BlockRaFT: A Distributed Framework for Fault-Tolerant and Scalable Blockchain Nodes

Manaswini Piduguralla
Indian Institute of Technology Hyderabad, India
cs20resch11007@iith.ac.in

Souvik Sarkar
Indian Institute of Technology Hyderabad, India
cs23mtech02001@iith.ac.in

Arunmoezhi Ramachandran
Independent Researcher, India
arunmoezhi@gmail.com

Sathya Peri
Indian Institute of Technology Hyderabad, India
sathya_p@cse.iith.ac.in

***Abstract*—Blockchain technology enhances transparency by maintaining a distributed ledger among mutually untrusting parties. Despite its advantages, scalability and availability remain critical bottlenecks that hinder widespread adoption. The increasing complexity of blockchain nodes further necessitates robust fault tolerance and high throughput to ensure seamless operations.**

We present BlockRaFT[1], a crash-tolerant distributed framework designed to improve both the scalability and reliability of blockchain node operations. BlockRaFT framework utilizes RAFT consensus protocol to elect a leader within a cluster of systems. The elected leader coordinates and distributes workloads across follower nodes, thereby optimizing resource utilization and work load balancing. We analyzed the tasks performed by blockchain nodes and partition them according to their stateful and stateless characteristics. Stateless operations are centralized at the leader, while stateful operations are replicated and coordinated across the cluster to ensure consistency and fault tolerance. We evaluate whether this distributed intra-node architecture provides measurable benefits over traditional single-node execution models in terms of scalability, availability, and performance.

Additionally, we introduce a concurrent Merkle tree optimization that decouples smart contract execution from tree updates, significantly reducing one of the significant performance overheads in blockchain systems. Our design philosophy is rooted in utilizing the well-established principles of distributed computing and customizing them for the blockchain domain rather than reinventing them.

***Index Terms*—Blockchain, Smart Contracts, Distributed system, crash tolerance, scalability**

I. Introduction

Satoshi Nakamoto introduced blockchain technology as the foundation for the cryptocurrency Bitcoin [1]. A decentralized ledger system that allows multiple parties to maintain a shared, immutable record of transactions without depending on a central authority. The system is built on well-established principles of cryptography. These cryptographic foundations ensure data integrity, enable secure transactions, and build trust in a network of untrusted participants. Initially developed for digital currencies, blockchain has evolved into a general-purpose distributed ledger technology. Its applications now extend far beyond cryptocurrency. Blockchain is being explored and adopted in various fields, including supply chain management, healthcare, real estate, identity management, and finance [2]. These use cases leverage blockchain's core features decentralization, transparency, immutability, and auditability.

A permissioned blockchain is a network with restricted access to approved participants [3]. One or more organizations govern these networks. They control who can join the network and what actions each participant can perform. In contrast, public blockchains such as Bitcoin or Ethereum are open to anyone and are maintained by decentralized communities. Permissioned blockchains offer several advantages for regulatory settings. Permissioned blockchains combine the blockchain framework's security, trust, and immutability properties with the necessary security and environmental control features that organizations need.

Drawbacks of Current Frameworks. Despite these benefits, scalability and availability remain the key challenges in permissioned blockchains. Blockchain networks often suffer from lower throughput and higher latency than traditional centralized systems [4]–[6]. Even in well-optimized systems, performance may lag by several orders of magnitude. Another limitation arises from the node architecture in permissioned settings. Organizations are typically limited in the number of nodes they operate. This is because the cost of running blockchain protocols increases with network size. Protocols such as gossip communication and consensus mechanisms become significantly more expensive as more nodes are added. Consensus protocols like Practical Byzantine Fault Tolerance (PBFT) [7] and Proof of Stake (PoS) [8] increase exponentially with an increase in the number of nodes.

Although the blockchain network is resilient to node failures, individual organizations may lose access if their nodes go offline. In such cases, the network continues to function. However, the affected organization cannot submit transactions, query the blockchain, or validate blocks until its nodes are restored, as illustrated in Fig. 1. Node crashes pose significant

This work is partly funded by Meity Project No.4(4)/2021-ITEA and a research grant from SUI foundation

[1]Code is available at: https://github.com/PDCRL/SCT-DistFramework.git

challenges in a permissioned blockchain. As blockchain adoption expands, researchers and developers are actively working on improving scalability, fault tolerance, and interoperability.

Contributions. This paper makes the following key contributions:

(a) We design a crash-tolerant and scalable distributed framework for permissioned blockchain nodes based on a RAFT leader-follower cluster model that eliminates single points of organizational failure in Section III.
(b) A three-phase concurrent Merkle tree optimization that decouples smart contract execution from state tree updates is introduced Section III.
(c) We provide the algorithmic design of the proposed framework (Section IV) and conduct extensive evaluations to validate its performance and effectiveness (Section V).

II. Background

A. Blockchain

Blockchain [1] is a distributed ledger system where changes to the ledger are made through transactions packed in blocks. Each block contains a set of transactions and is cryptographically linked to its predecessor through a hash of the previous block's contents. This chaining of blocks through hash ensures the integrity of the data, as altering a block's contents would require modifying all subsequent blocks. Blocks often contain multiple smart contract transactions (SCTs), which are self-executing pieces of code that execute the terms of an agreement between two or more parties. A network of nodes maintains this distributed ledger. Each node maintains a full or partial copy of the distributed ledger and participates in verifying and propagating new transactions and blocks. Agreement on a block being added to the chain is accomplished through various consensus mechanisms, like proof of work (PoW) [1], proof of stake (PoS) [8], and proof of elapsed time (PoET) [9].

B. Merkle Tree

A Merkle tree [10] is a cryptographic data structure widely used in blockchain systems to verify transactions and state information efficiently. In a Merkle tree, each leaf node holds the cryptographic hash of a piece of data (such as a transaction), while every non-leaf (internal) node contains the hash of the concatenated hashes of its child nodes. This hierarchical hashing structure means that even a single-bit change in any leaf node will propagate up the tree, altering the Merkle root at the top. As a result, the Merkle root serves as a compact and reliable fingerprint of all the data beneath it. By comparing just the Merkle roots, blockchain nodes can quickly and securely verify that they share the same data without exchanging the entire dataset.

C. RAFT Protocol

RAFT [11] is a crash-tolerant consensus algorithm developed as a more understandable alternative to Paxos [12]. It is designed to manage a replicated log across multiple nodes in a distributed system, ensuring consistency and fault tolerance. RAFT implements consensus through leader election, and the elected leader is responsible for the log management. Raft divides time into terms of arbitrary length, and each term begins with an election. A candidate wins an election if they receive votes from a majority of the voters in the same term. The votes are cast based on the first-come, first-served (FIFO) order. The majority rule ensures that at most one candidate can win the election for a particular term. We utilize the RAFT protocol for leader election in our proposed framework.

III. Proposed Framework

The proposed model introduces a distributed framework for blockchain nodes to improve two critical aspects of blockchain networks: scalability and fault tolerance. Blockchain networks suffer from lower throughput and higher latency compared to traditional centralized systems. Although blockchain networks are inherently designed to tolerate crashes and Byzantine faults at the network level, the reliability of individual nodes remains a concern. These issues must be addressed before blockchain networks become feasible alternatives for large-scale applications.

Denial of service due to node crashes are particularly pronounced in permissioned blockchain networks, where each node represents an organization. In such scenarios, the failure of an organization's node can result in the organization's operations being halted (Fig. 1). A simple solution would be for an organization to maintain multiple nodes representing itself in the network. This approach has two significant drawbacks. Increasing the number of nodes in the network, increases the message complexity. And increase in message complexity directly impacts the overall performance of the blockchain network. Secondly, this approach compromises the fairness of blockchains. If one organization maintains more nodes than others, it gains disproportionate representation in the consensus process. Therefore, adding more full nodes per organization is not a viable or fair solution.

A. BlockRaft Design

BlockRaft framework is designed such that it would not impact the network message complexity or skew the fairness. As illustrated in Fig. 1, a cluster of systems represents the blockchain node (node 2) in the network, and even if one or more nodes crash, the blockchain node still performs the operations. Our proposed BlockRaFT architecture is a leader-follower approach, and the node acting as a leader will represent the network. All the nodes are identical, and a leader is elected using the RAFT consensus. If a follower node crashes, the leader will redistribute the crashed follower's work to another node, and if the leader crashes, another node is elected as the new leader, and it starts performing the leader's duties. As we use RAFT consensus for leader election, the tolerance to crashes is $\lfloor (n-1)/2 \rfloor$ when n nodes exist in the cluster. In our approach to the blockchain network, each node still looks like a single, unified entity. This helps preserve the principle that every organization has equal representation. Inside the node, though, the workload is shared across multiple systems, which allows it to scale out without affecting its role

in the network. This framework improves performance and ensures that client access is not restricted if part of the node fails.

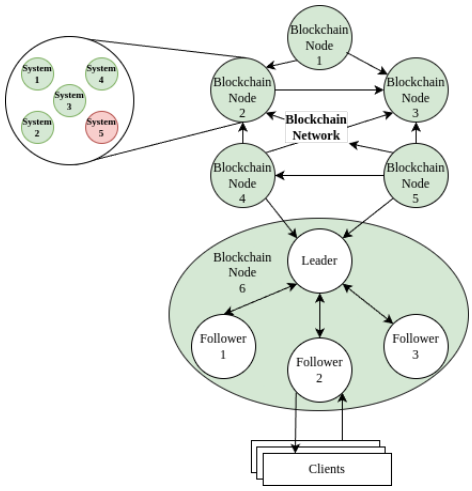


Fig. 1: Distributed Framework Design for the Blockchain Node.

Here, in Fig. 2 we present a detailed breakdown of the node architecture, and outline each module's specific roles and responsibilities in both leader and follower nodes. The architecture comprises several modules, like the network layer, Block Producer, DAG Module, Consensus Engine, and REST API. Each module is responsible for a distinct set of tasks, from managing internal state to interacting with clients. We group these modules into stateful and stateless categories. The leader handles all stateless operations, while stateful operations are maintained by every node, including the leader. The associated workload is shared across the cluster by the leader. The legend in Fig. 2 clarifies which components remain active under different roles, providing a comprehensive overview of system operations.

Node Module: The Node module is responsible for initializing and managing the role-based behavior of each node within the distributed blockchain framework. Each node participates in the RAFT consensus algorithm to elect a leader upon startup. Depending on its designated role, leader or follower, the node executes a corresponding set of tasks. Follower nodes periodically monitor the leader's status to see if the leader changes or crashes and wait for task assignments. Leader nodes initiate block production or validation based on the current blockchain state. Additionally, each node maintains a health monitoring thread to ensure reliability and responsiveness within the cluster.

Leader Module The Leader module coordinates the core blockchain operations. The leader node interacts with the network and represents the blockchain node to the network. During block production, the leader gathers pending transactions from the transaction pool, verifies them, and constructs a directed acyclic graph (DAG) of the dependencies among the transactions. It decomposes the DAG into independent components, assigns them to followers, and tracks their execution. The leader aggregates the resulting state changes, finalizes the block, and publishes it to the blockchain. In validation mode, the leader ensures that the computed state from worker nodes matches the expected final state embedded in the block.

Follower Module The Follower module executes transactional workloads delegated by the leader. Each follower reconstructs the DAG based on information received from the leader and executes its assigned components in parallel according to the DAG's topological ordering. Followers report completed execution states and synchronize changes across the cluster. They also respond to state commit or discard instructions following block validation.

Block Producer The Block Producer aggregates transactions from the shared transaction pool and constructs new blocks. The transaction pool is a shared distributed queue maintained among all nodes in the cluster. It performs essential preprocessing steps like transaction validation, signature verification, and expiration filtering. The resulting block is then forwarded to the Block Manager for further processing.

Block Manager The Block Manager acts as the central coordinator for block lifecycle management. It receives newly created blocks from the Block Producer and blocks from other nodes for validation. These blocks are then dispatched to the DAG Module for conflict analysis and decomposition. The Block Manager also interfaces with the Consensus Engine to confirm the consensus outcome and ensures that only validated blocks progress to final state computation and storage.

DAG Module

To facilitate efficient transaction execution, our framework incorporates a Directed Acyclic Graph (DAG) module, which operates in two distinct phases:

a) DAG Construction: The framework models inter-transaction dependencies using an adjacency matrix that captures read-write (RW), write-read (WR), and write-write (WW) relationships. Upon receiving a block of transactions, the DAG module employs multithreaded processing to analyze each transaction's read and write addresses. Dependencies are represented as directed edges in the adjacency matrix, where edges from transactions with lower IDs to those with higher IDs. This technique enforces acyclicity, a crucial property for maintaining deterministic execution order. The DAG module is inspired by the parallel execution model proposed in [6]. We have adapted the module for our distributed approach.

b) Component Detection: A Disjoint Set Union (DSU) [13] structure is used to detect connected components in the graph efficiently. Each node is initially placed in its own set, and the find function with path compression ensures that every node can quickly locate its root representative. When an edge is found between two nodes in the adjacency matrix, the unite operation merges their sets using union.

Parallel Scheduler The Parallel Scheduler is responsible for scheduling transaction execution based on the DAG structure provided by the leader. The scheduler identifies the transactions belonging to its assigned components for each follower and schedules them for parallel execution. An indegree array

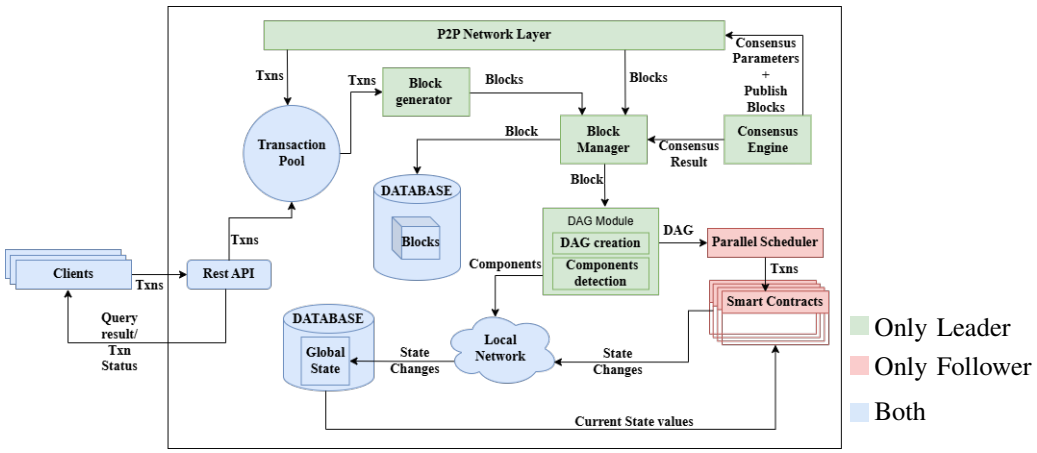


Fig. 2: Node Architecture in the Proposed Framework

is maintained to manage dependencies, recording the number of incoming edges (dependencies) for each transaction. The scheduler continuously searches for transactions with zero indegree, i.e., those with no unmet dependencies, and dispatches them for execution. After the transaction execution, the indegree values of all dependent transactions (i.e., those with outgoing edges from the completed transaction) are decremented. The scheduler ensures that transactions are scheduled in parallel, enabling efficient parallelism.

REST API The REST API is the gateway for client interaction with the blockchain system. It handles incoming requests for read-only queries and state-changing transactions (SCTs). Read-only requests are directly resolved from the global state, while SCTs are stored in the shared transaction pool for inclusion in future blocks.

Network Layer The Network Layer facilitates communication between the blockchain framework and external blockchain networks. It is responsible for relaying transactions to the transaction pool, forwarding blocks, and transmitting consensus information. The network layer is active in the node acting as the leader in the cluster.

Consensus Engine The Consensus Engine executes the chosen consensus protocol (e.g., Proof of Work, PBFT) to validate and approve new blocks. It ensures agreement among participating nodes on the correctness and ordering of blocks. The leader node participates in consensus on behalf of the cluster.

B. Concurrent Merkle Tree Optimization

The global state is stored in most blockchain systems using a Merkle tree. This data structure is used to ensure that any state changes can be verified consistently across all nodes in the network. The root of the Merkle tree, known as the Merkle root, is included in each block and serves as a cryptographic summary of the entire state. However, they naturally introduce update dependencies: modifying any leaf node requires updating all of its ancestors to the root. As a result, any two updates to the tree are inherently dependent, making it challenging to execute smart contracts in parallel while maintaining a consistent state. To address this bottleneck, we propose a three-phase optimization strategy:

1) Concurrent Execution with Hash Map: During transaction execution, instead of writing changes directly to the global state (i.e., the Merkle tree), we read the required state data but store any modifications in a concurrent hash map. This structure allows multiple threads to read from the global state while independently recording changes. Since no updates are made to the Merkle tree at this stage, parallel execution becomes feasible.

2) Parallel Application to Leaf Nodes: Once all transactions have been executed, each node in the cluster exchanges its recorded state changes with the others. The block's complete set of state updates is then applied in parallel, but only to the leaf nodes of the Merkle tree. Because each leaf node is modified exactly once (based on the final, post-transaction state), there is no contention, making parallel application efficient and safe.

3) Sequential Tree Recalculation: The modified leaf nodes' internal nodes (parents and ancestors) are updated in the final phase. This step is performed sequentially to compute the new Merkle root. This sequential recalculation is significantly less expensive than updating during transaction execution.

By decoupling transaction execution from Merkle tree updates and structuring the process into these three phases, we significantly reduce the overhead associated with Merkle tree operations.

4) Impact of Crashes: This subsection discusses the safety and crash-consistency implications of the proposed three-phase Merkle tree optimization.

a) Phase 1: Execution and Local State Recording.: During transaction execution, all transactions read state values from the committed global Merkle tree. Instead of directly updating the tree, modifications are recorded in a concurrent hash map local to each follower. The DAG module guarantees that no two transactions assigned to different followers operate on overlapping addresses within the same block. Consequently, transactions observe a consistent snapshot of the global state, and read-after-write hazards within a block are

avoided. Throughout Phase 1, the global Merkle tree remains unchanged and fully consistent.

b) Phase 2: Parallel Leaf Updates.: After all transactions in the block complete execution, each follower shares its recorded state changes with the leader. The leader aggregates these updates and distributes the combined state changes across the cluster. The leaf nodes corresponding to the modified keys are then updated in parallel. Because each key is updated exactly once based on the final execution result, no contention or inconsistency arises during this phase.

c) Phase 3: Parent Hash Recalculation.: Once all leaf updates are applied, the internal nodes of the Merkle tree are recomputed sequentially along the affected paths to produce the new Merkle root. The block is considered committed only after this recomputation completes successfully.

d) Leader Failures.: If the leader crashes at any point before Phase 3 completes, the block is not committed and the global state remains unchanged. Upon election of a new leader via RAFT, the cluster re-synchronizes and re-executes the block deterministically from the last committed state. Since intermediate updates are not externally visible, no inconsistent state is exposed.

e) Follower Failures.: If a follower crashes during Phase 1, the leader detects the failure via heartbeat monitoring and reassigns the corresponding transaction components to another active follower. If a follower crashes after sharing its local state changes, the aggregated updates are already preserved by the leader, and execution proceeds with the remaining quorum.

f) Block-Level Atomicity.: As in traditional sequential Merkle tree implementations, the block is treated as the atomic unit of state transition. The Merkle tree is not externally accessed during block execution; only the committed root after Phase 3 becomes visible. Therefore, the proposed optimization does not introduce new consistency windows compared to a sequential design. Instead, it reduces the duration of state recalculation while preserving atomicity and correctness guarantees.

IV. Algorithm Design

In the BlockRaft distributed framework, the Node Protocol (Algorithm 1) serves as the general lifecycle controller for each node in the cluster, determining whether a node acts as a leader or a follower at any given time. At initialization, each node participates in a consensus mechanism (Raft protocol) to elect a leader. Once elected, the leader node activates the Leader Protocol to coordinate transaction execution. All other nodes default to the Follower Protocol and await instructions and updates from the current leader.

The Leader Protocol (Algorithm 2) orchestrates the execution by collecting a new block of transactions from the network, generating a DAG from those transactions, and partitioning the DAG into independent components. These components represent a set of transactions that can be executed in parallel without conflicts. The leader assigns these components to available follower nodes based on the current cluster status, ensuring a quorum of active followers is available before proceeding. After broadcasting the component assignments and a start signal, the leader monitors the progress and completion of execution, eventually collecting state updates from the followers.

Algorithm 1: Node Protocol

```
   // Initialize leader and follower objects
1  leader ← leaderObj, follower ← followerObj
   // Start threads to monitor cluster health
2  Monitor ← startThread(clusterMonitorFunc)
   // Run Raft protocol until successful
3  while not raftSuccess() do
4      runRaftProtocol()
   // Execute node protocol as long as the cluster is healthy
5  while clusterHealth.load() do
6      if IsLeader() then
           // Start the recurring lease on the Leader key
7          leaderLease ←
             startThread(leaderHeartbeat)
           // Run leader protocol to produce or validate blocks
8          leader.leaderProtocol()
9      else
           // Start a thread to monitor the status of leader
10         leaderMonitor ←
             startThread(leaderMonitorFunc)
           // Run follower protocol
11         follower.followerProtocol()
   // Wait for monitor thread to complete
12 clusterMonitorFunc.join()
13 if IsLeader() then
       // Wait for leader lease thread if node is a leader
14     leaderHeartbeat.join()
15 return
```

On the other hand, the Follower Protocol (Algorithm 3) runs on each follower node and acts upon the data received from the leader. Each follower retrieves the current block and component assignments on startup and waits until the leader instructs them to begin execution. Once the "start" signal is received, followers execute their assigned components using a local scheduler object, which manages the parallel transaction execution. Meanwhile, they also monitor for new component assignments, allowing dynamic reassignment or load balancing. After completing their work, followers notify the leader and wait for a "finish" instruction, indicating either global completion or more work to be assigned.

A. Crash Handling

Fault tolerance in the BlockRaFT framework is built on the RAFT leader election algorithm. When a node starts, it immediately participates in RAFT to elect a leader. Once the leader is chosen, all nodes employ a heartbeat mechanism to signal their liveness. A leader's heartbeat confirms its authority for the current term. Within the leader module, a dedicated thread continuously monitors follower heartbeats, while in each follower, a thread tracks the leader's activity.

If the leader detects a follower crash and the number of failed nodes remains less than $(n-1)/2$, the work assigned to

Algorithm 2: Leader Protocol

```
// Initialize DAG module
DAGmodule ← DAGObj
// Get the latest block for validation from the network layer
latestBlock ← NetworkLayer.getBlock()
// Get the total member list of the cluster
n ← getClusterMembers()
// Get the list of active followers currently
k ← getActiveFollowers()
if k < ⌊n/2⌋ + 1 then
    return
// Create DAG from the transactions in the block
DAG ← DAGmodule.create(latestBlock)
// Derive the independent component list from the DAG
componentList ←
 DAGmodule.connectedComponents(DAG)
// Assign followers for each component in the list
assignFollowers(componentList)
// Start a thread to monitor followers' status
followersMonitor ←
 startThread(monitorFollowers)
// Share the block and components list with followers
sendFollowers(latestBlock)
sendFollowers(componentList)
// Inform the followers to start execution
sendFollowers("start")
// Check periodically the component execution status
checkComponents(componentList)
// Inform the followers that execution is complete
sendFollowers("finish")
// Collect state changes from the followers
saveData(memberList)
// Wait for threads to join and clean the objects
followersMonitor.join()
DAGmodule.dagClean()
componentList.clean()
return
```

Algorithm 3: Follower Protocol

```
// Initialize scheduler object
schedulerObj ← scheduler()
// Start a thread to watch for leader crashes
leaderMonitor ← startThread(monitorLeader)
// Get the block and components list from the leader
latestBlock ← receive(block)
componentList ← receive(components)
// Wait for leader's instruction to start execution
while receive(runKey) ≠ "start" do
    /* wait */
// Start monitoring for new component assignments
receive(watchComponentChanges)
// Execute transactions of assigned components
schedulerObj.execute(componentList)
// Inform leader about execution completion
sendLeader(componentID.status, ``done")
// Wait for leader's final instruction
while receive(runKey) ≠ "finish" do
    /* check for new work or wait */
// Collect state changes from other followers
sendData(ClusterList)
// Join monitor thread and cleanup
leaderMonitor.join()
schedulerObj.clean()
return
```

the crashed follower is reassigned to another follower, ensuring continuity. However, if crashes exceed $(n-1)/2$, the system halts all operations to preserve consistency. In case a follower detects the leader's failure, the cluster immediately triggers a new RAFT leader election to restore coordination.

B. Concurrent Merkle Tree

Our proposed algorithm aims to enable high-throughput transaction execution in blockchain systems while preserving the consistency guarantees provided by Merkle trees. Algorithm 4 presents the design of our concurrent Merkle tree operations. Instead of updating the tree directly during execution, the algorithm separates the process into distinct phases. This separation eliminates redundant work and allows safe parallelism. We have `updateValue` function, which records modifications in a concurrent hash map rather than directly updating the Merkle tree. By acquiring an accessor and writing the new value into `myMap`, multiple threads can apply updates concurrently without introducing conflicts. This ensures that transaction execution is decoupled from immediate tree modification, enabling parallelism. The retrieval operations `getRootHash`, `getValue`, and `getNode` provide efficient ways to query the state. The `getRootHash` function extracts the current Merkle root from persistent storage, while `getValue` and `getNode` retrieve values or deserialized nodes from the database. These lightweight functions do not interfere with concurrent updates, ensuring that read operations can occur alongside transaction execution.

The `parallelInsertFromMap` procedure implements the second phase of the optimization strategy. After execution, all recorded key–value pairs in the concurrent hash map are distributed across multiple threads in balanced chunks. Each thread inserts its assigned updates into the leaf nodes of the Merkle tree in parallel. Since every leaf node is updated exactly once based on the final state, this step avoids contention and maximizes throughput. Finally, once all leaf updates are complete, the algorithm performs a sequential recomputation of the Merkle tree's internal nodes using the `updateParentHashes` procedure. The ancestors up to the root are updated for each modified key to reflect the new leaf values. This ensures the correctness of the Merkle root while keeping the recomputation cost low because only affected paths are traversed. The `updateParentHashes` and `getRootHash` functions are implemented in the same manner as in a serial Merkle tree; therefore, we omit their details from the algorithm due to space constraints.

V. Experimental Results

Experimental Setup

Operating System: Ubuntu (64-bit) **Processor:**

- Model: AMD EPYC 7452 32-Core Processor
- Architecture: x86_64

Algorithm 4: ConcurrentMerkleTree Operations

```
Data Structure: myMap ← concurrent hash map
Function updateValue(key, value):
    Acquire accessor acc on myMap
    Insert (key) into myMap with acc
    acc.second ← value
Function getValue(key):
    keyHash ← computeHash(key)
    node ← getNode(keyHash)
    return node.value
Function getNode(key):
    data ← read from database using key
    if data exists then
        return data
    return empty Node
Function parallelInsertFromMap(nThreads):
    items ← all key-value iterators from myMap
    totalItems ← size of items
    chunkSize ← ⌈totalItems/nThreads⌉
    for i ← 0 to nThreads − 1 do
        start ← i × chunkSize
        end ← min(start + chunkSize, totalItems)
        Launch thread
        for j ← start to end − 1 do
            (key, value) ← items[j]
            insert(key, value)
    Join all threads
    foreach (key) ∈ myMap do
        updateParentHashes(key)
```

- CPU Count: 128 logical CPUs
- Memory: 251 GB, Buffer/Cache: 7.2 GB
- Base Frequency: 1.5 GHz, Max Boost: 2.35 GHz

Individual node configuration Operating System: Ubuntu (64-bit) **Processor:**

- CPU Count: 16 logical CPUs
- Memory: 9.7 GB, Buffer/Cache: 3.4 GB

We implemented the proposed BlockRaFT framework in C++, leveraging ETCD [14] as a distributed asynchronous shared memory system for communication and coordination among cluster nodes. ETCD facilitates consistent state exchange and simplifies coordination, while its built-in RAFT protocol supports leader election. In our implementation, we piggybacked on the ETCD leader for efficient cluster management. In addition to the ETCD-based shared-memory communication model, we also implemented a message-passing–based communication model, whose results are presented in Section V and Fig. 4a. We observed that both communication methodologies perform very similarly, with negligible differences in convergence behavior. We chose ETCD for the final system due to its practical performance and ease of integration.

Concurrent Merkle Tree Experiments

a) Baseline Scope and Design Rationale.: In our evaluation, the comparison is conducted against a traditional sequential Merkle tree implementation integrated within the same execution framework. This baseline performs updates incrementally during transaction execution, recomputing affected hashes immediately after each modification, which reflects the conventional update pattern used in many blockchain implementations.

We note that several advanced Merkle tree designs, such as Jellyfish [15] and Angela [16] exsists but these focus on improving persistence efficiency and versioning. These approaches address important challenges in blockchain state management and are highly effective. Our approach differs as we try to take advantage of batch based updates in blockchain and optimizing through creating a separation between transaction execution and state storage.

RQ 1: *Does the proposed concurrent Merkle tree strategy significantly reduce state update overhead compared to a sequential implementation?*

Our objective is to measure how the optimization behaves under varying workload compositions, transaction volumes, and levels of parallelism. We conduct three sets of experiments: (i) varying the read–write ratio from 0% to 100% in increments of 20% Fig. 3a, (ii) varying the total number of operations from 20K to 100K in increments of 20K Fig. 3b, and (iii) varying the number of threads from 2 to 64 in powers of two Fig. 3c

We employed the YCSB benchmark [17] suite using BenchBase repository [18]. Following prior work [19], we employ a dual-y-axis plot sharing a common x-axis, enabling an efficient and space-saving comparison between the serial and parallel results.

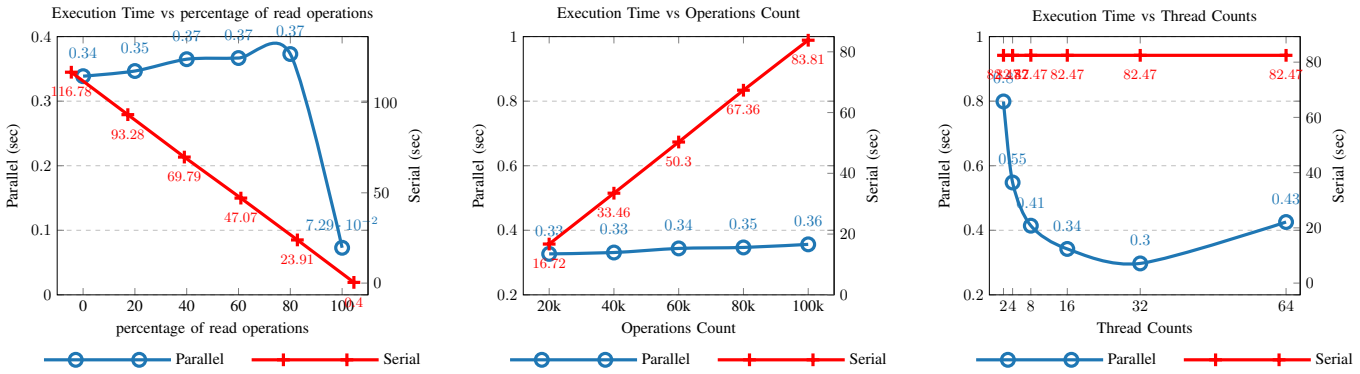
Fig. 3a evaluates how workload composition influences performance. As expected, write-intensive workloads benefit most from the proposed design, since Merkle tree updates dominate execution cost in these scenarios. At 100% writes, the concurrent implementation significantly outperforms the sequential baseline. Even in read-only workloads, where state updates are minimal, the design maintains approximately a 5× improvement due to parallel execution.

Fig. 3b measures scalability as the number of operations increases. The concurrent Merkle tree exhibits near-linear scaling, while the sequential baseline grows proportionally with workload size. At 100K operations, the optimized version consistently outperforms the serial implementation by more than two orders of magnitude.

Fig. 3c evaluates parallel scalability. Execution time decreases substantially as the thread count increases up to 32 threads, indicating effective parallelization of leaf updates and state recording. At 64 threads, a slight slowdown is observed due to synchronization overhead and resource contention.

Across all experiments, the concurrent Merkle tree consistently outperforms the sequential implementation, particularly under write-heavy and large-scale workloads. The results validate that separating transaction execution from Merkle tree updates substantially reduces state update overhead while preserving correctness.

A. BlockRaft Experiments



(a) Read/Write Percentage (b) Operations Count (c) Thread Count

Fig. 3: Performance comparison of Serial and Parallel execution under different workloads.

a) Comaprision models.: To evaluate the performance of our proposed BlockRaft framework, we implemented two distinct comparison models within our system architecture. All configurations share identical smart contract logic, DAG construction, storage mechanisms, and Merkle tree structures to ensure that performance differences arise solely from architectural design decisions.

1) **Single-core Execution Model:** This model represents the traditional blockchain processing approach. Transactions are executed sequentially on a single processing core. This model serves as a baseline for understanding the performance of conventional blockchain systems without any parallelism or workload distribution.
2) **Multi-core Execution Model:** This configuration executes transactions in parallel across multiple processing cores in a single system. However, unlike BlockRaft, this model does not incorporate sophisticated workload distribution or coordination mechanisms. The parallel execution benefits from concurrency and Merkle tree optimization but lacks a fault-tolerant design. This baseline isolates the benefits of intra-node distribution and fault tolerance by comparing BlockRaFT against a purely shared-memory parallel execution model.

These baselines are intentionally implemented within the same framework to ensure controlled and fair comparison. Our goal is quantify the benefits of distributed intra-node clustering and evaluate the overhead of crash-tolerant coordination.

RQ 2: Scalability *Does BlockRaFT maintain near-linear execution time scaling under varying conditions, and does it outperform the baseline models?*

RQ 3: Fault Tolerance *Is the performance overhead of BlockRaFT's distributed coordination is a reasonable trade-off for the fault-tolerance guarantees provided?*

We evaluate the performance of BlockRaFT using two smart contracts: *Voting* and *Wallet*. Our objective is to analyze how execution time is affected by key parameters, including the number of transactions per block, degree of dependency, thread count, execution delay, cluster size, and fault scenarios.

The degree of dependency is defined as the percentage of actual directed edges in the transaction dependency graph (DAG) relative to the total number of possible edges. This metric quantifies the level of interdependence among transactions within a block. A higher percentage indicates a denser dependency graph, which restricts parallelism and increases coordination overhead.

For each experiment, we compare BlockRaFT against multi-core and single-core baselines to assess scalability and distributed overhead. The Voting contract models a decentralized voting system that supports voter and candidate registration, vote casting, vote transfers, and secure result queries. Additional results for the Wallet contract are available in our repository [20].

Impact of Thread Count: Execution time was measured using 2, 4, 8, 16, 32, and 64 threads with blocks of 4000 transactions. As shown in Fig. 4c, increasing the thread count initially improves performance. However, the rate of improvement decreases as the number of threads increases, indicating diminishing returns due to synchronization and coordination overhead. At 64 threads, execution time increases for both BlockRaFT and the multi-core baseline, suggesting contention and parallelization limits.

Impact of Conflict Percentage: We evaluated performance under transaction dependency levels ranging from 0% to 5% using blocks containing 4000 transactions (Fig. 4a). At 0% dependency, performance is not optimal despite the absence of logical conflicts. This behavior results from the large number of unique addresses, which increases inter-node data transfer and exposes communication as a bottleneck. This trend is further illustrated in the breakdown analysis shown in Fig. 6.

The BlockRaFT-Msg configuration replaces ETCD with direct message passing for sharing state updates. Although this reduces coordination overhead, communication remains a limiting factor. In future work, we plan to reduce this communication bottleneck to further enhance scalability.

Impact of Transactions per Block: We varied the number of transactions per block from 1000 to 5000 under low-conflict conditions. As shown in Fig. 4b, execution time increases approximately linearly with transaction volume across all

configurations. The single-core baseline exhibits the highest growth rate and longest execution times. In contrast, multi-core and multi-node configurations scale more efficiently, particularly under heavier workloads. Although the multi-node setup introduces coordination overhead at lower workloads, this overhead remains modest while providing the additional benefit of crash fault tolerance.

Based on our experimental evaluation, both research questions are positively addressed. For RQ2 (Scalability), the results demonstrate that BlockRaFT maintains near-linear growth in execution time as transaction volume increases and achieves substantial performance improvements over the baseline models. While diminishing returns appear at higher thread counts due to synchronization overhead, the framework scales efficiently under varying workload and conflict conditions.

For RQ3 (Fault Tolerance), the additional overhead introduced by distributed coordination remains moderate, particularly at higher workloads where multi-node execution outperforms the single-core setup. This confirms that the performance cost of coordination is a reasonable trade-off for the crash fault-tolerance guarantees provided by BlockRaFT.

b) Impact of Node Failures: To evaluate fault tolerance, we introduced 1, 2, and 3 node failures in the distributed cluster, maintaining 16 threads and 5% conflict percentage in the voting smart contract.

RQ 4: *Does BlockRaFT maintain acceptable performance under node failures?*

As shown in Fig. 5, execution time increases after the first failure due to workload increase per system. However, the system continues to operate correctly as long as quorum is maintained. Notably, the performance impact of additional failures beyond the first is comparatively small, indicating that the system stabilizes after the initial redistribution overhead.

These results demonstrate that BlockRaFT maintains functional correctness and exhibits graceful degradation under crash scenarios.

c) Bottleneck Analysis: **RQ 5:** *What are the bottlenecks in the current distributed framework?*

As shown in Fig. 6, the total execution time increases sharply with the number of transactions per block. The main bottlenecks of the current distributed framework are the execution phase and the component detection mechanism, especially under high transaction loads. In future work, we plan to further optimize the execution layer and improve the efficiency of the component detection process to enhance scalability under high transaction loads. The values for breakdown plots and extended experiments are available in the repo [20]

VI. Related Work

Performance limitations remain one of the key obstacles preventing blockchain technologies from being adopted at scale [4]. In response, a variety of research for enhancement techniques have been well explored mainly: parallelized transaction processing, sharding mechanisms, DAG-based Blockchains and distributed transaction execution.

Parallel execution of smartcontracts in Blockchains comes with its own challenges apart from traditions parallel trasaction execution in traditional databases. This avenue for improving throughtput has been well explored in literatures and extensively studied [6], [21], [22]. ParBlockchain [23] precomputes a transaction dependency graph and executes only independent transactions in parallel deterministically. Block-STM [24] similarly accelerates smart-contract execution through optimistic concurrency control with versioned state validation.

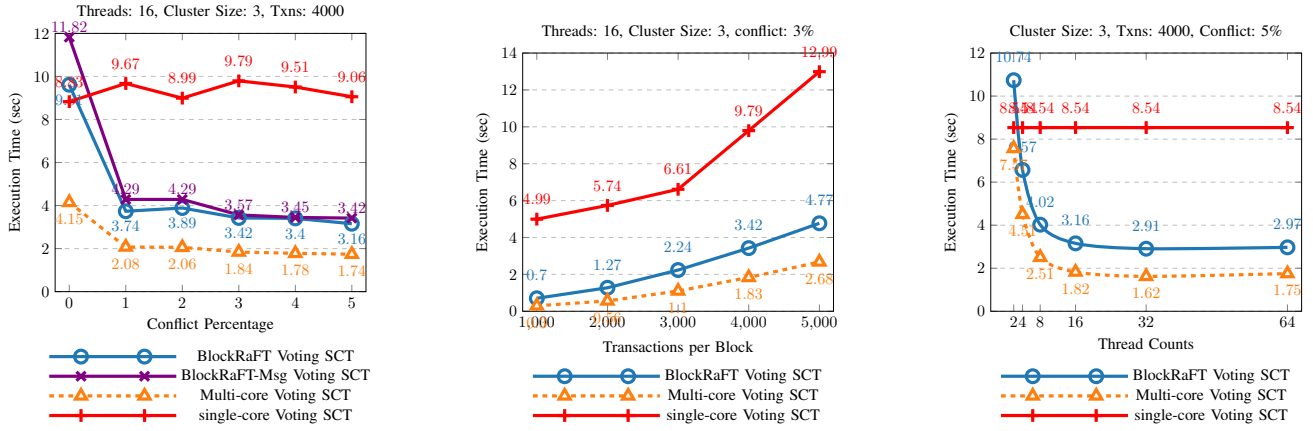
Several blockchain architectures have adopted directed acyclic graph (DAG) structures to address these limitations and replace the conventional chain-based model [25]. OTA's Tanglev [26] treats each transaction as a node in a DAG, requiring new transactions to approve two previous transactions. PHANTOM [27] extends the proof of work [1] consensus to a blockDAG, distinguishes between blocks mined properly by honest nodes and those created by non-cooperating nodes who choose to deviate from the mining protocol. Adopting DAG-based blockchain systems also implies that scalability is no longer constrained by block size, shifting the focus toward optimizing node performance and network-level concurrency to handle increased transaction loads effectively. Beyond sharding [28], [29] and parallelism, researchers have explored distributed blockchain architectures that improve throughput through workload distribution.

Feature / System	DiPETrans	PilotFish	BlockRaft	Sharding
Scalable	✓	✓	✓	✓
Node Crash Tolerance	✗	✗	✓	✗
Workload Distribution	✗	✗	✓	✗
Distributed SCTs Execution	✗	✓	✓	✓
Parallel Mining	✓	✗	✗	✗
Trustless Operation	✗	✓	✓	✓
Compatibility	✓	✓	✓	✗
Blockchain Security Impact	✗	✓	✓	✓

TABLE I: Comparison of DiPETrans [30], PilotFish [31], BlockRaft, and general sharding-based [29], [32] architectures

DiPETrans [30] presents a framework for parallelizing transaction execution within a block by leveraging a community of peers. Using static analysis, a leader node groups independent transactions into shards and assigns them to follower nodes for parallel execution. It employs a transaction ordering service to establish a global sequence of transactions and delegates execution to distributed worker nodes. Both mining and validation are parallelized, leveraging community compute power.

PilotFish [31] introduces a three-layer system architecture for a distributed execution engine consisting of a global transaction sequencer, a set of execution workers, and distributed storage. Transactions are processed across execution nodes in a pipelined fashion, and execution consensus maintains consistency without requiring all nodes to re-execute every transaction. This architecture reduces redundant computation and supports high throughput in permissioned blockchain



(a) Conflict Percentage (b) Transactions Count (c) Thread Count

Fig. 4: Performance comparison of Serial, Parallel and distributed blockchain frameworks under different workloads.

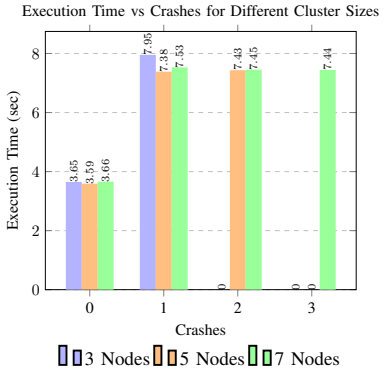


Fig. 5: Execution Time Varying cluster size with crashes

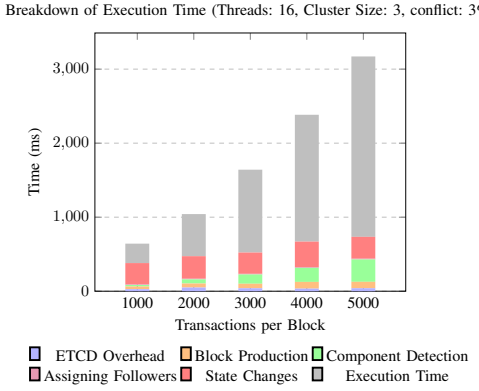


Fig. 6: Breakdown of Execution Time

settings. Pilotfish demonstrated linear scaling of an eightfold throughput increase with eight machines, while maintaining low latency and avoiding the batching delays.

In this work, we concentrate on workload division for not just execution but also all blockchain node operations. Improving the execution is insufficient to improve blockchain scalability because one should look at all the operations blockchain nodes perform. We also explore crash tolerance to decrease the load on the blockchain network caused by the replication solution for crash tolerance. The comparison in Table I highlights key differences among DiPETrans, PilotFish, BlockRaft, and general sharding-based architectures. While all systems demonstrate scalability, their levels of resilience and decentralization vary notably. BlockRaft is the only architecture that provides strong crash tolerance, whereas DiPETrans, PilotFish, and common sharding designs do not inherently tolerate node failures. BlockRaft also uniquely distributes the full blockchain workload, unlike the others. Parallel mining is exclusive to DiPETrans, as the remaining systems are not oriented around mining tasks. Trust models further differentiate the systems: PilotFish, BlockRaft, and sharding operate trustlessly, while DiPETrans relies on a trusted community. BlockRaft also integrating seamlessly into existing systems, whereas sharding typically requires architectural changes.

VII. Conclusion

In this paper, we presented BlockRaFT, a crash-tolerant and scalable distributed framework for blockchain nodes. Our approach uses a leader-follower model to distribute workloads efficiently and applies storage optimizations to improve smart contract performance. Additionally, we incorporate data storage optimizations that are particularly effective for handling smart contract operations, further enhancing system performance. We implemented and evaluated our framework through a series of experiments, demonstrating that BlockRaFT consistently outperforms single-core implementations in terms of throughput and responsiveness. BlockRaFT introduces only a small overhead compared to multicore setups, which is a reasonable trade-off considering its improved crash tolerance and distributed resilience. We further propose a Merkle tree optimization that decouples database updates from smart contract execution.

As part of our future work, we aim to optimize BlockRaFT's performance by introducing a data sharing mechanism. This is motivated by our observation of overhead introduced by data sharing, particularly when operating at 0% conflict percentage. We also intend to explore a more decentralized, peer-to-peer architecture.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," https://bitcoin.org/bitcoin.pdf, 2008.

[2] U. Bodkhe, S. Tanwar, K. Parekh, P. Khanpara, S. Tyagi, N. Kumar, and M. Alazab, "Blockchain for industry 4.0: A comprehensive review," *IEEE Access*, vol. 8, pp. 79 764–79 800, 2020.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," ser. EuroSys, 2018.

[4] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding Concurrency to Smart Contracts," ser. PODC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 303–312.

[5] P. S. Anjana, H. Attiya, S. Kumari, S. Peri, and A. Somani, "Efficient Concurrent Execution of Smart Contracts in Blockchains Using Object-Based Transactional Memory," in *Networked Systems*, C. Georgiou and R. Majumdar, Eds. Cham: Springer International Publishing, 2021, pp. 77–93.

[6] M. Piduguralla, S. Chakraborty, P. S. Anjana, and S. Peri, "Dag-based efficient parallel scheduler for blockchains: Hyperledger sawtooth as a case study," in *Euro-Par 2023: Parallel Processing*, J. Cano, M. D. Dikaiakos, G. A. Papadopoulos, M. Pericàs, and R. Sakellariou, Eds. Cham: Springer Nature Switzerland, 2023, pp. 184–198.

[7] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 51–68. [Online]. Available: https://doi.org/10.1145/3132747.3132757

[8] P. Vasin, "Blackcoin's proof-of-stake protocol v2," *URL: https://blackcoin. co/blackcoin-pos-protocol-v2-whitepaper. pdf*, vol. 71, 2014.

[9] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten, "POET: Target-System Independent Visualizations of Complex Distributed-Applications Executions," *The Computer Journal*, vol. 40, no. 8, 1997.

[10] H. Liu, X. Luo, H. Liu, and X. Xia, "Merkle tree: A fundamental component of blockchains," in *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*, 2021, pp. 556–561.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[12] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[13] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, no. 2, p. 215–225, Apr. 1975. [Online]. Available: https://doi.org/10.1145/321879.321884

[14] T. etcd Authors, "etcd: A distributed, reliable key-value store for critical data," GitHub repository, 2025, accessed: 2025-02-19. [Online]. Available: https://github.com/etcd-io/etcd

[15] Z. Gao, Y. Hu, and Q. Wu, "Jellyfish merkle tree (2021)," 2021.

[16] J. Kalidhindi, A. Kazorian, A. Khera, and C. Pari, "Angela: A sparse, distributed, and highly concurrent merkle tree," *UC Berkeley, Berkeley*, 2018.

[17] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[18] cmu-db, "Benchbase: Multi-dbms sql benchmarking framework via jdbc," https://github.com/cmu-db/benchbase, 2025, accessed: 2025-08-09.

[19] K. Gao, C. Sun, S. Wang, D. Li, Y. Zhou, H. H. Liu, L. Zhu, and M. Zhang, "Buffer-based end-to-end request event monitoring in the cloud," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 829–843. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/gao-kaihui

[20] Anonymous. (n.d.) Blockraft. Anonymous code repository hosted on 4open.science. [Online]. Available: https://anonymous.4open.science/r/BlockRAFT-00C1

[21] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An Efficient Framework for Optimistic Concurrent Execution of Smart Contracts," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 83–92.

[22] Y. Hay and R. Friedman, "Batch-schedule-execute: On optimizing concurrent deterministic scheduling for blockchains," in *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*, 2024, pp. 163–174.

[23] M. J. Amiri, D. Agrawal, and A. El Abbadi, "ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1337–1347.

[24] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing," 2022.

[25] Q. Wang, J. Yu, S. Chen, and Y. Xiang, "Sok: Dag-based blockchain systems," *ACM Comput. Surv.*, vol. 55, no. 12, Mar. 2023. [Online]. Available: https://doi.org/10.1145/3576899

[26] H. Hellani, L. Sliman, A. E. Samhat, and E. Exposito, "Tangle the blockchain:towards connecting blockchain and dag," in *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2021, pp. 63–68.

[27] Y. Sompolinsky, S. Wyborski, and A. Zohar, "Phantom ghostdag: a scalable generalization of nakamoto consensus: September 2, 2021," in *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, ser. AFT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 57–70. [Online]. Available: https://doi.org/10.1145/3479722.3480990

[28] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards Scaling Blockchain Systems via Sharding," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 123–140.

[29] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 17–30.

[30] S. Baheti, P. S. Anjana, S. Peri, and Y. Simmhan, "DiPETrans: A framework for Distributed Parallel Execution of Transactions of Blocks in Blockchains," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 10, p. e6804, 2022.

[31] Q. Kniep, L. Kokoris-Kogias, A. Sonnino, I. Zablotchi, and N. Zhang, "Pilotfish: Distributed execution for scalable blockchains," in *Financial Cryptography and Data Security (FC)*, Miyakojima, Japan, Apr. 2025.

[32] P. Li, M. Song, M. Xing, Z. Xiao, Q. Ding, S. Guan, and J. Long, "Spring: Improving the throughput of sharding blockchain via deep reinforcement learning based state placement," ser. WWW '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2836–2846. [Online]. Available: https://doi.org/10.1145/3589334.3645386