RainBlock: Faster Transaction Processing in Public Blockchains

Soujanya Ponnapalli[1], Aashaka Shah[1], Souvik Banerjee[1],
Dahlia Malkhi[2], Amy Tai[3], Vijay Chidambaram[1,3], and Michael Wei[3]

[1]University of Texas at Austin, [2]Diem Association and Novi Financial, [3]VMware Research

Abstract

We present RAINBLOCK, a public blockchain that achieves high transaction throughput without modifying the proof-of-work consensus. The chief insight behind RAINBLOCK is that while consensus controls the rate at which new blocks are added to the blockchain, the number of transactions in each block is limited by I/O bottlenecks. Public blockchains like Ethereum keep the number of transactions in each block low so that all participating servers (miners) have enough time to process a block before the next block is created. By removing the I/O bottlenecks in transaction processing, RAINBLOCK allows miners to process more transactions in the same amount of time. RAINBLOCK makes two novel contributions: the RAINBLOCK architecture that removes I/O from the critical path of processing transactions (txs), and the distributed, multi-versioned DSM-TREE data structure that stores the system state efficiently. We evaluate RAINBLOCK using workloads based on public Ethereum traces (including smart contracts). We show that a single RAINBLOCK miner processes 27.4K txs per second (27× higher than a single Ethereum miner). In a geo-distributed setting with four regions spread across three continents, RAINBLOCK miners process 20K txs per second.

1 Introduction

Blockchains maintain the history of transactions as an immutable chain of blocks; each block has an ordered list of transactions, and is processed after its parent or previous block. Blockchains can be public, allowing untrusted servers to process transactions [1,2], or private, allowing only a few specific servers [17]. With their decentralized nature, fault tolerance, transparency, and auditability, public blockchains have led to several applications in a wide range of domains like cryptocurrencies [1, 2, 19], games [29], and healthcare [43].

In general, public blockchains work in the following manner. Servers participating in the blockchain, termed miners, receive transactions from users. Miners execute the transactions and package them up into blocks. A consensus protocol, like proof-of-work (PoW) [34], decides the next block to be added to the blockchain. With PoW, miners release a new block at a regular cadence (*e.g.,* every 10–12 seconds in Ethereum [2]).

Problem: Low throughput. Public blockchains suffer from low transaction throughput. Two popular public blockchains, Bitcoin [1] and Ethereum, process only tens of transactions per second, limiting their applications [33, 65].

Metric	No state	State: 10M	Ratio
Time taken to mine txs (s)	1047	6340	6× ↑
# Txs per block	2150	833	2.5× ↓
Tx throughput (txs/s)	28.6	4.7	6× ↓

Table 1: **Impact of system state on blockchain throughput**. This table shows the throughput of Ethereum with proof-of-work consensus when 30K txs are mined using three miners, in two scenarios: first, there are no accounts on the blockchain, and in the second, 10M accounts have been added. Despite no other difference, tx throughput is 6× lower in the second scenario; we trace this to the I/O involved in processing txs.

Prior work traces back the low throughput of blockchains to their proof-of-work (PoW) consensus. PoW limits the block creation rate so that miners have enough time to receive and process the previous block before the next block is created [59]. This ensures that most of the miners are building on the same previous block, preventing forks in the blockchain. Researchers have proposed new consensus protocols [5, 30, 31, 42, 46] to increase the transaction throughput.

Insight. We observe that while PoW limits the block creation rate, it *does not* limit the block size (number of transactions in each block). The block size is limited by the rate at which miners can process transactions (§2). Processing transactions involves executing a transaction and modifying the system state accordingly; this becomes more expensive as the state grows. Table 1 experimentally shows that when the number of accounts increases, block size reduces in Ethereum even with PoW consensus (only among three miners). The chief insight in this paper is that *if we could increase the rate at which transactions are processed, we could increase the block size safely, without modifying the PoW consensus protocol.*

How can we increase the block size safely? Miners will continue to release a block every 10–12s after proof-of-work; however, miners can pack more transactions into each block due to faster processing. Typically, increasing the block size increases the time taken to transmit that block, and time taken by miners to process the block. Previously, when Ethereum

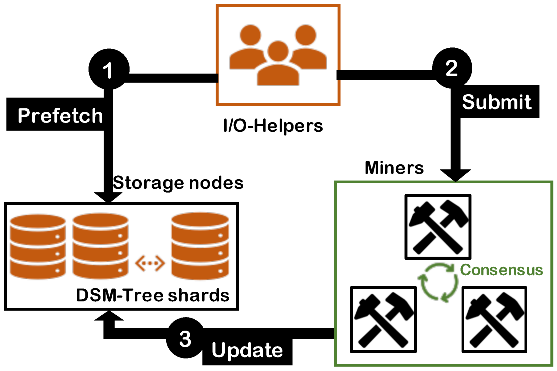


Figure 1: **RainBlock architecture**. I/O-Helpers read data from in-memory storage nodes (out of the critical path) on behalf of the miners. Miners execute txs without performing I/O; storage nodes are updated asynchronously.

increased the maximum block size [21], it was observed that the time taken to propagate the block increased marginally, while the time taken to process transactions saw a significant increase [22]. Thus, if we can increase the rate at which transactions are processed, miners can pack more transactions into each block. Note that as proof-of-work is not modified, the safety properties will continue to hold. The block creation rate, and the transaction confirmation latency will remain the same. However, the rate at which transactions are confirmed will increase as there will be more transactions in each block.

Approach: reducing I/O bottlenecks in tx processing. This work takes the novel approach of increasing tx throughput by tackling I/O bottlenecks in tx processing. Processing a transaction involves executing the transaction and verifying that the execution result is valid. In Ethereum, both execution and verification require reading and writing system state that is stored in a Merkle tree [11,44] on local storage. Tx processing is bottlenecked by I/O: for example, processing a single block of 100 txs in Ethereum requires performing more than 10K random I/O operations (100× higher) and takes hundreds of milliseconds even on a datacenter-grade NVMe SSD (§2).

These I/O bottlenecks arise from two sources. First, the Merkle tree is serialized and stored in a RocksDB [10] key-value store. As a result, traversing the Merkle tree requires multiple, expensive RocksDB reads [53]. Second, Ethereum miners process txs one by one in the critical path; this reduces parallelism and allows I/O bottlenecks to limit the tx throughput. These I/O bottlenecks are an intrinsic part of how Ethereum (and more generally, public blockchains that use Merkle trees) are designed [54]; merely upgrading to faster storage (or even holding all state in memory) will not resolve these problems. By removing these I/O bottlenecks, this work accelerates tx processing, and allows miners to pack more txs in each block, and thereby increases the overall throughput.

This work. This paper presents RainBlock, a new architecture for building public blockchains, that increases the overall throughput with faster transaction processing. RainBlock tackles I/O bottlenecks by designing a custom storage solution for blockchains; at the heart of this storage is a novel data structure, the Distributed, Sharded Merkle Tree (DSM-Tree). The DSM-Tree stores data in a multi-versioned fashion across shards, and allows concurrent reads and writes. DSM-Tree eliminates the inherent serialization from using key-value stores like RocksDB. RainBlock deconstructs miners into three entities: storage nodes that store data, miners that process txs, and I/O-Helpers that fetch data and proofs from storage nodes and provide them to miners, as shown in Figure 1. By having I/O-Helpers prefetch data on behalf of miners, I/O is removed from the critical path of tx processing; moreover, multiple I/O-Helpers can prefetch data at the same time.

Challenges. The RainBlock architecture has to solve several challenges to be effective. For example, Ethereum considered using stateless clients [24] with separate storage nodes; proofs that validate the data sent (termed witnesses) are included in the blocks. The size of these blocks that must be sent over the network made the scheme impractical [25,27], and the proposal was abandoned [57]. RainBlock handles this by co-designing the storage nodes and caches at the miners, minimizing the data sent over the network. A second challenge is prefetching data for Turing-complete smart contracts [56]. As smart contracts can execute arbitrary code, it is not straightforward for the I/O-Helpers to prefetch all the required data. RainBlock solves this by having I/O-Helpers *speculatively pre-execute* the transaction to obtain data and proofs. A third related challenge is that I/O-Helpers may submit stale proofs to the miners (miners may update storage after I/O-Helpers finish reading). RainBlock handles this by having miners tolerate stale proofs whenever possible, via *witness revision*, allowing transactions that would otherwise abort to execute.

Implementation. We implement the RainBlock prototype using Ethereum. We chose Ethereum as our implementation base and comparison point for two reasons. First, Ethereum has been operating as a public blockchain for nearly six years, providing large amounts of data to test our assumptions. Second, Ethereum supports Turing-complete smart contracts, allowing the codification of complex decentralized applications.

Security and Trust. Ethereum has increased the block size in the past [14], without triggering any negative events such as increased forks [16]. RainBlock does the exact same thing, without changing the PoW consensus or block creation rate; as a result, RainBlock inherits the security guarantees of Ethereum. RainBlock introduces storage nodes and I/O-Helpers; however, neither storage nodes, nor I/O-Helpers, nor miners, trust each other. All data exchanged between these entities is authenticated using Merkle trees and can be verified. As a result, the new architecture of RainBlock, while certainly increasing the complexity of the system, does not change its core security or trust assumptions (§3.6).

Evaluation. To evaluate RainBlock, we generate synthetic workloads that mirror transactions on Ethereum mainnet. We analyzed Ethereum transactions and observed that user accounts involved in transactions have a Zipfian distribution: 90% of transactions involve the same 10% of user accounts. We observed that only 10–15% of Ethereum transactions invoke smart contracts. We build a workload generator that faithfully reproduces these distributions. We evaluate RainBlock using these workloads and observe that a single RainBlock miner processes 27× more transactions than an Ethereum miner; we provide a breakdown of the performance difference (§6). When the RainBlock miners are geo-distributed across three continents (thus incurring significant network latency), RainBlock throughput reduces only by 20% compared to a single miner, achieving 20K transactions per second. Since RainBlock has the same proof-of-work consensus and block creation rate as Ethereum, RainBlock finalizes 20× more transactions (with the same latency) as Ethereum.

In summary, this paper makes the following contributions:

- The novel RainBlock architecture (§3)
- The novel DSM-Tree authenticated data structure (§4)
- Empirical evidence that throughput in public blockchains can be increased without modifying PoW consensus (§6)

2 Background and Motivation

We provide some background on public blockchains and detail the I/O bottlenecks in transaction processing.

2.1 Public Blockchains and Ethereum

Blockchains are decentralized databases with support for transactions (txs). Blockchains log these txs as a chain of blocks; every block stores an ordered list of txs along with the cryptographic hash of its previous block (parent). Private blockchains allow specific servers to extend the blockchains, while public blockchains allow anyone to participate. As a result, all participating servers in public blockchains are untrusted; any of them can be malicious. Public blockchains depend on the non-malicious majority for correct behavior.

How do public blockchains work? We use Ethereum, a popular public blockchain, as reference. Servers that attempt to add a new block to the blockchain are termed *miners*. Miners receive txs submitted by users, execute these txs, and group them into a block. We term executing txs and grouping into a block as *processing txs*. Several miners compete to add a block to the blockchain. Each miner tries to solve a proof-of-work (PoW) puzzle; the miner that solves the puzzle attaches the solution and broadcasts its block to other miners. Other miners verify the PoW solution and the txs in the block, and build on top of the block. A tx is *confirmed* or *finalized* once ten blocks are built on top of the block containing the tx.

Typically, a miner has two threads as shown in Figure 2. The worker thread executes and groups txs into a partial block. The sealer thread obtains the partial block from the worker, and then tries to solve the PoW puzzle; PoW consensus has miners emitting a block every 10–12s, for example. While the sealer thread is working on one block, the worker thread tries to get the next block ready. Thus, the worker has about 10–12s to process txs; if it can process txs faster, it can pack more txs into the partial block that it passes to the sealer.

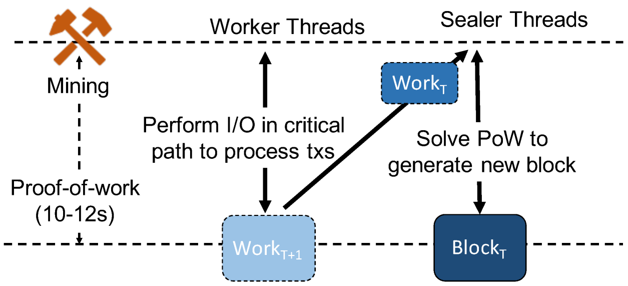


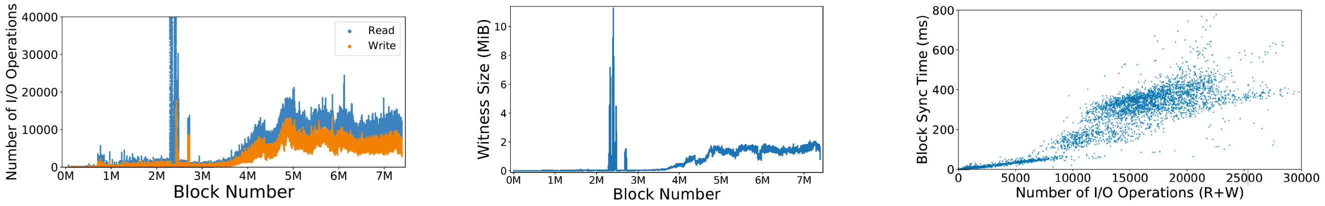
Figure 2: **How Ethereum miners work**. The worker thread processes txs, packages them into a future/partial block, and hands them to the sealer thread. The sealer thread solves the PoW puzzle (in 10–12s), and propose a new block; the worker thread must process txs in this time-frame. I/O bottlenecks result in worker threads packing fewer txs into each block.

2.2 Problem: Low throughput

Ethereum and other public blockchains suffer from low throughput: only tens of txs are added to the blockchain per second. The low throughput comes from two factors. First, the PoW consensus limits the block creation rate to one block every 10–12s so that a majority of miners can receive and process a block before a new one is released; this ensures that every miner is building on the same previous block, preventing forks in the blockchain. Note that while PoW consensus limits the block creation rate, it doesn't directly limit the size of the block. The factor limiting the block size is tx processing time: the rate at which the worker thread can process txs limits the maximum size of the block, as shown in Figure 2.

Tx processing. Processing a tx involves executing the tx and modifying the system state like account values. Since miners do not trust each other in a public blockchain, miners *authenticate data* and can prove that the data they provide is correct. This is done by maintaining a Merkle tree [44] over the data and publishing the latest Merkle root in the blocks; another miner is able to independently execute txs in a block and verify that its local Merkle root matches the root in the block. An account value and its vertical path in the Merkle tree (termed a *witness*) are sufficient to verify the correctness of that value.

Unfortunately, *accessing and authenticating data becomes more expensive as the total size of the data increases* [64]. As a result, tx processing increasingly becomes bottlenecked on I/O. We demonstrate this with an experiment. We create two private Ethereum networks using the Geth client [13];



(a) Number of I/O operations

(b) Witness sizes

(c) Block processing latency

Figure 3: **I/O bottlenecks in processing Ethereum txs**. The increasing system state increases (a) the number of I/O operations performed per block, and (b) the amount of authenticated data accessed (Merkle witnesses) per block. Further, (c) the direct correlation between the block processing time and the number of I/O operations highlights the I/O bottleneck. Spikes in (a) I/O operations and (b) witness sizes are due to a DDOS attack that targeted the system state [64] by creating dummy user accounts.

each network has three miners, 30K txs to mine, and the same proof-of-work (PoW) configurations. While the first network has only 3 miner accounts (total size: 220 MB), the second has 10M additional accounts (total size: 4 GB). Note that Ethereum currently has ≈130M accounts (total size: > 400 GB [12, 15]) in its blockchain state. Overall, the second network takes 6× more time and 2.5× more blocks to mine all txs using PoW consensus (Table 1). Using profilers, we see that in the second scenario, the time spent solving the PoW puzzle has increased proportionately; however, the worker thread takes 6× more time to process txs, and 69% of this time is spent in accessing and updating the authenticated system state. Thus, miner's tx processing rate depends on the system state; this impacts the block size and overall throughput. Ethereum uses the modified Merkle Patricia Trie [11] to authenticate the state, we refer to it as the Merkle tree.

Empirical Study. We now study the I/O overheads observed on the Ethereum public blockchain. We use the Parity 2.2.11 client [8] to initialize a new server that joins the Ethereum network, and replays the blockchain (until 7.3 million blocks) to measure various costs. Since we want to observe the historical I/O cost, we do not mine new blocks; rather, we replay the blockchain and execute transactions in blocks that have already been added, termed syncing. The I/O cost remains the same as when mining the block for the first time.

Reading Eth. accounts results in multiple I/O operations. For processing a single block with around 100 transactions, Ethereum performs more than 10K random I/O operations (two orders of difference). Most of these I/O operations are performed for reading and updating user accounts in the Merkle tree. Figure 3(a) shows the number of I/O operations incurred in each block while processing txs, till 7.3 M blocks.

Data authentication causes significant I/O overhead. A witness is the vertical path in the Merkle tree required to verify a data item. Witness sizes represent the amount of data read and modified per block while reading and updating paths in the Merkle tree. In Ethereum, which uses secure 256-bit cryptographic hashes, the witness size of a single 100 byte user account (or value) can be above 4 KB, resulting in 40-60× overhead. The witness size also increases as the total data in the Ethereum state increases, as shown in Figure 3(b).

Tx processing is bottlenecked by I/O. We measure the time taken to process (or sync) each block: executing the txs in that block, and verifying if the resultant local Merkle root matches with the Merkle root in that block. Processing an Ethereum block with about 100 txs takes hundreds of milliseconds even on a datacenter-grade NVMe SSD. Figure 3(c) shows the direct correlation between the time taken to process Ethereum blocks and the number of I/O operations performed, indicating that tx processing in Ethereum has I/O bottlenecks.

Summary. The I/O bottlenecks stem from storing the Merkle tree in an LSM-based [49] key-value store like RocksDB [10] that has inherent I/O amplification [52, 53]; Merkle nodes are indexed with their hash, randomizing node locations on disk.

2.3 Straw-man solutions

We consider some straw-man solutions and discuss why they are not suitable for handling I/O bottlenecks in tx processing.

Storing entire system state in memory. Public blockchains like Ethereum do not want to place restrictions on the hardware specifications of miners. If every miner is required to hold the entire authenticated state in memory, only machines with large amounts of DRAM would be able to process transactions. This solution breaks one of the central tenets of Ethereum that contributes to the decentralization of its miners.

Increasing block size. Increasing the block size is the goal of this work; however, doing this naively would not work. If we simply increased the number of txs in each block, miners receiving the block would need more time to process the large block and build on top of it. As a result, the block creation rate would have to be lowered to ensure that previous block is processed by a majority of the miners before a new block is released. Overall, tx throughput would not increase though the block size increased. Over time, I/O bottlenecks exacerbate with the increasing state size, and new servers will take longer to sync and participate in mining. This weakens decentralization. Thus, tackling I/O bottlenecks is crucial to safely increasing block size, and maintaining decentralization.

Alternative consensus protocols. Faster consensus protocols would result in blocks being released quicker, increasing the overall throughput. The goal in this work is to increase throughput *without* changing the consensus, and thus is orthogonal to the work on new consensus protocols. Researchers have noticed that even with faster consensus, blockchains ultimately run into the I/O bottlenecks in tx processing [67].

Thus, we need a mechanism to reduce the I/O bottlenecks in transaction processing. RAINBLOCK achieves this goal with a new architecture and a novel authenticated data structure, the DSM-TREE. With faster transaction processing, RAINBLOCK enables larger blocks and maintains decentralization, *without* changing the PoW consensus or the block creation rate.

3 RainBlock

RAINBLOCK is a public blockchain based on Ethereum that increases overall throughput with faster transaction processing. RAINBLOCK minimizes the I/O bottlenecks in transaction processing, allows miners to safely pack more transactions into each block, and thereby increases the overall throughput.

3.1 Overview

RAINBLOCK minimizes I/O bottlenecks using two techniques. First, it makes each I/O operation cheaper by storing system state in the novel DSM-TREE. In contrast to Ethereum, which accesses data from the RocksDB key-value store on SSDs (where each RocksDB get operation takes between a few hundred microseconds to a few milliseconds), RAINBLOCK stores the system state in memory using the DSM-TREE. DSM-TREE is a sharded, multi-versioned, in-memory authenticated data structure. Second, RAINBLOCK introduces a new architecture that removes I/O from the critical path. RAINBLOCK deconstructs miners into three entities: storage nodes, miners, and I/O-Helpers that read data from storage and submit to miners. In the common case, miners can process txs without performing I/O. Neither miners, nor storage nodes, nor I/O-Helpers trust each other: all data supplied by other entities is verified using Merkle witnesses before use.

RAINBLOCK differs from Ethereum in exactly two aspects: 1) miners are replaced by storage nodes, miners, and I/O-Helpers, and 2) the RocksDB-based storage is replaced with DSM-Tree. Everything else, like the proof-of-work consensus and the block creation rate, remains the same.

3.2 Building up the design step by step

In this section, we start with the problems that our study on Ethereum highlights. We discuss how RAINBLOCK solves these problems and addresses the resulting challenges.

Problem-I: storing authenticated state in key-value stores leads to expensive I/O. Ethereum stores system state in a Merkle tree [11], which is stored in the RocksDB [10] key-value store. Traversing such a Merkle tree requires looking up nodes using their hashes. Hashing is computationally expensive and results in the nodes of the tree being distributed to random locations on storage. As a result, traversing the Merkle tree to read a leaf value requires several random read operations. Further, the log-structured merge tree [47] that underlies RocksDB results in high I/O amplification [52, 53].

Solution: store state in an optimized in-memory representation. RAINBLOCK introduces an in-memory version of the Merkle tree. Persisting the data is done via a write-ahead log and checkpoints. Traversing the Merkle tree is *decoupled* from hashing; obtaining the next node in the tree is a simple pointer dereference (§4). Note that simply running RocksDB in memory would not be effective, as serializing and hashing Merkle nodes would still add significant overhead.

Resulting challenge: scalability and decentralization. As the blockchain grows, the amount of state in the Merkle tree will increase; soon, a single server's DRAM will not be sufficient. Furthermore, for maintaining decentralization, we cannot require miners participating in the blockchain to have significant amounts of DRAM.

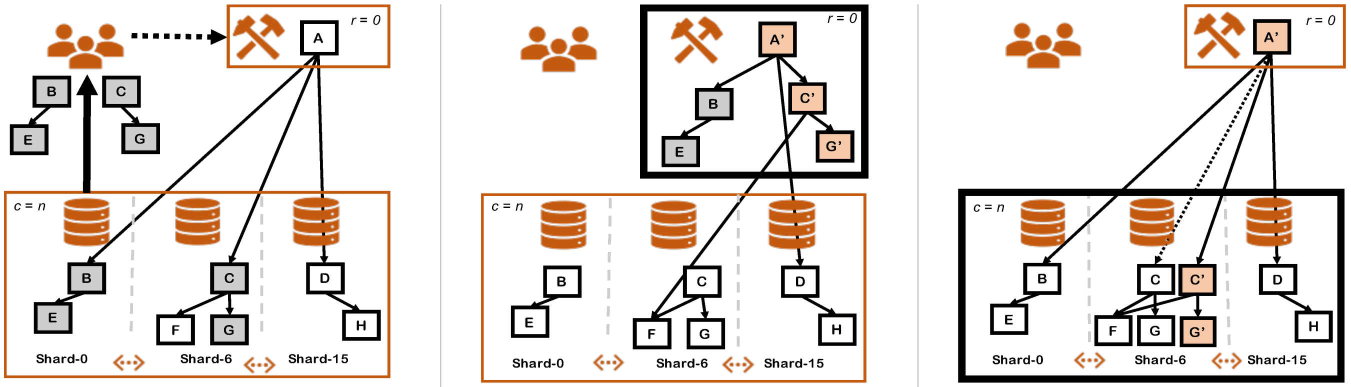
Solution: decouple storage from miners and shard the state. RAINBLOCK solves this problem using separate storage nodes, each of which is a commodity server. RAINBLOCK shards the Merkle tree into subtrees such that each subtree fits in the memory of a storage node. As the amount of data in the state increases, RAINBLOCK increases the number of shards. In this manner, RAINBLOCK scales with commodity miners and storage nodes without reducing the decentralization.

Problem-II: Miners perform slow I/O in the critical path. Transaction processing in Ethereum includes performing slow I/O operations in the critical path, and these transactions are processed one at a time.

Solution: decouple I/O and transaction execution. RAINBLOCK solves this problem by removing the burden of doing I/O from the miners. RAINBLOCK introduces I/O-Helpers that prefetch data and witnesses from the storage nodes and submit them to the miners. Miners use this information to execute transactions without performing I/O and asynchronously update the storage nodes. This architecture also increases parallelism as multiple I/O-Helpers can be prefetching data for different transactions at the same time.

Resulting challenge: Prefetching I/O for smart contracts. One challenge with I/O-Helpers prefetching data is that some transactions invoke smart contracts. Smart contracts are Turing-complete programs that may execute arbitrary code. Thus, how does the I/O-Helper know what data to prefetch?

Solution: pre-execute transactions to get their read and write sets. RAINBLOCK solves this problem by having the I/O-Helpers pre-execute txs. As part of this pre-execution, I/O-Helpers read data and witnesses from the storage nodes. One challenge is that the pre-execution may have different results than when the miner executes the tx (*e.g.,* the smart contract



(A) I/O-Helpers prefetch witnesses

(B) Miners execute txns without I/O

(C) Shards update asynchronously

Figure 4: **RainBlock architecture**. This figure shows how RainBlock processes a transaction that reads and updates accounts in two different shards that are along the paths *ABE* and *ACG*. (A) I/O-Helpers prefetch witnesses *BE* and *CG* from storage nodes and submit them to the miners. (B) Miners verify the witnesses and use them to execute the transaction against their top layer of the DSM-Tree. Then, miners update the storage nodes. (C) Storage nodes verify updates from miners and asynchronously update their bottom layer, creating a new version of the modified Merkle nodes $A'C'G'$. Here, miners retain only the root node of the DSM-Tree ($r = 0$), and the storage nodes send full un-compacted witnesses ($c = n$ or $c = \infty$).

may execute different code based on the block it appears in). We will describe how I/O-Helpers handle smart contracts correctly despite stale data from pre-execution (§3.4).

Resulting challenge: Consistency in the face of concurrency. Another challenge that arises from decoupling I/O from transaction execution is consistency. Multiple I/O-Helpers are reading from the storage nodes, and multiple miners are updating them in parallel. Using locks or other similar mechanisms will reduce concurrency and throughput.

Solution: using the two-layered, multi-versioned DSM-Tree. RainBlock uses DSM-Tree to store the system state. The DSM-Tree has two layers: the *bottom layer* is sharded across the storage nodes and contains multiple versions. Every write causes a new version to be created in a copy-on-write manner; there are no in-place updates. As a result, concurrent updates from miners simply create new versions and do not conflict with each other. Further, each miner uses a DSM-Tree *top layer*: a private, writeable, consistent snapshot of the data, that reflects the miner's latest version.

Resulting challenge: RainBlock has higher network traffic. Finally, the architecture of RainBlock trades local disk I/O for remote network I/O. As a result, RainBlock results in more network utilization, and the network bandwidth may become the bottleneck.

Solution: RainBlock reduces network I/O via deduplication and the synergy between the DSM-Tree layers. RainBlock uses multiple optimizations to reduce network I/O. First, the bottom and top layer of DSM-Tree collaborate with each other to reduce network traffic. Second, when any component of RainBlock sends witnesses over the network, it will batch witnesses and perform deduplication to ensure only a single copy of each Merkle tree node is sent. Finally, miners send logical updates to storage nodes rather than physical updates as logical updates are smaller in size.

3.3 Architecture

RainBlock introduces three kinds of participating entities: I/O-Helpers, miners, and storage nodes. Users send txs to I/O-Helpers, which pre-execute these txs and prefetch data and witnesses from storage nodes. Figure 4(a) shows how I/O-Helpers submit txs and the prefetched information to miners. Miners are responsible for creating new blocks of transactions and extending the blockchain; each miner maintains a private DSM-Tree top layer. Figure 4(b) shows how miners use the submitted information to execute these transactions without performing I/O. Finally, miners create a new block, gossip it to other miners, and update the storage nodes. Storage nodes are responsible for maintaining and serving the system state. They use the multi-versioned bottom layer of the DSM-Tree to provide consistent data to I/O-Helpers while handling concurrent updates from miners. Figure 4(c) shows how the miners asynchronously update the bottom layer of the DSM-Tree at the storage nodes.

3.4 Speculative Pre-Execution

I/O-Helpers read all the witnesses required for executing a tx from storage nodes. While this is straight-forward for simple txs, how do I/O-Helpers handle Turing-complete smart contracts that may access arbitrary locations? I/O-Helpers handle this by *speculatively* pre-executing the smart contract to obtain the read and write set.

However, smart contracts can use the timestamp, or block number of the block in which they appear, during their execution at the miner. These values are not yet known during their pre-execution at the I/O-Helpers. I/O-Helpers speculatively return an estimated value while pre-executing the contract.

Our analysis of Ethereum contracts shows that despite providing estimated values, I/O-Helpers still successfully prefetch the correct witnesses and node bags. For example, the CryptoKitties `mixGenes` function references the current block number and its hash. Since these numbers only affect written values (and not the read set), substituting approximate values does not affect the witnesses that are prefetched.

We observe that I/O-Helpers can pre-execute with *stale* data and still prefetch the correct witnesses. For example, many contracts are *fixed-address* contracts: their behavior depends only on call inputs. To deal with rare *variable-address* contracts, the miner may asynchronously read from storage nodes after the transaction is submitted. Even in these cases, the I/O-Helper will have retrieved some of the correct witnesses required for the tx (*e.g.,* the `to` and `from` accounts).

3.5 Life of a Transaction in RainBlock

We outline the various actions that take place from the time a tx is submitted, to when it becomes a part of the blockchain.

1. I/O-Helper pre-executes the transaction by fetching data and witnesses from the storage nodes
2. I/O-Helper batches and deduplicates Merkle nodes across multiple witnesses and sends these optimized witnesses (termed node bags), txs, and data, to the miner
3. Miner verifies the node bags and advertises them to others.
4. Miner executes the tx using its top layer and the node bags, without any I/O; miner caches all Merkle nodes it reads or revises from these node bags in its top layer
5. Miner, on solving PoW, creates and advertises a new block to other miners; miner also sends the block (with a new Merkle root) and the logical updates to the storage nodes
6. Storage nodes first validate the block (check if PoW solution in the block solves the puzzle), and then persist the logical updates and return successful to the miner.
7. Storage nodes apply the updates asynchronously, and check if their Merkle root matches the root in the block
8. Other miners validate the block and gossip it to others. Then miners execute its txs using node bags, and accept the block by mining new blocks on top of this block
9. Once a majority of the miners receive, validate, and accept the block, the tx becomes part of the blockchain
10. Once ten or more blocks are mined on this block, the tx is confirmed; storage nodes garbage collect the associated versions from the unconfirmed, competing blocks

3.6 Discussion

We note that RainBlock differs from Ethereum in only two aspects: its architecture and its storage. We discuss how these affect trust, incentives, and security, and discuss the trade-offs.

Trust assumptions. RainBlock does not require trust between any of its components. Miners operate without trusting I/O-Helpers or the storage nodes, as miners re-execute transactions and verify the data they receive from I/O-Helpers; I/O-Helpers verify the data they read from storage nodes; and storage nodes verify new blocks and updates from miners.

Incentives. RainBlock shares elements of its architecture with the Ethereum stateless clients proposal [24] that received community support. The central question is how are the storage nodes incentivized to store and serve the latest system state? We propose a model where I/O-Helpers or users pay storage nodes; stateless clients proposal had a similar solution where users pay storage nodes for access to state via state channels. I/O-Helpers or users can always detect if the data served by storage nodes is incorrect or stale, and penalize any malicious shards. We also believe an ecosystem will develop around RainBlock architecture, with commercial entities offering active storage backups; market economics drives these backups to provide and maintain the latest system state with high availability. Note that I/O-Helpers are an optimization and users can prefetch data themselves if required, or pay the I/O-Helpers. Finally, RainBlock miners behave similar to the miners in Ethereum, and are incentivized to process txs via block rewards. Miners are incentivized to broadcast correct updates to storage nodes to aid the acceptance of their fork. Thus, we outline a few ways to incentivize RainBlock components and leave the full solution to future work.

Security. RainBlock provides similar security guarantees as Ethereum, as it does not change the proof-of-work consensus or trust assumptions between participating servers. RainBlock does not impact the block creation rate, as it packs more transactions per block without changing the total time taken to process a block of transactions.

Availability and DDoS attacks. RainBlock decouples storage from miners and has separate storage nodes. RainBlock requires only one replica of each storage shard to be available for making progress. While DDoS attacks can be mounted on storage nodes in RainBlock, they are not new, cannot tamper with the data, and do not impact the correctness of RainBlock; DDoS attacks are also possible on Ethereum and have been successfully executed in the past [60, 64].

Trade-offs. RainBlock introduces new storage nodes and I/O-Helpers; while this adds more complexity into the system, Ethereum was already considering adding storage nodes and stateless clients. Thus, we believe the additional complexity of RainBlock is a good trade-off for its scalability and performance benefits. RainBlock trades off local storage I/O for accessing memory over network, so the network may become a bottleneck. RainBlock recognizes this risk and uses multiple techniques like utilizing the memory available at the miners to cache the top layer of the DSM-Tree, and performing witness compaction and deduplication to reduce network traffic (§4.3). Although RainBlock uses extra resources for separate storage nodes, storage nodes are shared across miners, amortizing the costs. I/O-Helpers also use extra resources; however, they are a performance optimization and can execute read-only txs without involving the miners. If users judge I/O-Helpers not useful, users can prefetch from storage nodes, or turn off prefetching, causing miners to perform I/O.

RainBlock adoption. RainBlock can be deployed incrementally in principle. While RainBlock miners rely on storage nodes and I/O-helpers, other miners can use their local RocksDB-based storage, as the two would be compatible.

4 DSM-Tree

RainBlock stores the system state in the DSM-Tree data structure. The Distributed, Sharded Merkle Tree (DSM-Tree) is an in-memory, multi-versioned, sharded, two-layer variant of the Merkle tree. We first present the in-memory representation of the two layers, then describe each layer in detail, and then discuss how the layers collaborate and their trade-offs in different configurations.

In-Memory Representation. DSM-Tree builds the Merkle tree in memory using pointers. Tree traversal is decoupled from hashing and node serializations: traversing the DSM-Tree requires dereferencing pointers to the next Merkle node; in contrast, Ethereum's Merkle tree reads a Merkle node from RocksDB using its hash, and then deserializes the node to find the cryptographic hash of the next node. DSM-Tree uses periodic checkpoints for persisting the data. The checkpoints are only used to reconstruct the in-memory data structure in case of failures; reads are always served from memory.

Lazy Hash Resolution. When a leaf node in a Merkle tree is updated, hashes of nodes from the leaf to the root need to be recomputed. Recomputing hashes is expensive as nodes have to be serialized before being hashed. DSM-Tree defers this recomputation; on writes, only the leaf nodes are updated; on a subsequent read, all the modified nodes are rehashed exactly once. Thus, lazy hash resolution improves performance significantly by reducing the number of expensive node hashes and RLP (Recursive Length Prefix) serializations [9].

4.1 Bottom Layer

The bottom layer consists of a number of shards. Each shard is a vertical subtree of the Merkle tree, stored in DRAM. The bottom layer supports multiple versions to allow concurrent updates, as shown in Figure 5. The bottom layer has a write-ahead log to persist logical updates.

Multi-versioning. Each write to the bottom layer creates a new logical version of the tree, in a copy-on-write manner. There are no in-place updates. This versioning is required as miners may submit multiple blocks concurrently that potentially conflict with each other; the bottom layer creates a new version for each write. Thus, writes never conflict with each other, and DSM-Tree does not require locking or additional coordination among miners or I/O-Helpers.

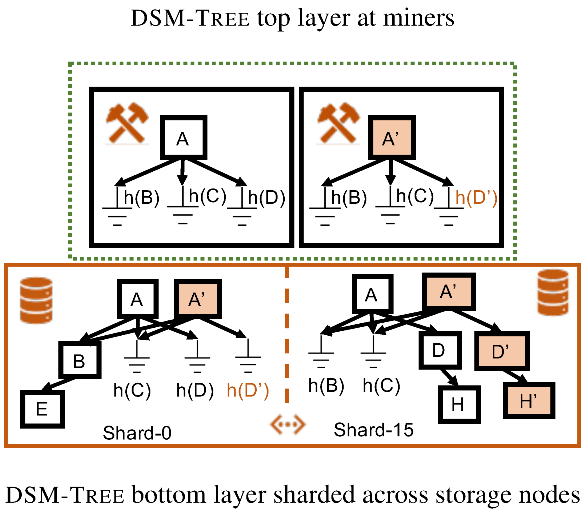

DSM-Tree top layer at miners
A
h(B)
h(C)
h(D)
A'
h(B)
h(C)
h(D')
A
A'
B
E
h(C)
h(D)
h(D')
Shard-0
A
A'
h(B)
h(C)
D
D'
H
H'
Shard-15
DSM-Tree bottom layer sharded across storage nodes

Figure 5: **DSM-Tree design in RainBlock**. This figure shows the two-layered DSM-Tree where miners have their private copy of the top layer for consistency and the bottom layer is sharded for scalability. h(C) is a hash node for C.

Sharding. Each subtree at the root node of the Merkle tree is a shard; root node in the modified Merkle Patricia Trie has 16 children, so DSM-Tree has 16 shards by default. Moreover, individual subtrees in each shard can further be partitioned, allowing shards to increase with the increasing system state.

Garbage collection. Garbage collection of versions is driven by the higher-level blockchain semantics. When multiple miners are working on competing forks of the blockchain, multiple versions are maintained. Eventually, one of the forks is confirmed and accepted as the mainline fork, and the others are discarded; associated versions from discarded blocks are garbage collected by the bottom layer of the DSM-Tree.

4.2 Top Layer

Given that the bottom layer maintains multiple versions across multiple shards, what data should be read by the miners? To resolve this question, miners use the top layer. Each miner has a private top layer, that represents a consistent, writeable snapshot of the system state; it contains the first few levels of the Merkle tree, till a configurable retention level (r), and has the Merkle root node that summarizes a consistent snapshot of the entire system state. When a miner executes transactions, all reads return values from this snapshot of the system.

As a miner executes txs, their top layer is updated, switching to another consistent view of the state as shown in Figure 5. The miner asynchronously updates the bottom layer's shards.

Caching and Pruning. The top layer acts as an in-memory cache of witnesses for the miner. By design, the top layer stores the recently used and the frequently changing parts of the Merkle tree. When the miner receives witness from I/O-Helpers, the top layer uses the witnesses to reconstruct a *partial* Merkle tree that allows miners to execute transactions, typically without performing I/O. The top layer also supports pruning the partial Merkle tree to help miners reclaim memory. Pruning replaces the nodes at the retention level $(r+1)$ with *Hash nodes*. Hash nodes are placeholders that help miners to identify the DSM-TREE shard which has the pruned nodes.

Witness Revision. The bottom layer of the DSM-TREE is updated asynchronously by miners. As a result, the top layer (miner) may receive stale witnesses from the bottom layer (via I/O-Helpers). We introduce a new technique termed *witness revision* to tolerate stale witnesses. A witness is determined to be stale or incorrect because the Merkle root in the witness doesn't match the top layer's Merkle root. However, this could happen because of an *unrelated update* to another part of the Merkle tree. For example, the top layer might contain one vertical path; a different vertical path might have been updated. The top layer detects when this happens, and *revises* the witness to make it current by applying updates to the witness that are known to the miner. If the Merkle root matches now, then the witness is accepted. Witness revision is similar to doing `git push` (trying to upload your changes), finding out something else in the repository has changed, doing a `git pull` (obtaining the changes in the repository) to merge changes, and then doing a `git push`. With witness revision, the top layer tolerates stale data from the bottom layer and allows miners to execute non-conflicting transactions that would otherwise get rejected. Note that, witness revision cannot revise every potential stale witnesses. If the top layers are pruned aggressively, they may have insufficient information to detect if the changes are from an unrelated part of the Merkle tree.

4.3 Synergy between the layers

The top and bottom layers collaborate to reduce network traffic. We also briefly discuss the potential DSM-TREE configurations with r (retention at the top layer) and c (compaction level at the bottom layer), and the tradeoffs involved.

Witness compaction. As the top layer of the DSM-TREE stores the top levels of the Merkle tree, the storage nodes need not send a full witness. Like the configurable retention level at the top layer r, the bottom layer has a configurable compaction level, c. If witnesses are larger than c, witnesses are compacted by removing the top few Merkle nodes, and are sent over the network. In addition, multiple witnesses are batched and Merkle nodes are deduplicated (node bagging), further reducing the network burden of transmitting witnesses.

Configurations. The DSM-TREE can be configured to operate entirely from local memory without any network overhead, or just from remote memory with high network utilization. For example, if the top layer of the DSM-TREE has $r=\infty$, then the top layer caches the entire Merkle tree and is fully served from local memory. Similarly, if the bottom layer has $c=n$ or $c=\infty$, then un-compacted witnesses are sent over the network and accessed entirely from remote memory, as shown in the Figure 4. These parameters (r and c) can be tuned based on the available memory and network capacity.

Tradeoffs. In a Merkle tree that has n levels, any DSM-TREE configuration that satisfies $c >= (n-r)$ allows the top layer to use compacted witnesses from the bottom layer. Note that having a higher r results in a lower number of transaction aborts, as the top layer has more information to detect non-conflicting updates and perform witness revision. The top layers should therefore set r based on the amount of memory available. Pruning the top layer should only be done under memory pressure.

4.4 Summary

The DSM-TREE is a novel variant of the Merkle tree, modified for faster transaction processing in public blockchains. DSM-TREE presents a new point in the design space of authenticated data structures. DSM-TREE highlights the benefits of in-memory pointer-based tree traversals in comparison to using node hashes and RocksDB lookups. The top layer exploits the cache-friendliness of the Merkle tree (top nodes are frequently read and updated), while the sharded bottom layer relies on the fact that witness creation only requires a vertical slice of the tree. DSM-TREE top layer uses witness revision to handle stale data from concurrent updates. While the DSM-TREE supports transactions and is exclusively used with RAINBLOCK in this paper, it can be easily modified to work with other blockchains and applications.

5 Implementation

We implement RAINBLOCK and DSM-TREE in Typescript, targeting node.js. Miners and storage nodes use the DSM-TREE library[1]. The performance critical portions of the code, such as `secp256kp1` key functions for signing transactions and generating `keccak` hashes, are written as C++ node.js bindings. To execute smart contracts, we implement bindings for the Ethereum Virtual Machine Connector interface (EVMC) and use Hera (v0.2.2). Hera can run contracts implemented using Ethereum flavored WebAssembly (ewasm) or EVMC1 bytecode through transcompilation. The I/O-Helper is implemented in C++. DSM-TREE and RAINBLOCK, together 15K lines of code, are open source and available on Github[2]. RAINBLOCK assumes 16 shards by default; this is configurable.

6 Evaluation

We seek to answer the following questions:

- What is the performance of a single miner? (§6.1)

[1] www.npmjs.com/package/@rainblock/merkle-patricia-tree

[2] https://github.com/RainBlock

System/optimization	Get	Put
In-memory Ethereum MPT	1x	1x
Pointer-based traversal	2.7×	2.3×
Batching and Lazy hash resolution	56×	69×
All optimizations	150×	160 ×

Table 2: **Performance breakdown**. The table shows the throughput of DSM-Tree (relative to in-memory Ethereum merkle tree) on gets and puts with different optimizations.

- What is the end-to-end performance of RainBlock in a geo-distributed setting? (§6.2)
- How is performance affected by tunable parameters? (§6.3)
- What are the overheads of RainBlock? (§6.4)

Our technical report [50] contains more details for these experiments, along with additional experimental results.

Experimental setup. We run the experiments in a cloud environment on instances which are similar to the `m4.2xlarge` instance available on Amazon EC2 with 32GB of RAM and 48 threads per node. We use Ubuntu 18.04.02 LTS, and node.js v11.14.0. For the end-to-end benchmarks, each storage node, miner, and I/O-Helper is deployed on its own instance.

Workloads. We evaluate the performance of RainBlock against synthetically generated workloads that mirror transactions on the Ethereum public *mainnet* blockchain. Since Ethereum transactions are signed, they cannot be used in experiments: we cannot change transaction data or the source accounts, because we do not have the `secp256k1` private key. To tackle this challenge, we analyze the public blockchain to extract salient features, and develop a *synthetic workload generator* which generates accounts with private keys we control so our I/O-Helpers can run and submit signed transactions.

Synthetic Workload Generator. We analyze the transactions in the Ethereum `mainnet` blockchain to build a synthetic workload generator. We analyzed 100K recent (since block 7M) and 100K older blocks (between blocks 4M and 5M) in the Ethereum blockchain to determine: 1) the distribution of accounts involved in transactions, and 2) the fraction of all transactions that invoke smart contracts. We observe that 10-15% of Ethereum transactions are contract calls and the rest are simple transactions. This is true of both recent blocks and older blocks. It is also the case that a small percentage of accounts are involved in most of the transactions. Based the analyzed data, we generate workloads where 90% of accounts are called 10% of the time, and 10% of the accounts are called 90% of the time. Smart contracts are invoked 15% of the time.

6.1 Performance of a single miner

The performance of a single miner depends on three things: the DSM-Tree data structure, the I/O-Helpers, and the top-layer cache at the miner. We first evaluate the performance of the DSM-Tree data structure, and then measure overall tx processing performance on a single node varying the number of I/O-Helpers. We then show the performance of the miner when varying the retention level of the top-layer cache.

DSM-Tree performance. For a fair comparison, we configure Ethereum to use an in-memory key-value store for storage. Ethereum uses the in-memory Merkle Patricia-Trie (MPT) [6] implemented using the `memdown` red-black tree [7]. We use `put` operations to recreate the system state corresponding to four million blocks on the Ethereum public chain. This results in 1.19M accounts. We read all accounts sequentially using `get` operations. `Put` operation creates or updates user account with 160-bit Ethereum address; `get` operation returns the RLP-encoded [9] Ethereum account at the address, along with its witness containing RLP-encoded Merkle nodes. We also compare the memory used.

Performance breakdown. DSM-Tree outperforms Ethereum's in-memory Merkle tree significantly in both gets and puts, as shown in Table 2. Note that both data structures are in memory, so the performance difference comes from other optimizations. Eth MPT has to use node hashes to traverse the Merkle Tree; in contrast, DSM-Tree uses pointers for constructing the tree, and eliminates hashing. This feature improves performance by 2.7×. Eth MPT has to hash and serialize, or deserialize each node in the Merkle tree while writing or reading them; DSM-Tree optimizes this with memoization and batching. Memoization allows remembering the hashes and RLP-encodings of unmodified Merkle nodes. Memoizing RLP-encoded nodes in DSM-Tree increases `get` performance by reducing redundant node de-serializations. Further, with batching, common nodes in the upper part of the Merkle Tree are only deserialized once, increasing the `get` performance by 56× relative to MPT, bringing the overall performance difference between DSM-Tree and Eth MPT to be 150×. A good overall intuition for these performance improvements is the difference between a linked list in memory versus a linked list where each node is serialized and stored in an in-memory key-value store using the node's hash as its key.

The performance difference for puts is similar (160×). The efficient memory representation of DSM-Tree contributes to 2.3× of this performance difference; the rest of the difference is due to lazy hash resolution. Lazy hash resolution defers recomputing the hashes of inner tree nodes until they are read. As a result, only the leaf nodes are updated in the critical path. If we force all the Merkle nodes to be updated and rehashed after every thousand updates, the performance difference with Eth MPT drops to 5× overall for puts.

Memory consumption. For the same system state with 1.19M accounts, DSM-Tree consumes 34× lower memory (775 MB) than Ethereum MPT (25 GB). This results from Eth MPT storing each node as a key-value pair. The reduced memory consumption of DSM-Tree is important since we

Optimizations	Config	Txs/s
Baseline	Ethereum, 1 miner	1K (1×)
RAINBLOCK	1 miner, 1 helper, r=0	2.6 K (2.6×)
Prefetch in parallel	1 miner; 4 helpers, r=0	7.7K (7×)
DSM-TREE tuning	1 miner; 4 helpers, r=7	27.4 K (27×)
Geo-distributed	4 miners; 16 helpers, r=8	20K (20×)

Table 3: **Performance breakdown**. The table shows the throughput of RAINBLOCK with different optimizations. All configs use 16 storage shards. Helpers indicate I/O-Helpers. While parallel prefetching increases RAINBLOCK throughput by 2.9× from 2.6K to 7.7K tps, the DSM-TREE top layer caching and witness compaction further increase throughput by 3.5×, from 7.7K to 27.4K tps.

want each shard to fit in the memory of a commodity server.

Overall performance. An Ethereum miner can process 1000 txs per second, if its system state is stored in an in-memory key-value store. We measured the performance of a single RAINBLOCK miner with one I/O-Helper when the top-layer caching is disabled; the miner is accessing system state from remote in-memory shards. In this setting, RAINBLOCK processes 2600 txs per second (2.6× higher than Ethereum).

Performance with multiple I/O-Helpers. Increasing the number of I/O-Helpers, increases the performance of RAINBLOCK, till up to four I/O-Helpers per miner. With four I/O-Helpers, parallel prefetching increases performance to 7700 txs per second (2.9× higher). Thus, RAINBLOCK miner (with four I/O-Helpers) outperforms an Ethereum miner by 7.7×.

Performance with DSM-TREE tuning. When we configure the top layer of the DSM-TREE to retain the first seven levels ($r = 7$, $c = n - 7$), the miner can process 27400 tps (3.5× higher than when top layer caching and witness compaction was disabled, and 27× higher than a single Ethereum miner).

6.2 End-to-End Geo-distributed Experiment

We run a geo-distributed experiment, with varying numbers of regions across three continents. Each region has four I/O-Helpers, one miner, and 16 storage nodes, caching eight levels of the DSM-TREE tree ($r = 8$, $c = n - 8$).

RAINBLOCK in a single region has a throughput of 25000 txs per second; this is slightly reduced from the 27400 tps in the previous section since the storage nodes are being accessed over a wide-area network. When we scale to four regions, the throughput drops to 20000 txs per second, thus retaining 80% of the single-region performance. When we ran a workload consisting purely of smart contracts (`OmiseGO Token`), RAINBLOCK achieved 17900 tps. Table 3 captures RAINBLOCK performance in various settings, from a single miner with one I/O-Helper to the geo-distributed experiment.

Tx confirmation latency remains the same as in Ethereum, as RAINBLOCK and Ethereum share the same block creation and confirmation logic (confirmed after ten blocks build on the block containing the tx). As more txs are present in each block, more txs are confirmed per second. These experiments use the same PoW consensus in Ethereum, thus demonstrating that RAINBLOCK achieves higher tx throughput without modifying the consensus protocol.

6.3 Perf impact of tunable parameters

We discuss the impact of varying two configuration parameters: the retention level r (number of Merkle tree levels stored at the top layer), and the compaction level c (bottom c levels of the witnesses are sent over the network by the shards).

Impact of tuning retention level. Increasing the retention level at miners increases overall tx throughput, but also increases the memory requirements at the miners. Pruning the cache to a certain retention level (r) helps reclaim the memory consumed by miners. For the Merkle tree constructed in the geo-distributed experiment described previously, miners caching till a tree depth of five ($r = 5$) consumes only 40% of the memory consumed by storing the full DSM-TREE in memory. As we increase the number of I/O-Helpers, the impact of higher r decreases. For example, in the geo-distributed experiment in a single region, there was no performance difference between $r = 7$ and $r = 8$ with four I/O-Helpers, but performance improved by 35% between $r = 7$ and $r = 8$ with two I/O-Helpers.

Retention Level and Tx Abort rate. If the top layer doesn't cache enough levels, stale witnesses cannot be revised, leading to tx aborts. We evaluate RAINBLOCK with 16 storage shards, 1 miner, and 4 clients, with various r and c configurations to measure the transaction abort rate. Increasing r reduces the transaction abort rate. Further, when there are a large number of accounts, the contention on Merkle tree nodes reduces, increasing the number of witnesses that can be revised and thus, reducing the abort rate for a fixed r. With 1M accounts and r=6, RAINBLOCK aborts less than two txs per second. In cases where the txs get aborted, I/O-Helpers or users themselves can fetch up-to-date witnesses from storage nodes and resubmit transactions to miners.

Tuning compaction level. A lower compaction level c increases the DSM-TREE shard throughput for I/O-Helpers (as it reduces the size of witnesses transmitted over network); with 10M accounts in state and $c = n$, shards process account reads at 1.36K ops/sec and with $c = n - 6$, they process 9.4K ops/sec (7× increase in throughput per shard). The compaction level should be tuned alongside the retention level, with $c >= (n - r)$ for a Merkle tree with n levels.

6.4 Overheads

RAINBLOCK has two main sources of overhead. First, it trades local storage I/O for network I/O, hence resulting in more network traffic. Second, it requires the participation of more commodity servers as I/O-Helpers and storage nodes. We discuss these overheads and how RAINBLOCK mitigates them.

Network bandwidth requirements. RAINBLOCK can be configured to produce blocks of a given size. For example, if the network can only handle 1 MB blocks, RAINBLOCK can be configured to produce blocks of this size. In our experiments, we do not constrain RAINBLOCK, and see that RAINBLOCK can pack about 240K transactions into each block (480× higher than Ethereum), on average, with the same proof-of-work consensus and block creation time. RAINBLOCK blocks are about 24MB in size, compared to the recent Ethereum blocks that are 40-60 KB. Our geo-distributed experiment used 24 MB blocks over the wide-area-network without running into network bottlenecks. The second source of network traffic is gossiping witnesses between miners. Using witness compaction, and node bagging (batching and deduplication), RAINBLOCK reduces witness sizes by 95%, allowing miners to advertise witnesses with commodity network bandwidths. Our witnesses sent over the network were a few KB in size.

Additional resources. RAINBLOCK requires I/O-Helpers and storage nodes. While storage nodes keep all state in memory, the state is sharded so that each shard fits in the DRAM of a commodity server. Storage nodes are shared among all miners, and hence they do not significantly increase the overall cluster requirements; I/O-Helpers can also be shared by miners.

7 Related Work

In this section, we place our contributions RAINBLOCK and DSM-TREE in the context of prior research.

Stateless Clients. The Stateless Clients [24] proposal seeks to insert witnesses into blocks, allowing miners to process a block without I/O. Despite active discussions [3, 25, 27], this proposal has not been implemented due to large witness sizes [57]; a single, simple transaction can have 4-6KB witness sizes, resulting in 40-60× the network overhead. In contrast, DSM-TREE reduces witness sizes by 95%; RAINBLOCK does not insert witnesses in blocks, and uses I/O-Helpers to reduce the I/O burden on miners.

Hyperledger Fabric. Fabric [17] is private while RAINBLOCK is public, resulting in significant differences. Fabric introduces a new execute-order-validate architecture where txs are executed only on a subset of servers. In RAINBLOCK all miners execute every transaction. While Fabric relies on signatures from trusted nodes (which can become the bottleneck), RAINBLOCK uses witnesses from untrusted servers to authenticate data. Peers in Fabric store the entire state, while RAINBLOCK storage nodes store partitions of the system state. RAINBLOCK improves performance with I/O-efficient transaction processing, while Fabric derives high performance from optimistic execution and efficient consensus.

Sharding. Sharding the blockchain into independent parallel chains that operate on subsets of state [36, 39, 41, 61, 62, 70] reduces I/O overheads; however, requires syncing the independent chains for consistency, is less resilient to failures or attacks [51, 55, 69], and require complex cross-shard transactions protocols. In contrast, RAINBLOCK does not shard the blockchain; the storage is sharded, but all miners add to a single chain. RAINBLOCK does not require locking or additional communication for executing transactions across multiple storage shards. Payment channels [4, 32, 35, 38, 40, 45] that offload work to side chains are complementary to our work.

Dynamic accumulators. Merkle trees belong to a general family of dynamic accumulators [20, 26]. Merkle trees, allow fast processing but, proofs grow with the underlying state. Constant-size dynamic accumulators based on RSA signatures [20, 26] have fixed size proofs but, have low processing rates; improving their performance is an ongoing effort [23]. DSM-TREE provides a practical solution to achieve high processing rates and small witness sizes. Recent work has proposed many new authenticated data structures [18, 28, 37, 53, 54, 58, 66, 68]. In contrast to these works, DSM-TREE scales Ethereum's Merkle Patricia trie [11] without changing its core structure, or how proofs are generated.

Transaction execution. RAINBLOCK adopts a design similar to Solar [71] and vCorfu [63], where transactions are executed based on data from sharded storage. RAINBLOCK modifies the design for decentralized applications and authenticated data structures. This allows RAINBLOCK to execute transactions on sharded state without requiring locking or additional coordination among miners. Similar to RAMCloud [48], the DSM-TREE design argues that large random-access data structures can get higher throughput and scalability when served from memory over the network.

8 Conclusion

We have presented RAINBLOCK, a public blockchain architecture that increases transaction throughput without changing the proof-of-work consensus protocol. RAINBLOCK achieves this by tackling the I/O bottleneck in transaction processing, allowing miners to pack more transactions into each block. RAINBLOCK introduces a novel architecture that moves I/O off the critical path, and the DSM-TREE, a new authenticated data structure that provides cheap access to system state. Please refer to our technical report [50] for more details about RAINBLOCK and the DSM-TREE. The RAINBLOCK prototype is publicly available at `https://github.com/RainBlock` and we welcome working with the community on its adoption.

Acknowledgements

We thank our shepherd, Abhinav Duggal, and the anonymous reviewers at ATC'21, VLDB'21, NSDI'20, and SOSP'19 for their insightful comments and suggestions. This work was supported by NSF CAREER #1751277, and donations from VMware, Google, and Facebook.

References

[1] Bitcoin. https://bitcoin.org/en/, 2019.

[2] Ethereum. https://github.com/ethereum/, 2019.

[3] Ethereum improvement proposals repository. https://github.com/ethereum/EIPs, 2019.

[4] Fast, cheap, scalable token transfers for ethereum. https://raiden.network/, 2019.

[5] Hybrid casper ffg. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1011.md, 2019.

[6] Implementation of the modified merkle patricia tree as specified in the Ethereum's yellow paper. https://github.com/ethereumjs/merkle-patricia-tree, 2019.

[7] In-memory abstract-leveldown store for node.js and browsers. https://github.com/Level/memdown, 2019.

[8] Parity ethereum 2.2.11-stable. https://github.com/paritytech/parity-ethereum/releases/tag/v2.2.11, 2019.

[9] Recursive Length Prefix Encoding. https://github.com/ethereum/wiki/wiki/RLP, 2019.

[10] RocksDB | A persistent key-value store. http://rocksdb.org, 2019.

[11] The modified Merkle Patricia tree. https://github.com/ethereum/wiki/wiki/Patricia-Tree, 2019.

[12] Full Node sync with Default Settings. https://etherscan.io/chartsync/chaindefault, 2020.

[13] Geth ethereum 1.9.25-stable. https://github.com/ethereum/go-ethereum/tree/v1.9.25, 2020.

[14] Ethereum average gas limit chart. https://etherscan.io/chart/gaslimit, 2021.

[15] Number of unique addresses in ethereum. https://etherscan.io/chart/address, 2021.

[16] Uncles per day. daily count of uncles generated by the ethereum network. https://www.etherchain.org/charts/unclesPerDay, 2021.

[17] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[18] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. {SPEICHER}: Securing lsm-based key-value stores using shielded execution. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 173–190, 2019.

[19] BCNext. The nxt cryptocurrency. https://nxt.org, November, 2013.

[20] Josh Benaloh and Michael De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[21] The block. Ethereum miners are increasing the network's gas limit by 25 = https://www.theblockcrypto.com/linked/69053/ethereum-miners-vote-for-25-gas-limit-increase,, June 20, 2020.

[22] bloXroute Labs. Increasing eth's gas limit: What we can safely do today. = https://ethresear.ch/t/increasing-eth-s-gas-limit-what-we-can-safely-do-today/8121,, October 2020.

[23] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[24] Vitalik Buterin. The Stateless Clients Concept. https://ethresear.ch/t/the-stateless-\client-concept/172, 2017.

[25] Vitalik Buterin. Detailed analysis of stateless client witness size, and gains from batching and multi-state roots. https://ethresear.ch/t/detailed-analysis-of-stateless-client-\witness-size-and-gains-from-batching-and-\multi-state-roots/862, 2019.

[26] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual International Cryptology Conference*, pages 61–76. Springer, 2002.

[27] Alexander Chepurnoy. A possible solution to stateless clients. https://ethresear.ch/t/a-possible-\solution-to-stateless-clients/4094, 2019.

[28] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. Edrax: A cryptocurrency with stateless transaction validation. 2018.

[29] Tonya M Evans. Cryptokitties, cryptography, and copyright. *AIPLA QUARTERLY JOURNAL*, 47(2):219, 2019.

[30] Ittay Eyal, Adem Efe Gencer, Emin Gun 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, Santa Clara, CA, 2016. USENIX Association.

[31] 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*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[32] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 473–489, 2017.

[33] Zane Huffman. CryptoKitties is Clogging the Ethereum Network. https://themerkle.com/cryptokitt\ies-is-clogging-the-ethereum-network/, 2019.

[34] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Communications and Multimedia Security*, 1999.

[35] Thaddeus Dryja Joseph Poon. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-\network-paper.pdf, 2019.

[36] Vitalik Buterin Joseph Poon. Plasma: Scalable autonomous smart contracts. https://plasma.io/plasma.pdf, 2019.

[37] Janakirama Kalidhindi, Alex Kazorian, Aneesh Khera, and Cibi Pari. Angela: A sparse, distributed, and highly concurrent merkle tree. 2018.

[38] Rami Khalil and Arthur Gervais. Revive: Rebalancing off-blockchain payment networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 439–453, 2017.

[39] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598, May 2018.

[40] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

[41] 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*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[42] Loi Luu, Yaron Velner, Jason Teutsch, and Prateek Saxena. Smartpool: Practical decentralized pooled mining. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1409–1426, Vancouver, BC, 2017. USENIX Association.

[43] Thomas McGhin, Kim-Kwang Raymond Choo, Charles Zhechao Liu, and Debiao He. Blockchain in healthcare applications: Research challenges and opportunities. *Journal of Network and Computer Applications*, 135:62–75, 2019.

[44] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[45] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *CoRR abs/1702.05812*, 306, 2017.

[46] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[47] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[48] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[49] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[50] Soujanya Ponnapalli, Aashaka Shah, Amy Tai, Souvik Banerjee, Vijay Chidambaram, Dahlia Malkhi, and Michael Wei. Rainblock: Faster transaction processing in public blockchains, 2020.

[51] Tayebeh Rajab, Mohammad Hossein Manshaei, Mohammad Dakhilalian, Murtuza Jadliwala, and Mohammad Ashiqur Rahman. On the feasibility of sybil attacks in shard-based permissionless blockchains. *arXiv preprint arXiv:2002.06531*, 2020.

[52] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[53] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making Authenticated Storage Faster in Ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, 2018. USENIX Association.

[54] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*, pages 376–392. Springer, 2017.

[55] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. *arXiv preprint arXiv:1901.11218*, 2019.

[56] Nick Szabo. Smart contracts. *Unpublished manuscript*, 1994.

[57] Peter Szilagyi. Are stateless clients a dead end? https://www.reddit.com/r/ethereum/comments/e8ujfy/are_stateless_clients_a_dead_end/, December, 10, 2019.

[58] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. Aggregatable subvector commitments for stateless cryptocurrencies. *IACR Cryptol. ePrint Arch.*, 2020:527, 2020.

[59] Vitalik Buterin. Toward a 12-second Block Time. https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/, 2014.

[60] Vitalik Buterin. Transaction spam attack: Next Steps. https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/, 2016.

[61] 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.

[62] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 95–112, Boston, MA, February 2019. USENIX Association.

[63] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 35–49, 2017.

[64] Jeffrey Wilcke. The Ethereum network is currently undergoing a DoS attack. https://ethereum.github.io/blog/2016/09/22/ethereum-network-currently-undergoing-\dos-attack/, 2016.

[65] JI Wong. Cryptokitties is causing ethereum network congestion (2017).

[66] Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*, pages 141–158, 2019.

[67] Lei Yang, Vivek Bagaria, Gerui Wang, Mohammad Alizadeh, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:1909.11261*, 2019.

[68] Mingchao Yu, Saeid Sahraei, Songze Li, Salman Avestimehr, Sreeram Kannan, and Pramod Viswanath. Coded merkle tree: Solving data availability attacks in blockchains. *arXiv preprint arXiv:1910.01247*, 2019.

[69] Jusik Yun, Yunyeong Goh, and Jong-Moon Chung. Trust-based shard distribution scheme for fault-tolerant shard blockchain networks. *IEEE Access*, 7:135164–135175, 2019.

[70] 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*, CCS '18, pages 931–948, New York, NY, USA, 2018. ACM.

[71] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. Solar: towards a shared-everything database on distributed log-structured storage. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 795–807, 2018.