Commit Protocol Consolidation

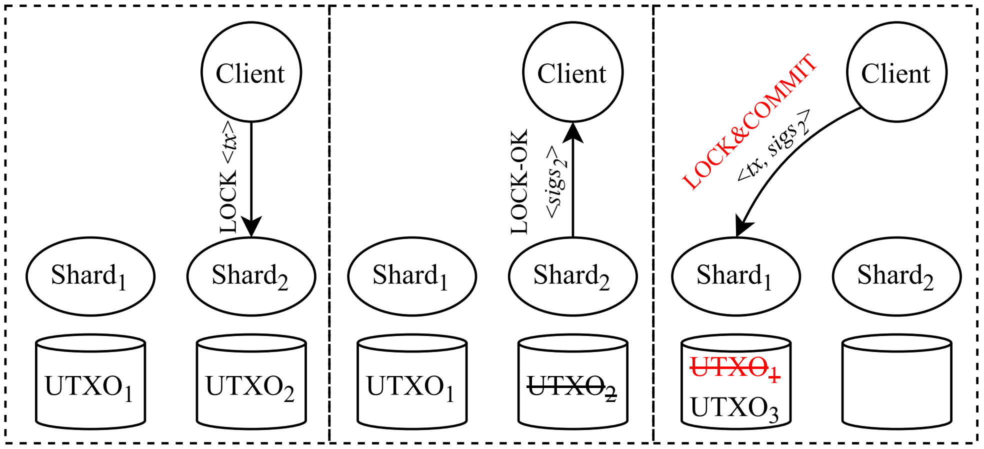
As RGP2 takes transaction dependencies into account, the vast majority of cross-shard transactions are placed in one of their input shards. We refer to such output shards as *input-output shard*s, i.e., $Shard_1$ in Figure 6.11a. This placement pattern is quite different



(a) Input-output shard



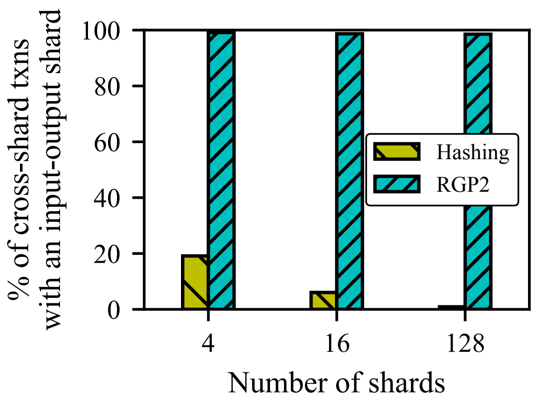
(b) Plain Atomix



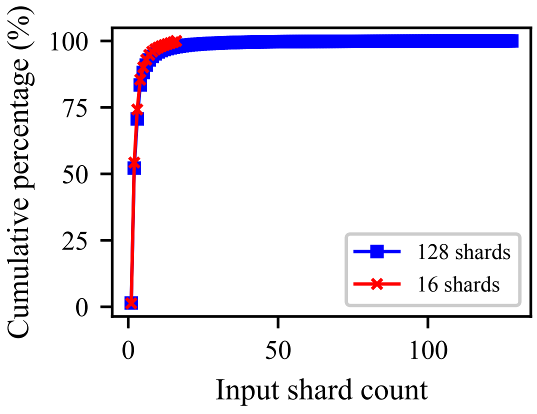
(c) Consolidated Atomix

Figure 6.11: Consolidation of Atomix

than that of hashing placement, which only places a few transactions to their input shards, as illustrated in Figure 6.12a. This difference opens up opportunities for atomic commit protocol optimization. Specifically, the lock and commit requests can be combined into one request for input-output shards. Figure 6.11 takes Atomix as an example in order to illustrate atomic commit protocol consolidation (ACPc). Instead of requesting the two input shards to lock the corresponding UTXOs as in Figure 6.11b, consolidated Atomix merges the `LOCK` request and the `COMMIT` request into a `LOCK&COMMIT` request for the input-output shard (Figure 6.11c). Upon receiving the `LOCK&COMMIT` request, the input-output shard checks for the following conditions: 1) the input UTXO(s) to lock exist, and the transaction is signed properly by the owners, 2) the signatures from other input shards are valid, and 3) total input value is not less than the total output value. If all the three conditions are met, the input-output shard deems the transaction valid and executes the transaction by removing the input UTXO(s) and adding output UTXO(s) to its UTXO database. Otherwise, the input-output shard informs the client about failed locking using a signed `LOCK-NOT-OK` message, which can be used as proof to restore input UTXOs in other input shards.



(a) Cross-shard transactions with input-output shards



(b) CDF of input shard count

Figure 6.12: The vast majority of cross-shard transactions are assigned to one of their input shards under RGP2, and over 80% of cross-shard transactions have 2∼4 input shards (dataset $D1$).

In the successful scenario, consolidated Atomix saves two messages and one shard signature. For two-input-shard transactions, the saving is almost half of the processing cost, which includes five messages and two shard signatures, as shown in Figure 6.11b. Considering that over 50% of cross-shard transactions touch only two input shards, and over 80% of cross-shard transactions span two to four input shards (Figure 6.12b), ACPc should produce obvious performance improvement. One may notice that there is a small portion (around 1.4%) of cross-shard transactions with one input shard. Such transactions are not placed to their input shards probably for balancing load. They will not benefit from ACPc since they lack input-output shards. The input shard count distribution in Figure 6.12b results from the fact that parent transaction numbers follow a power-law distribution [87]. Other transaction datasets in Table 6.1 show the same pattern as dataset $D1$ does in Figure 6.12.

ACPc is compatible with DPV. When the two techniques are deployed together, the `LOCK&COMMIT` message in Figure 6.11c splits into two parts: one message carrying the transaction, and the other carrying the signatures from other input shards as well as the transaction hash. The input-output shard utilizes the former to pre-verify dependent transactions, and the latter to learn that other input shards have successfully executed the transaction.

6.4 Evaluation

We implemented RGP2 as a client-side algorithm, which means a client runs the algorithm to compute the output shard ID before sending its transaction to blockchain peers. This architecture is compatible with OmniLedger, where clients are the atomic commit protocol coordinators, and is also adopted by OptChainV2.

We also implemented OptChainV2 as well as an OmniLedger-like sharding protocol based on Bitcoin Core. In the OptChainV2 paper, the authors modify the OmniLedger protocol to avoid excessive bandwidth usage by letting clients send requests directly to the destination shards instead of gossiping requests. We make the same modification. In addition, each peer maintains a dependency graph of pending transactions to enforce dependency-respecting transaction execution order. We also implemented DPV and ACPc, and measured the system performance when they are deployed together with RGP2.

6.4.1 Testbed

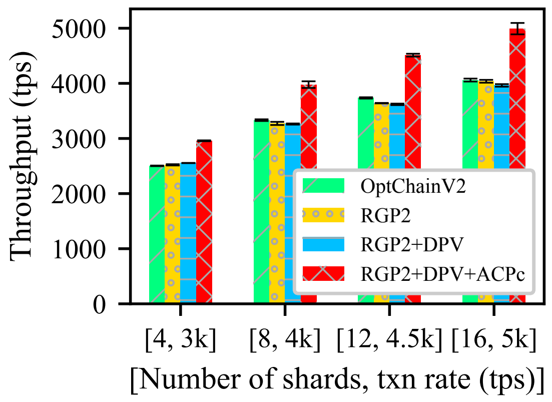
Experiments are done on a local cluster. We run up to 64 peers on 16 machines, each with dual Xeon E5-2620 at 2.1 GHz (12 cores) and 64GB RAM. A shard consists of four peers co-located on the same machine. Each peer is scheduled on two fixed cores for isolation purposes. To emulate a geo-distributed environment, network delay is injected between each pair of peers, and bandwidth limits are imposed on every peer using Linux `NetEm` and traffic control facilities. Properties of links between peers co-located on the same machine are also configurable through loopback interfaces. One client runs on another machine with dual Xeon E5-2630 at 2.6 GHz (12 cores, 2 hyperthreads per core) and 256GB RAM. The client reads historical transactions from the Bitcoin blockchain sequentially and sends them to the peers. To measure the maximum throughput of the system, we must saturate the peers with client requests. Considering the high processing power of the cores, we limit CPU usage to 50%.

To evaluate the performance improvement under different types of workloads, two transaction datasets are employed: transactions in Bitcoin block [0, 136k) and transactions in Bitcoin block [200k, 205k). The first dataset contains simple dependencies, whereas the second dataset is quite the opposite, according to Table 6.2. Both datasets contain approximately 1 million transactions. In 6.4.2 and 6.4.3, we measure the system performance under the two workloads respectively with a 40ms round-trip delay injected between each pair of peers, and a 200-Mbps bandwidth limit imposed on every peer. This mild network configuration allows us to saturate the peers without saturating the network. The impacts of various network configurations will be evaluated in section 6.4.4.

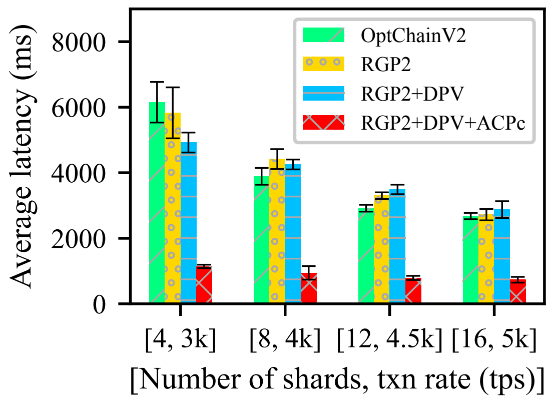
6.4.2 Performance Under Light-Dependency Workload

We compare the performance of four systems. In the first system, the client places transactions to shards according to OptChainV2; in the second system, we replace OptChainV2 with RGP2; in the third system, we add DPV to the second system; in the fourth system, we add ACPc to the third system. In figures presenting experimental results, the four systems are denoted by OptChainV2, RGP2, RGP2+DPV, and RGP2+DPV+ACPc, respectively. Figure 6.13 demonstrates that all four systems scale with the number of shards. The performance of RGP2 is very close to that of OptChainV2. Adding DPV does not improve the performance due to the simple transaction dependencies, but adding ACPc improves the throughput by approximately 20% and reduces the average execution latency by 75% at the specified transaction rates.

Next, we compare the four systems from different aspects in a 16-shard environment. From Figure 6.14a, one can see that ACPc improves the maximum throughput by 37%. Furthermore, Figure 6.14b shows that the first three systems become overloaded when transactions arrive at a 3.8k-tps rate, so the throughput can no longer grow linearly with



(a) Throughput

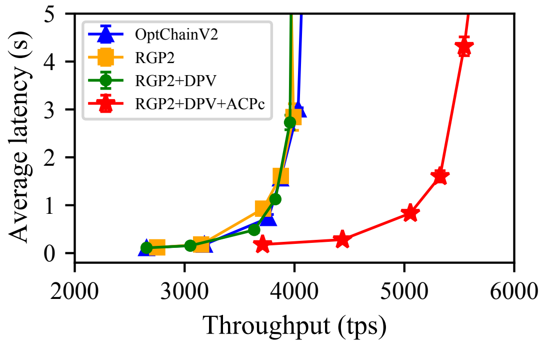


(b) Latency

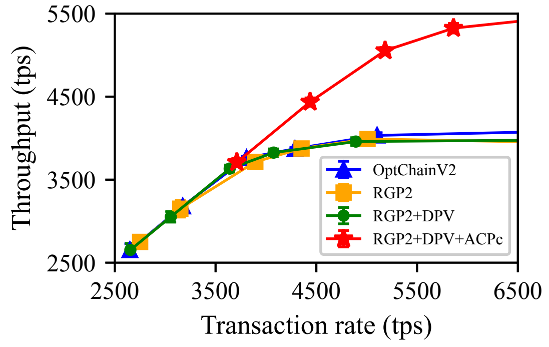
Figure 6.13: Scalability

the transaction rate. ACPc raises the turning point to 5k tps. Unsurprisingly, the latency CDF of OptChainV2, RGP2, and RGP2+DPV are fairly close as well. In contrast, ACPc significantly shortens tail latency: the 95th percentile latency is cut down by 83%, from 30s to 5s. ACPc's success in reducing tail latency implies that slow cross-shard transactions are the reason for long tail latency.

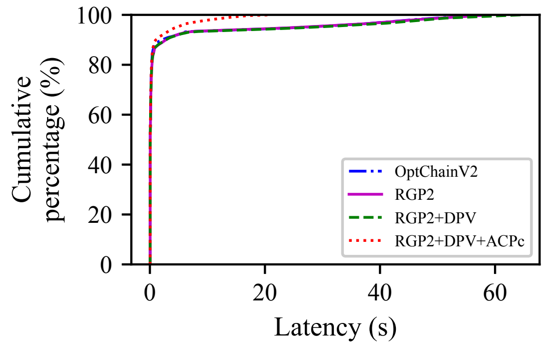
To better understand the performance of the four systems, we investigate the number of *dependency-bound requests*. In our experiments, single-shard transactions are wrapped in `TX` requests, whereas cross-shard transactions are accomplished via `LOCK` and `COMMIT` requests as in Atomix. Dependency-bound requests refer to `TX` requests and `LOCK` requests that are waiting for their parent transactions to be executed, so that their input UTXOs become available. Figure 6.14d shows how the number of dependency-bound requests varies over time, where the y-axis represents the sum of dependency-bound requests across all shards. Unsurprisingly, the curves do not climb up in the first 100s because most transactions are coinbase transactions at the early stage of Bitcoin. After that, the RGP2+DPV+ACPc curve climbs much slower than the other three curves because cross-shard transactions are processed faster and thus hinder less dependent transactions. Contrary to intuition, DPV does not reduce dependency-bound requests. The reason will be given by a comparative analysis in Section 6.4.3.



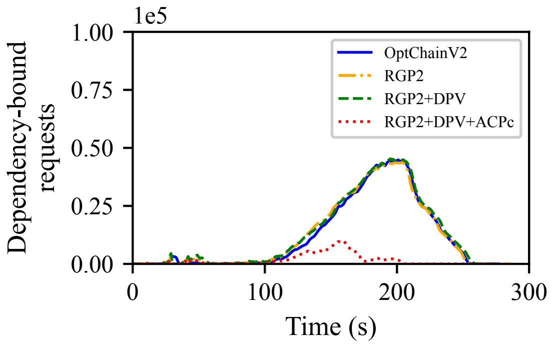
(a) Average latency versus throughput



(b) Throughput versus transaction rate



(c) Cumulative latency

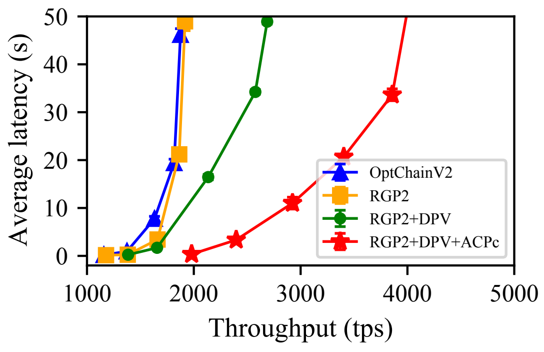


(d) Dependency-bound requests

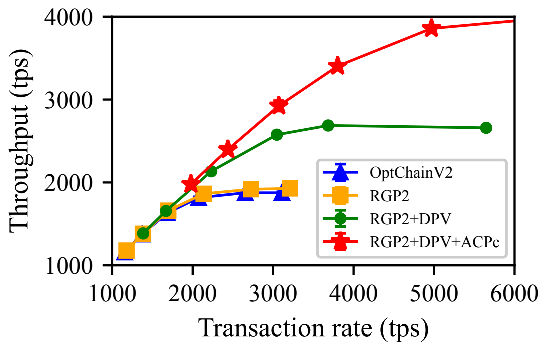
Figure 6.14: Performance with light-dependency transactions (16 shards). The transaction rate is 5k tps in the last two subfigures.

6.4.3 Performance Under Heavy-Dependency Workload

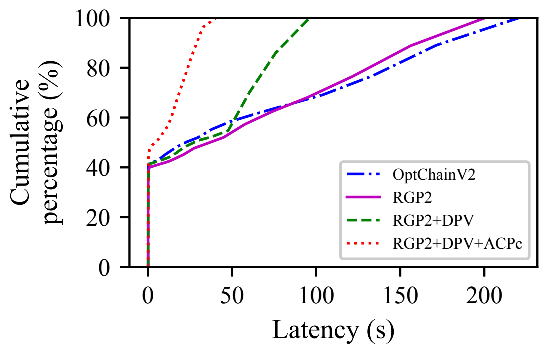
Transactions in blocks [200k, 205k) contain more predecessor-dependent transactions, hence a challenging workload. Nevertheless, RGP2 can still match the performance of OptChainV2 as demonstrated in Figure 6.15a $\sim$ 6.15c. Moreover, adding DPV to the system improves the maximum throughput from 1.9k tps to 2.7k tps (42% up), and ACPc further boosts the maximum throughput to 3.9k tps (another 44% up). In other words, DPV and ACPc jointly double the maximum throughput of the system. Besides, DPV notably lowers tail latency: the 95th percentile latency is halved. ACPc also improves execution latency due to fast cross-shard transaction processing. As a result, DPV and ACPc collectively reduced the 50th percentile latency and 95th percentile latency by 80% and 84% respectively.



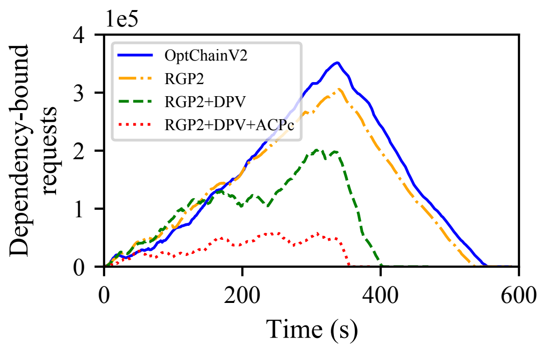
(a) average latency versus throughput



(b) throughput versus transaction rate



(c) cumulative latency

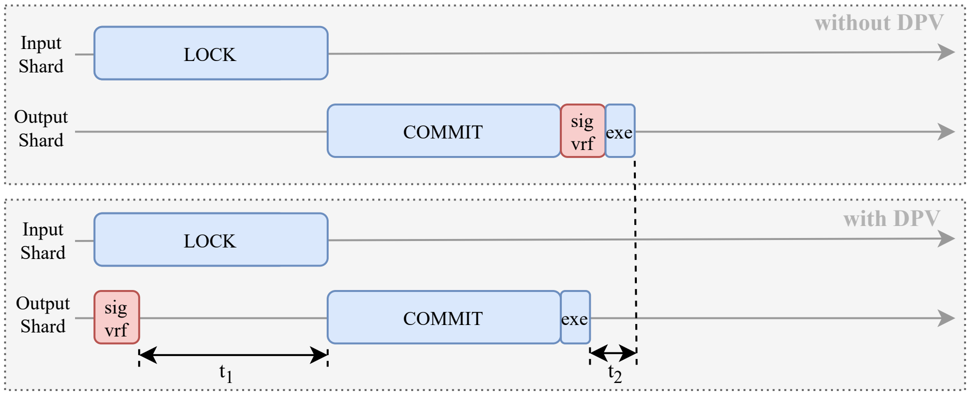


(d) dependency-bound requests

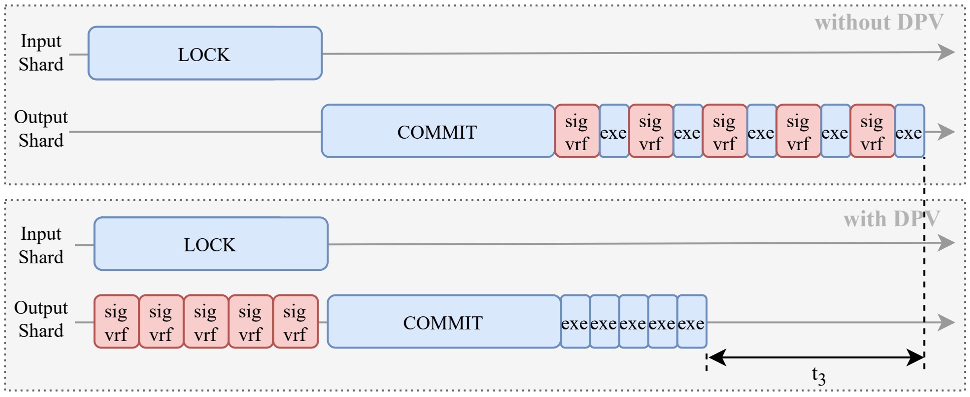
Figure 6.15: Performance with heavy-dependency transactions (16 shards). The transaction rate is 3k tps in the last two subfigures.

DPV improves the system performance because it reduces the number of dependency-bound requests, as shown in Figure 6.15d. With DPV, dependency-bound requests could be pre-verified and later executed immediately after their parent transactions are executed. Therefore, such pre-verified requests have shorter execution latency and are less likely to stall other requests. ACPc also reduces the dependency-bound requests, as cross-shard transactions are processed efficiently and thus become less hindering.

The discrepancy between DPV's performance under the light-dependency workload and that under the heavy-dependency is due to the different ratios of dependency-bound requests to pending `COMMIT` requests. For example, in Figure 6.16a, where the ratio is 1:1, DPV only reduces the overall processing time by t_2, which equals the signature verification time of the dependent request. DPV cannot take advantage of the idling period t_1 since there are no more dependent requests to pre-verify. By contrast, in Figure 6.16b, where the ratio is 5:1, DPV significantly shortens the overall processing time.



(a) Light dependencies



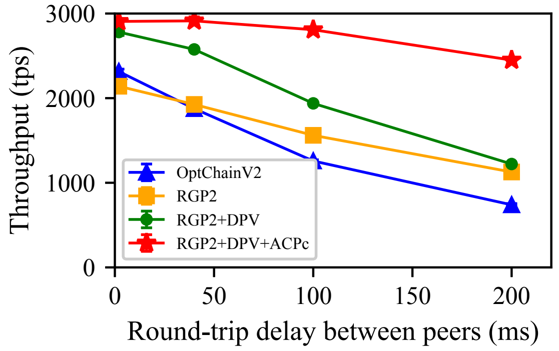
(b) Heavy dependencies

Figure 6.16: DPV saves more time for heavy-dependency workloads.

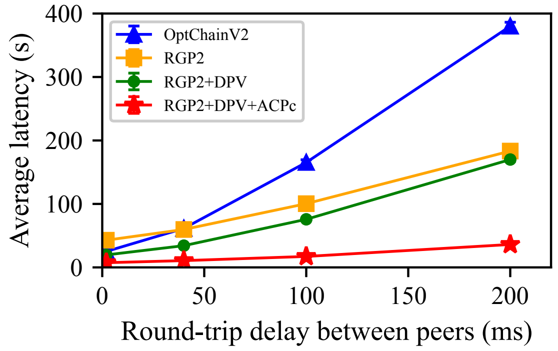
6.4.4 Performance Under Various Network Configurations

Transactions in blocks [200k, 205k) are used in this section, as dependencies in blocks [0, 136k) are too simple to show the effectiveness of DPV.

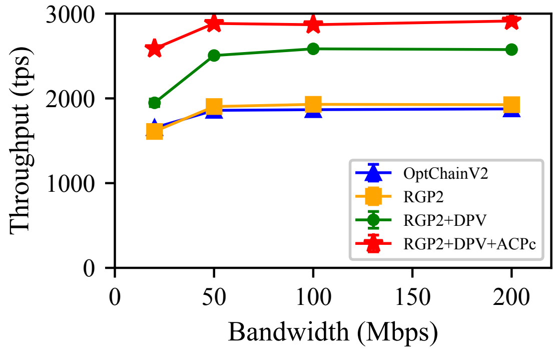
Figures 6.17a and 6.17b demonstrate that all four systems suffer a performance drop as network delay grows. The RGP2+DPV curve goes down faster than the RGP2 curve because long network delays "shift" the performance bottleneck from computation to the network, but DPV only optimizes computation. The OptchainV2 curve also decreases more rapidly than the RGP2 curve. We suspect the reason is that OptChainV2 estimates the transaction queuing delay of a shard by sampling the recent execution rate (denoted by r_e) and the transaction queue length (denoted by n_q). Ideally, $r_e \cdot n_q$ would be the queuing delay that a new transaction would experience if placed to the shard. However, with a high network delay between peers, the high n_q value amplifies the sampling error of r_e.



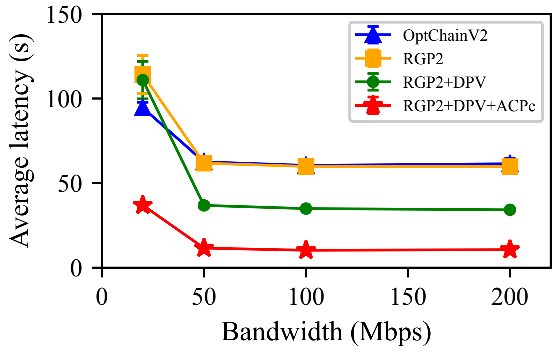
(a) Throughput versus delay



(b) Latency versus delay



(c) Throughput versus bandwidth



(d) Latency versus bandwidth

Figure 6.17: Performance under various network conditions (16 shards, 3k-tps transaction rate)

In terms of bandwidth, all four systems can achieve their full potential with 50-Mbps bandwidth. Extremely low bandwidth hurts the performance of all systems, with DPV suffering the most. This results from the same aforementioned reason—the network becomes the bottleneck instead of computation. In general, ACPc and DPV jointly improve the system performance significantly under all network conditions, though DPV alone is not particularly helpful in harsh network environments.

6.5 Discussion

6.5.1 Advantage of RGP

Although RGP does not surpass OptChainV2 in performance, it offers two big advantages in other aspects. First, RGP does not rely on trust entities. Specifically, information about ancestor transactions—e.g., their respective output shards and whether they are partially spent—are all available on shard ledgers; the partition size of a shard can be estimated with the number of transactions on the shard's ledger. By contrast, to place a transaction with OptChainV2, the client must obtain the fitness score arrays of the parent transactions, which could be generated by other clients. Because fitness score arrays are only available at the generating clients, client-to-client trust is necessary to ensure that clients share fitness score arrays honestly. We consulted the authors about this issue, and the two example use cases they provided are to run OptChainV2 as a public service or inside secure hardware (e.g., Intel SGX). Either case introduces an additional trust point.

Second, RGP does not rely on extra information about shards, i.e., information that cannot be inferred from shard ledgers. Conversely, OptChainV2 requires clients to frequently sample the transaction queue size of every shard for transaction latency estimation. In addition to communication overhead and poor scalability with the number of clients, the sampling is also faced with a security challenge—gleaning true transaction queue sizes from Byzantine-faulty peers is not trivial.

6.5.2 Generalization

In this section, we discuss whether RGP, DPV, and ACPc can generalize to account-balance blockchains. First, RGP does not apply to account-balance blockchains. This is because the difference between UTXO-based blockchains and account-balance blockchains necessitates different approaches to reducing cross-shard transactions, i.e., transaction placement versus account placement, as mentioned in Section 3.3. Although a graph of accounts could be built to partition accounts, RGP does not apply to such graphs. The reason is that RGP is essentially a streaming graph partitioning algorithm, but accounts are not created in a streaming manner.

DPV can be adapted to account-balance blockchains by modifying the definition of dependent transactions accordingly. In an account-balance blockchain, dependencies should be established based on read-write conflicts or write-write conflicts. For example, if two transactions tx_1 and tx_2 both update the same account, and tx_1 is ordered before tx_2 on the shard ledger, then tx_2 is a dependent transaction of tx_1 and must be executed after tx_1. Once dependent transactions are identified, their signatures can be verified before their parent transactions are executed, because signature verification comprises only stateless computing.

Lastly, ACPc does not generalize to account-balance blockchains due to the fundamental difference between the UTXO model and the account-balance model. In the UTXO model, a UTXO is not supposed to be spent by two or more transactions, so coordinators do not have to inform input shards of successful commits. Actually, given a cross-shard transaction, the following two scenarios are equivalent in terms of preventing future transactions from claiming its input UTXO(s): 1) the input shards receive messages confirming the cross-shard transaction's successful execution, and 2) the input shards have locked the input UTXOs but do not receive any message indicating whether the transaction succeeds or not. However, in account-balance blockchains, an account could be updated by multiple transactions, so transaction isolation should be enforced to avoid concurrency issues [30][55]. Therefore, all shards involved in a transaction must be aware of the commit decision so that they can release the involved accounts for other transactions' access. As a result, no shard can skip signing a locking result and sending it to the coordinator, which renders ACPc inapplicable.

6.5.3 Incentive for Clients

When RGP runs on the client side, clients may not be sufficiently motivated to follow the algorithm merely for keeping blockchain systems healthy. To address this, we suggest a transaction fee mechanism where following RGP will lead to low transaction fees. For example, a shard could calculate the fee of a transaction as inversely proportional to the number of its ancestors in the shard. This way, clients will be much more incentivized to send their transactions to the shards with the most ancestor transactions. Shard loads can also be factored into transaction fee calculation in a similar way.

6.6 Chapter Summary

Hashing placement is a common transaction placement algorithm used in blockchain sharding protocols, but produces a large number of cross-shard transactions due to ignoring dependencies. We have developed Rooted Graph Placement and demonstrated that it significantly reduces cross-shard transactions. By considering two levels of ancestor transactions, RGP can match the performance of OptChainV2, a state-of-the-art transaction placement algorithm with additional trust requirements. For the remaining cross-shard transactions, we have devised Dependent Transaction Pre-verification and Atomic Commit Protocol Consolidation to speed up the processing of them and their dependent transactions. DPV makes use of idling computational resources, and ACPc reduces both computational work and network usage. Our experiments have demonstrated that DPV and ACPc jointly can double the maximum throughput under heavy-dependency workloads.

Chapter 7

Conclusions

7.1 Concluding Remarks

This thesis presents techniques to bootstrap new blockchain nodes fast and to improve the performance of sharded blockchains. We believe sharding will be the prevalent approach to scalable blockchains since it has been widely deployed in distributed systems for scalability and will be a key feature of Ethereum 2.0. The techniques described in Chapters 5 and 6 can address some performance bottlenecks of sharded blockchains without compromising security.

Fast Bootstrapping. Long bootstrapping time hinders the expansion of blockchain networks. Our snapshot synchronization approach saves over 99% of the bootstrapping time. By taking advantage of the system state database, we also reduce the snapshot storage overhead. Moreover, the approach can be incorporated into already-deployed blockchains without causing a hard fork.

Transaction Verification Result Sharing. Transaction verification is computationally intensive and thus should only be performed when necessary. Collaborative Transaction Verification allows nodes to reuse the verification work of other nodes, effectively reducing per-node verification work. Dependencies graphs ensure that nodes can verify transactions in different orders without breaking the state machine replication model. Non-sharded BFT-based blockchains could also benefit from Collaborative Transaction Verification.

Smart Transaction Placement. Hashing placement produces too many cross-shard transactions to unleash the performance potential of sharding. We develop Rooted Graph

Placement to reduce cross-shard transactions, together with two techniques for the efficient processing of the remaining cross-shard transactions and their dependent transactions. The three ideas collectively can significantly improve the throughput of a sharded blockchain.

7.2 Future Research Directions

Block Archiving

With snapshot bootstrapping, new nodes do not need access to all the historical blocks for initial synchronization. Although historical blocks may be necessary for other reasons such as analysis or research, such scenarios are arguably infrequent and thus hardly justify the current block storage pattern—every node stores the entire blockchain locally. Moreover, Bitcoin and Ethereum blockchains are over 400GB [14] and 800GB [36] respectively and keep growing, which necessitates a block archiving mechanism to release nodes from heavy storage commitment. Simply selecting some nodes for storing the entire blockchain risks losing block data and puts a heavy storage burden on the selected nodes. A more promising approach is to distribute block storage among peers (i.e., each peer stores only a subset of blocks) with each block replicated sufficient times to tolerate node churn.

Collaborative Signature Verification

In Chapter 5, we have seen that the complexity of Collaborative Transaction Verification mainly comes from transaction dependency handling. However, the most expensive step in transaction verification is signature checking, whose results are not affected by transaction dependencies. Thus sharing only signature verification results instead of whole transaction verification results can greatly simplify the algorithm.

Due to the existence of faulty nodes, a verification result cannot be trusted until it is signed by $f + 1$ nodes. When f is large, verifying the $f + 1$ signatures may cause too much overhead. Collective signing [61] could be explored to aggregate a large number of signatures, thereby reducing the signature verification cost.

Adapting RGP for General Streaming Graph Partitioning

Streaming graph partitioning has many practical applications such as distributed online social network data processing [81][1][94]. Chapter 6 has showcased that two levels of ancestors are enough for RGP to produce reasonably good Bitcoin transaction partitions. It is interesting to explore whether or not RGP also performs well using such little knowledge about the global dependency graph in other applications. In order to adapt RGP for general streaming graph partitioning, the weight for partially spent ancestors should be removed since the UTXO model is not common among systems other than blockchains.

References

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*, pages 76–88, Xi'an, China, 2021. ACM.

[3] Lance Ashdown, Tom Kyte, Jonathan Creighton, Bjørn Engsig, Steve Fogel, Bill Habeck, Min-hank Ho, Bill Hodak, Yong Hu, Pat Huey, et al. Oracle® database concepts 11g release 2 (11.2). 2011.

[4] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *International Conference on Security and Cryptography for Networks*, pages 538–557. Springer, 2014.

[5] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *International conference on financial cryptography and data security*, pages 142–157. Springer, 2016.

[6] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[7] Mirko Bez, Giacomo Fornari, and Tullio Vardanega. The scalability challenge of ethereum: An initial quantitative analysis. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 167–176. IEEE, 2019.

[8] Bitcoincash.org. Bitcoincash: https://www.bitcoincash.org/.

[9] Bitcoin.org. Bitcoin glossary.

[10] Bitcoin.org. Coinbase Input: The Input Of The First Transaction In A Block: https://bitcoin.org/en/developer-reference{#}raw-transaction-format.

[11] Bitcoin.org. Coinbase input: The input of the first transaction in a block, May 2020.

[12] Blockchain.com. Bitcoin Explorer. https://www.blockchain.com/btc/block/680000. Accessed: 2021-09-09.

[13] Blockchain.com. Block 732770.

[14] Blockchain.com. Blockchain size.

[15] Blockchain.com. Bitcoin Explorer. https://www.blockchain.com/btc/tx/91c40e195524962aa3e6cd588e2038b392368382d0815aba7034f51c3ce2579b, 2021. Accessed: 2021-09-08.

[16] Blockchain.com. Block 601147, June 2021.

[17] Ethereum Foundation Blog. Ask about geth: Snapshot acceleration.

[18] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE symposium on security and privacy*, pages 104–121. IEEE, 2015.

[19] Zoltán Böszörményi. *PostgreSQL Replication*. Packt Publishing Ltd, 2013.

[20] Christian Cachin et al. Architecture of the hyperledger blockchain fabric. In *Workshop on distributed cryptocurrencies and consensus ledgers*, pages 1–4. Chicago, IL, 2016.

[21] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. *ACM SIGOPS operating systems review*, 37(5):298–313, 2003.

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

[23] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[24] Rick Cattell. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

[25] Usman W Chohan. The double spending problem and cryptocurrencies. *Available at SSRN 3090174*, 2021.

[26] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[27] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190, 2006.

[28] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Red belly: a secure, fair and scalable open blockchain. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*, 2021.

[29] Kyle Croman, Christian Decker, Ittay Eyal, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[30] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140, 2019.

[31] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10. IEEE, 2013.

[32] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[33] John R Douceur. The sybil attack. In *International workshop on peer-to-peer systems*, pages 251–260. Springer, 2002.

[34] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Annual Cryptology Conference*, pages 585–605. Springer, 2015.

[35] ethereum.org. Shard chains. https://ethereum.org/en/upgrades/shard-chains/, 2022. Accessed: 2022-02-27.

[36] Etherscan. Ethereum full node sync (default) chart.

[37] eth.wiki. On sharding blockchains FAQs. https://eth.wiki/sharding/Sharding-FAQs, 2022. Accessed: 2022-02-28.

[38] Stack Exchange. Sync with bitcoin-qt very slow (0,01%), August 2017.

[39] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. {Bitcoin-NG}: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, pages 45–59, 2016.

[40] Behrouz A Forouzan. *Cryptography & network security*. McGraw-Hill, Inc., 2007.

[41] Bitcoin Forum. Are there faster methods of syncing bitcoin core?, July 2017.

[42] Ethereum Foundation. State tree pruning.

[43] Enrique Fynn and Fernando Pedone. Challenges and pitfalls of partitioning blockchains. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 128–133, Luxembourg, 2018. IEEE.

[44] Adem Efe Gencer, Soumya Basu, Ittay Eyal, Robbert van Renesse, and Emin Gün Sirer. Decentralization in bitcoin and ethereum networks. In *International Conference on Financial Cryptography and Data Security*, pages 439–457. Springer, 2018.

[45] Sanjay Ghemawat and Jeff Dean. Leveldb, 2011.

[46] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[47] Github. Dogecoin block header.

[48] Github. eth/63 fast synchronization algorithm.

[49] Github. Litecoin block header.

[50] GitHub. Bitcoin core integration/staging tree, June 2020.

[51] James N Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[52] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376, 2010.

[53] Rachid Guerraoui and Jingjing Wang. How fast can a distributed transaction commit? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 107–122, 2017.

[54] Jelle Hellings, Daniel P. Hughes, Joshua Primero, and Mohammad Sadoghi. Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing, 2020.

[55] Jelle Hellings and Mohammad Sadoghi. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment*, 14(11):2230–2243, 2021.

[56] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1(1):36–63, 2001.

[57] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Proceedings of The International Conference on Parallel Processing*, 1995.

[58] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388. Springer, 2017.

[59] Sanghyeok Kim, Jeho Song, Sangyeon Woo, Youngjae Kim, and Sungyong Park. Gas consumption-aware dynamic load balancing in ethereum sharding environments. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 188–193, Umea, Sweden, 2019. IEEE.

[60] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[61] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th {usenix} security symposium ({usenix} security 16)*, pages 279–296, 2016.

[62] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[63] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[64] Butler Lampson and Howard E Sturgis. Crash recovery in a distributed data storage system. 1979.

[65] Derek Leung. *Vault: Fast bootstrapping for cryptocurrencies.* PhD thesis, Massachusetts Institute of Technology, 2018.

[66] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for the algorand cryptocurrency. In *NDSS*, 2019.

[67] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[68] Chenxin Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 515–528, 2020.

[69] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *IJ Network Security*, 19(5):653–659, 2017.

[70] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 63–79, 2019.

[71] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[72] Alex Manuskin, Michael Mirkin, and Ittay Eyal. Ostraka: Secure blockchain scaling by node sharding. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 397–406. IEEE, 2020.

[73] Roman Matzutt, Benedikt Kalde, Jan Pennekamp, Arthur Drichel, Martin Henze, and Klaus Wehrle. How to securely prune bitcoin's blockchain. In *2020 IFIP Networking Conference (Networking)*, pages 298–306. IEEE, 2020.

[74] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.

[75] Ralph C Merkle. Method of providing digital signatures, January 5 1982. US Patent 4,309,569.

[76] Avi Mizrahi and Ori Rottenstreich. Blockchain state sharding with space-aware representations. *IEEE Transactions on Network and Service Management*, 2020.

[77] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[78] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[79] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. Optchain: optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535. IEEE, 2019.

[80] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. Optchain: optimal transactions placement for scalable blockchain sharding. *arXiv preprint arXiv:2007.08596v2*, 2021.

[81] Anil Pacaci and M Tamer Özsu. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1375–1392, 2019.

[82] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[83] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. *Cryptology ePrint Archive*, 2016.

[84] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[85] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[86] Reddit. Full node slow to sync. help needed, October 2018.

[87] Liuyang Ren and Paul A. S. Ward. Transaction placement in sharded blockchains, 2021.

[88] Liuyang Ren and Paul AS Ward. Pooled mining is driving blockchains toward centralized systems. In *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 43–48. IEEE, 2019.

[89] Team Rocket. Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies. *Available [online].[Accessed: 4-12-2018]*, 2018.

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

[91] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High performance MySQL: optimization, backups, and replication.* " O'Reilly Media, Inc.", 2012.

[92] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, 1981.

[93] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[94] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, Beijing, 2012. ACM.

[95] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. Mir-bft: High-throughput robust bft for decentralized networks. *arXiv preprint arXiv:1906.05552*, 2021.

[96] Douglas R Stinson. *Cryptography: theory and practice.* Chapman and Hall/CRC, 2005.

[97] Michael Stonebraker, Lawrence A Rowe, and Michael Hirohama. The implementation of postgres. *IEEE transactions on knowledge and data engineering*, 2(1):125–142, 1990.

[98] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. IEEE, 2017.

[99] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 526–545. Ieee, 2016.

[100] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[101] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *Proceedings of the 2006 IEEE international conference on network protocols*, pages 2–11. IEEE, 2006.

[102] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International workshop on open problems in network security*, pages 112–125. Springer, 2015.

[103] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. Non-fungible token (nft): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447*, 2021.

[104] Wenbo Wang, Dinh Thai Hoang, Zehui Xiong, Dusit Niyato, Ping Wang, Peizhao Hu, and Yonggang Wen. A survey on consensus mechanisms and mining management in blockchain networks. *arXiv preprint arXiv:1805.02707*, pages 1–33, 2018.

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

[106] Yang Xiao, Ning Zhang, Wenjing Lou, and Y Thomas Hou. A survey of distributed consensus protocols for blockchain networks. *IEEE Communications Surveys & Tutorials*, 22(2):1432–1465, 2020.

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