A Hierarchical Sharded Blockchain Balancing Performance and Availability

Yongrae Jo and Chanik Park

Abstract—Blockchain networks offer decentralization, transparency, and immutability for managing critical data but encounter scalability problems as the number of network members and transaction issuers grows. Sharding is considered a promising solution to enhance blockchain scalability. However, most existing blockchain sharding techniques prioritize performance at the cost of availability (e.g., a failure in a few servers holding a shard leads to data unavailability). In this paper, we propose PyloChain, a hierarchical sharded blockchain that balances availability and performance. PyloChain consists of multiple lower-level local chains and one higher-level main chain. Each local chain speculatively executes local transactions to achieve high parallelism across multiple local chains. The main chain leverages a directed-acyclic-graph (DAG)-based mempool to guarantee local block availability and to enable efficient Byzantine Fault Tolerance (BFT) consensus to execute global (or cross-shard) transactions within a collocated sharding. PyloChain speculatively executes local transactions across multiple local chains to achieve high parallelism. In order to reduce the number of aborted local transactions, PyloChain applies a simple scheduling technique to handle global transactions in the main chain. PyloChain provides a fine-grained auditing mechanism to mitigate faulty higher-level members by externalizing main chain operations to lower-level local members. We implemented and evaluated PyloChain, demonstrating its performance scalability with 1.49x higher throughput and 2.63x faster latency compared to the state-of-the-art balanced hierarchical sharded blockchain.

Index Terms—Blockchain, Sharding, Availability, Performance

I. INTRODUCTION

Blockchain networks support decentralization, transparency, and immutability to manage critical data. However, as the number of network members and transaction-issuing users increases, it faces a scalability problem in efficiently handling a large volume of transactions. Sharding-based parallelism [1]–[5], [5]–[20] offers a promising solution by partitioning the blockchain to process transactions concurrently. Two primary schemes exist: availability-favored and performance-favored sharding. Availability-favored sharding [16]–[19] replicates a full copy of shards (i.e., collocated) across every server,

Fig. 1. Blockchain Sharding Schemes.

(a) Availability Sharding (b) Performance Sharding (c) Balanced Sharding
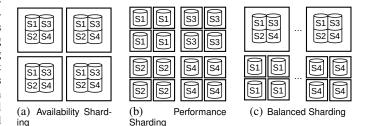
allowing parallel consensus among corresponding shards, ensuring data availability when a certain group of servers fails. Performance-favored sharding [1]–[12] replicates a single copy of a shard (i.e., isolated) to a disjoint group of servers, obviously enhancing performance by reducing storage, networking, and consensus overhead within each server.

However, the availability-favored sharding suffers from limited scalability because it requires all servers to hold a complete copy of the shards. This significantly burdens each member as the number of members and shards increases. Meanwhile, performance-favored sharding may pose risks to data availability because it inherently replicates data to fewer servers; For instance, certain disastrous scenarios—such as a single point of management failure caused by bad software updates [21], region-wide earthquakes [22], and data center fire [23]—can lead to data unavailability in a certain group of servers holding a specific shard. These constraints considerably hinder the practical use of sharded blockchains in real-world business operations. Thus, it is desirable to motivate the need for a balanced approach where the previous two approaches are combined to find a sweet spot between availability and performance. We depict those sharding schemes in Fig. 1

A hierarchical blockchain architecture with sharding zones presents an effective way to realize this balanced approach. By organizing the network into hierarchical sharded zones, each full member, which participates representatively in cross-zone communication, manages the higher-level main chain (i.e., a full copy of all shards). Meanwhile, local members within each zone manage the lower-level local chain (i.e., a single shard). This architecture ensures data availability even if a specific shard fails, while still benefiting from performance advantages of sharding-based parallelism. Additionally, a hierarchical sharded blockchain can efficiently process cross-shard transactions by exploiting collocated shards within full

We identify three key challenges in designing the hierarchical blockchain architecture: First, zone scalability; For data availability, lower-level local blocks need to be reliably propagated via cross-zone communications over an asynchronous network and integrated into the higher-level main chain. As the number of shards increases, large volumes of local blocks from multiple zones are generated concurrently, making the scalability of the main chain consensus crucial. Second, transaction scalability: As a very large volume of local blocks from many sharding zones is integrated into the main chain, efficiently processing these transactions becomes increasingly important. In particular, global (or cross-shard) transactions may abort local transactions that have been speculatively executed within a local chain, increasing synchronization overhead across the hierarchy. Third, malicious full member: Each full member plays a critical role in cross-zone communications, responsible for relaying essential data across the hierarchy. The main challenge arises from the fact that local members are unaware of what happens on the main chain, which malicious full members may exploit. Therefore, auditing the trustworthiness of full members is crucial for correct hierarchical operation.

In this paper, we propose PyloChain, a hierarchical sharded blockchain that balances availability and performance by addressing the key challenges. For the first challenge, PyloChain employs a DAG-based mempool [24] to append highly concurrent local blocks from multiple sharding zones simultaneously in a leaderless manner, ensuring local block availability as well as their efficient total ordering with high throughput. To the second challenge, PyloChain adopts and enhances the order-execute-order-validate (O-X-O-V) transaction processing model [20] by incorporating a simple scheduling technique. This model enables speculative parallel execution of local transactions across zones and batch-processing of cross-shard transactions at the end of each main block processing cycle to reduce aborted transactions, significantly reducing the burden on main block processing. For the third challenge, we design a fine-grained auditing mechanism that externalizes the full member's critical actions on the main chain, allowing them to be audited by local members. To achieve this, PyloChain divides the operational semantics of the DAG-based main chain protocol into fine-grained phases, enabling asynchronous audits to verify the full member's behavior at each phase.

We implement PyloChain in GoLang [25], using Hyperledger Fabric [26] as the baseline local chain within each sharding zone and Narwhal/Bullshark [27] as the higher-level DAG-based mempool with BFT consensus. Our evaluation of PyloChain demonstrates that it achieves practical performance across various workloads in a network with up to 18 zones, including 18 full members and 72 local members. We observed that PyloChain consistently demonstrates better performance scalability compared to the state-of-the-art balanced hierarchical sharded blockchain [20]. Specifically, under 20% global transactions and 12 zones, PyloChain achieved 1.49x higher throughput and 2.63x faster latency. Also, we show that PyloChain effectively balances performance and availability, by comparing the other sharding schemes.

In summary, our key contributions are:
- We propose PyloChain, a hierarchical sharded blockchain

balancing performance and availability.
- PyloChain leverages a DAG-based consensus on the main chain to ensure availability and high throughput, enhancing transaction scalability through parallel execution across local chains and a simple scheduling technique.
- PyloChain addresses the full member auditing problem by externalizing fine-grained semantics of main chain operations.
- We evaluate PyloChain to demonstrate its feasibility.

The rest of this paper is organized as follows. Section II provides background and related work. Section III describe the system overview of PyloChain. Section IV details PyloChain's design. Section V presents evaluation results of PyloChain. Section VI discusses PyloChain with future work, and Section VII concludes.

II. BACKGROUND AND RELATED WORK

A. Blockchain Sharding

Sharding is regarded as a promising solution to enhance blockchain scalability. For instance, one approach to sharding divides the entire blockchain network into multiple subnetworks, with each subnetwork managing only a portion of the complete blockchain ledger, referred to as a "shard." This division distributes the workload across multiple subnetworks, ultimately reducing consensus, network communication, and storage overhead in each smaller network with fewer members.

However, introducing sharding could reduce the availability of each shard, as each shard is managed by a smaller number of members. The failure of a single shard group could lead to the complete unavailability of that shard, causing severe damage to the system. For instance, this could pose a critical issue for applications dealing with business-critical data, such as finance, in consortium blockchains. Consequently, blindly pursuing high scalability through performance sharding may not be a viable option for real-world applications handling highly security-sensitive data (e.g., supply chains on consortium blockchains).

Therefore, we argue that building a practical sharded blockchain requires balancing both performance and availability. To achieve this, it is necessary to explore tradeoffs between traditional performance sharding and availability sharding. As a result, we propose a hierarchical sharded blockchain that divides the network into a two-level structure: the higher level maintains a full copy of all shards, while the lower level holds only a single copy of each shard. This hierarchical design provides excellent opportunities to effectively explore a balanced approach.

B. Sharding Schemes

We provide existing solutions based on the three sharding schemes to clearly compare our approach.

1) Performance Sharding: Performance sharding is a traditional approach to achieving blockchain scalability due to its maximal parallelism and has therefore been widely adopted in various blockchain systems [1]–[11]. A key challenge in performance sharding lies in the cross-shard protocol, which must ensure atomic commitment of cross-shard transactions across

involved shards. For example, traditional coordinator-based two-phase commit (2PC) protocols [1]–[4], [6], [7], [15], methods that split transactions into multiple sub-transactions [5], [8], techniques that move related states to a single shard [3], [8], or a decentralized protocol where all participants communicate directly without a coordinator [1] have been developed to address this complexity. Those cross-shard protocol is known to be notoriously complex and inefficient. Interestingly, some methods [2], [11] introduce a hierarchy within performance sharding to further enhance scalability and simplify cross-shard protocols. In these blockchains, lower-level members store only a single shard of their ledger, while higher-level members either maintain a summarized view of their child ledgers, system-wide metadata for tracking locality, or order cross-shard transactions [2], [11]. An approach to enhancing cross-shard scalability in performance sharding utilizes a Trusted Execution Environment (TEE) [10], where a TEE-generated proof enables lightweight cross-shard block validation, allowing remote shard servers to avoid broadcasting entire blocks.

This performance sharding approach, due to the smaller size of each shard group, involves a significant trade-off in availability, despite achieving higher scalability. In contrast, PyloChain adopts a balanced approach: it uses a hierarchical structure in which lower level members can benefit from performance sharding in terms of reduced storage/networking, and parallel executions, while higher level members maintain a full copy of all shards, ensuring availability in the event of sharding zone failures, and simplifying the cross-shard protocol as well. Also, PyloChain does not rely on TEE.

2) Availability Sharding: Availability sharding allows each member to maintain a full copy of all shards, thereby preventing total data loss in the event of a single shard group failure [16]–[19]. Moreover, it offers the advantage of leveraging collocated sharding within each member for cross-shard transactions, which helps avoid the complex coordination required for isolated shards as in performance sharding. In Meepo [16], [17], a sharded consortium blockchain, all members maintain a full copy of the shards, enabling higher cross-shard efficiency by internalizing cross-shard transaction processing within a single member while ensuring shard availability. In PShard [18], every member participates in all shards, which consist of a root chain and multiple child chains, and executes cross-shard transactions using a root chain-driven 2PC-style commit protocol. Similarly, OHIE [19] achieves excellent throughput and availability by allowing each member to maintain up to k parallel chains. In synchronous network settings, each member in OHIE periodically derives a sequence of confirmed blocks (SCB) to establish a global total order across chains for their local view of the parallel chains.

However, availability sharding systems impose a significant burden on each member, as they require every member to hold a full copy of all shards. This becomes especially demanding for less-capable members as the number of members and shards grows.

3) Balanced Sharding: There exists a balanced sharding scheme harmonizing the above two approaches, with some members maintaining a full copy of shards and others a single

TABLE I
DIFFERENCES FROM EXISTING BLOCKCHAIN SHARDING SYSTEMS.

	Sharding	Sharding Scheme	Cross-Shard Scalability
GeoBFT [30]	✗	N/A	N/A
Meepo [16]	✓	Availability	Low
OHIE [19]	✓	Availability	Low
HieraChain [11]	✓	Performance	High
Saguaro [2]	✓	Performance	High
DyloChain [20]	✓	Balanced	Moderate
PyloChain	✓	**Balanced**	**Enhanced**

copy. Pyramid [28] uses layered sharding with i-shards and b-shards, which hold a single shard and all shards, respectively. To handle cross-shard transactions, Pyramid employs a b-shard-driven 2PC protocol: the b-shard proposes a cross-shard block to i-shards, which accept or reject based on local transaction conflicts. The b-shard then commits or aborts based on the responses. In contrast, PyloChain streamlines this by executing cross-shard transactions on the main chain based on total order and notifying local chains in a single phase. Unlike Pyramid, which supports cryptocurrency applications in a public blockchain, PyloChain is designed for key-value (KV)-based general workloads [4], [29] in a permissioned blockchain.

DyloChain [20] employs a hierarchical sharded model where higher-level M-nodes aggregate and order local blocks for the main chain, validate transactions, and execute cross-shard transactions within an order-execute-order-validate (O-X-O-V) model, synchronizing results into local chains. However, DyloChain's reliance on a synchronous network, requiring a fixed number of local blocks per synchronous round, reduces practicality as slower members can delay main block production, causing substantial latency. Additionally, as main block size grows, the likelihood of local transactions being aborted by global transactions increases, leading to higher synchronization overhead. PyloChain builds on DyloChain by introducing a DAG-based BFT for scalability in a partially synchronous network, enhancing transaction scalability and enabling fine-grained auditing within the DAG context. We present the differences between existing approaches and PyloChain in Table I.

C. Other Scaling Solutions

Another approach to improving blockchain sharding systems involves optimal transaction placement or resharding techniques [6], [9], [11], [13], [31]–[33], which aim to reduce the number of costly cross-shard transactions. This is achieved by observing historical transaction patterns over a system-defined fixed interval and performing a locality-aware mathematical analysis to appropriately place blockchain states across shards, thereby minimizing cross-shard transactions in the future. A simpler version of resharding exploits geographical locality, in which the system tracks users' locations as they move between zones and, accordingly, relocates the state of mobile users to match the new locality [2], [3], [20]. Yet, while resharding techniques can facilitate PyloChain, they are not considered in this paper.

Parallel smart contract execution engines [34]–[38] process transactions using multiple threads within a member after consensus output. In contrast, PyloChain differs by executing transactions first through parallel local chains and performing validation (for local transactions) after consensus.

III. SYSTEM OVERVIEW

A. Assumptions and Models

We assume a permissioned consortium blockchain where the identities of all members are known in advance. We say members are *Byzantine* if they arbitrarily deviates from the protocol, others we call them *honest*. A zone is a logically partitioned sub-network that independently operates a sharded blockchain. There exists f_L and f_F Byzantine failures out of $3f_L + 1$ local members within each zone and $3f_F + 1$ full members across zones, respectively, implying $3f_L+2$ members in a zone. There could be additional machines to be recruited upon detecting the failure of the full member in a zone for recovery. A zone could become unavailable; yet the maximum number of zone failures is limited to f_F, and the probability of zone failure is mutually independent between zones. We assume cross-zone communication is limited to full members; local members do not communicate across zones. We assume that a PyloChain server participates either as a full member or as a local member within each zone, but not both.

We assume a partially synchronous network where synchronous and asynchronous intervals alternate. During the asynchronous intervals, network partitioning is allowed, but this period only lasts for the global stabilization time (GST), denoted by Δ_{GST}. After this, a sufficiently long synchronous interval is maintained, ensuring that all message transmissions are eventually delivered within Δ_{Sync}. We assume that cryptographic primitives used in PyloChain are not broken.

B. Overview

PyloChain is a hierarchical blockchain composed of multiple independent sharding zones. Within each zone, a lower-level sharded local chain is managed by local members, while across zones, a higher-level main chain is managed by full members, as shown in Fig. 2. A full member maintains the higher-level DAG-based main chain [24], [39] with a full copy of all shards for availability. It is responsible for relaying protocol messages across the hierarchy, including bottom-up messages that relay local blocks with local certificates verifying their correctness to the main chain, and top-down messages that relay main block processing certificates with sync entries containing relevant states (e.g., from aborted local or global transactions) to local members.

A local member maintains the lower-level local chain with a single copy of a shard and, using a PBFT-like protocol [40], [41] with other local members in the same zone, reaches consensus on the local blocks proposed by their respective full member to produce a locally agreed-upon sequence. Local members can verify the state of local transactions but cannot verify global transactions. They also detect potentially faulty full members through a fine-grained auditing mechanism and,
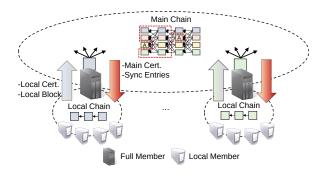

Fig. 2. System Overview. PyloChain is composed of multiple lower-level local chains and a single higher-level DAG-based main chain.

upon detecting failure, can replace the faulty full member with a newly recruited honest one.

Clients submit incoming user transactions to a member within the same zone operating a PyloChain server that they trust. Each user transaction includes read/write sets for a state database, where each entry contains a (key, value, version) tuple, with the version being defined by a block number and transaction offset [29]. Each state is associated with ownership information (i.e., a shard index), which is used to determine the transaction type. As PyloChain uses the O-X-O-V model [20], it categorizes transaction types into local (single shard) and global (multiple shards or remote shards). Local transactions are executed in parallel across multiple local chains, speculatively updating local states, while global transactions are executed and validated only on the main chain. Global transactions may abort conflicting local transactions, also known as interference [20].

IV. PYLOCHAIN: THE PROTOCOL DESIGN

This section provides the protocol design of PyloChain.

A. Local Chain

Local chains are sharded blockchains managed independently within each zone, where full members and local members reach BFT consensus independently from other zones. The primary responsibility of local chains is the speculative execution of local transactions, significantly reducing processing overhead on the main chain. Local chains may need to revert their states if local transactions are finally aborted on the main chain due to interference or if global transactions require updating local chain states.

We describe operations of local chains as follows. A full member receives transaction requests from clients within the same zone, checks its validity (e.g., signature and invoking smart contract's arguments), and categorizes each transaction type based on their accessed states (e.g., through simulation or static analysis). For local transactions, the full member executes the transaction, calculates the read/write set, and updates the local state speculatively. For global transactions, the full member does not execute them and simply includes them in the next local block. Subsequently, the full member generates a local block that includes the aforementioned user transactions and broadcasts it to the local members in the
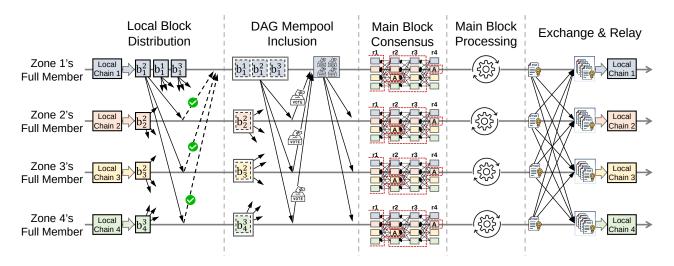
Fig. 3. The Main Chain Protocol of PyloChain. Within each sharding zone, full members concurrently distribute their local blocks to ensure availability. They then propose an ordering for these blocks—while preserving their local sequence—for inclusion in the DAG mempool. Subsequently, once a main block is generated by a consensus, each full member processes the main block. Then, the processing results are exchanged among the full members, forming a certificate signed by a quorum of full members before being delivered to the local chains.

same zone. The local members use their local BFT consensus protocol (e.g., [40]) to verify the integrity of the full member's local block proposal. They check whether the full member has correctly executed the local transactions and calculated a valid read/write set. If validated, each local member speculatively updates its corresponding local state and confirms the agreed-upon sequence of the proposed local block in their local chain by submitting endorsements, resulting in a local commit certificate for the block. When a full member becomes behaves maliciously within a zone, PyloChain simply follows a PBFT-like view-change mechanism within the same zone.

B. Main Chain

Once a local block is committed within a zone, each full member initiates the main chain protocol as illustrated in Fig. 3, to commit the local block at the main chain. We now describe the detailed phase-by-phase operations of the main chain protocol.

a) Local Block Distribution: The goal of *local block distribution* phase is to ensure the availability of local blocks. Specifically, full members collect commit certificates for their local blocks and then reliably broadcast these blocks to other full members, guaranteeing local block availability by receiving $2f_F + 1$ acknowledgments from other full members. For efficiency, full members effectively leverage a parallel workers architecture [24] to concurrently distribute local blocks. While such concurrent distribution of local blocks provides performance benefits by maximizing network bandwidth utilization, it is highly likely that the locally agreed sequence of local blocks will not be preserved in the final consensus order. For example, if there are two local blocks, b_1^1 and b_1^2, from zone 1, b_1^2 might complete its broadcast before b_1^1 due to network conditions. As a result, b_1^2 can appear before b_1^1 in the final order, which violates the locally agreed sequence. We address this issue in the next DAG mempool inclusion phase.

b) DAG Mempool Inclusion: The purpose of *DAG Mempool Inclusion* phase is to include the digest of $2f_F + 1$ availability-guaranteed local blocks, into a vertex, and incorporate it into the DAG mempool. This phase consists of two main operations: First, local order-aware vertex creation: the vertex to be proposed is created by each full member, in a way that it maintains the locally agreed sequence of concurrently broadcasted local blocks. For this, each full member proposes the local block by matching its number in one-to-one correspondence with a monotonically increasing round number as defined by the DAG mempool. This ensures that even if local blocks are completed in different orders during the local block distribution phase, their local sequence is preserved in the DAG mempool seamlessly, with no gaps. Second, DAG mempool inclusion: The full member proposes the created vertex to other full members and obtains $2f_F + 1$ votes to finally produce a DAG mempool inclusion certificate. This certificate is then re-broadcast so that other full members incorporate the vertex into their respective local views of the DAG mempool. For details on the process, including round advance rule, voting rule and commit rule, refer to [24], [39].

Note that although proposing only one local block per round is conceptually simple, but it may lead to performance issues, as the local block corresponding to a particular round number might be completed much later, while subsequent local blocks complete much earlier. In such cases, idle time increases, and communication overhead can surge dramatically. To address this, PyloChain can apply a simple optimization allowing multiple monotonically increasing local blocks to be simultaneously proposed in a single vertex. Since each vertex proposal only includes a list of small-sized block digests, the performance penalty in terms of network bandwidth is negligible.

c) Main Block Consensus: For local blocks included in the DAG mempool, full members periodically apply a locally executed deterministic BFT algorithm, reaching consensus on a committed sub-DAG, i.e., main block. To achieve this,

we follow the partially synchronous Bullshark [39] protocol, summarized as follows: In every even-numbered round, each full member selects a predefined anchor vertex (e.g., in a round-robin manner) and includes in the sub-DAG all vertices within the causal history [39] of the anchor, specifically those between the current anchor and the previous anchor vertex. The vertices included in the sub-DAG are then arranged according to some deterministic topological ordering logic (e.g., depth-first search) to establish a total order across local chains. For example, in Fig. 3, the anchor vertices are denoted by "A" in the second and fourth rounds. The vertices between these two anchors form the sub-DAG, indicated by the red dotted line. The main block, which includes all vertices in the sub-DAG, outputs the corresponding local blocks' digests, the block number, its block hash, and the previous hash. In summary, by performing consensus locally and using a single certificate for the anchor block to commit multiple mempool blocks—i.e., all of its causal histories—the main chain can produce a communication-efficient and high-throughput main block.

d) Main Block Processing: After the main block consensus outputs a main block, i.e., an ordered list of local blocks from the committed sub-DAG, the full member starts to process all transactions within the main block. For main block processing, the full member maintains the main chain-level state DB and version map [20], which enables consistent validation across local chains. For local transactions, since they have already been executed in the local chain, they contain a read/write set. The full member identifies this read/write set and performs validation based on multi-version concurrency control (MVCC) [29]. If validation succeeds, the write set is applied to both the state DB and the version map accordingly. For global transactions, they are executed in the order determined by the main block, calculating their read/write sets, which are then applied to the state DB as well as the version map.

The processing results of the main block include aborted local transactions and all global transactions with their corresponding main chain states. Each full member appends these results to the *syncentries*, which are then delivered to the respective local chains.

However, one issue is that speculatively executed local transactions can be aborted by global transactions on the main chain. Such aborts increase the size of sync entries and reduce the number of meaningful committed transactions, negatively impacting overall performance. To address this, PyloChain applies a simple yet effective scheduling technique: The full member filters global transactions to process all local transactions first, before handling the global transactions during main block processing. This ensures that global transactions within a main block do not interfere with local transactions in the same main block. We consider this effect quite significant, as a DAG-based main block with its throughput-oriented and higher concurrency, can include a very large volume of transactions from multiple local chains, thereby avoiding many local transactions being aborted by global transactions.

Additionally, this scheduling technique offers an opportunity to process mutually independent local blocks from different local chains in parallel, as all global transactions are filtered to the end. This can boost the processing speed of each main block, especially as the number of zones increases. For comparison, we illustrate our processing methods in Fig. 4, showing the DAG-only PyloChain and the scheduling-enhanced version of PyloChain, denoted as PyloChain (DAG) and PyloChain (DAG+Sched), respectively, alongside a naive baseline approach that simply collects a fixed number of local blocks from each local chain (e.g., [20]).

Note that our scheduling within a single main block does not cause fairness issues regarding user-experienced commit latency, unlike scheduling across multiple blocks. This is because a transaction commit event is emitted only after a main block is completely processed, so reordering transactions within a main block does not affect the timing of user-experienced commit events. From the perspective of transaction abortability, although global transaction scheduling clearly prevents interference among local transactions within a main block, it may still introduce interference in subsequent main blocks. Nonetheless, since this interference is independent of the scheduling, fairness in terms of transaction abortability is not an issue. Additionally, the computational overhead of scheduling is minimal since it merely involves iterating over blocks to filter global transactions. Moreover, as this filtering can be executed in parallel across blocks, latency can be further reduced. However, fairness concerns may arise outside the protocol path, especially if the PyloChain server operator monitors execution logs in real-time and externally notifies transaction results during main block processing.

e) Exchange and Relay: After main block processing, each full member generates a main block processing proof for the main block, summarizing the results of main block processing, and broadcasts it among themselves to collect $f_F + 1$ proofs. The proof includes the main block number, main block hash, previous main block hash, the sequence of local block headers within the main block, the identity of the creator, and the digest values of the sync entries that need to be committed to each local chain. Each sync entry consists of a key, value, and transaction ID. Once $f_F + 1$ proofs are gathered, each full member creates a main block processing certificate and propagates it along with the payloads of the sync entries, to their respective local chain. Then, the local members verify the main block processing certificate and sync entries, allowing them to reliably confirm that the local blocks from their local chain have been committed to the main chain. Additionally, they safely synchronize the states of aborted local transactions and committed global transactions to the local chain using the payloads contained in the sync entries.

C. Algorithms

We provide the pseudocode of the main chain protocol of PyloChain in Algorithm 1 and Algorithm 2 to clearly illustrate its operations. We introduce additional notations as follows. A mempool is denoted by $\mathcal{M}$, and the mempool for zone z by $\mathcal{M}_z$. $\mathcal{M}_z$ adds zone z's local blocks with the method Add(), which stores the block and internally increments the latest monotonically increasing block number. It supports
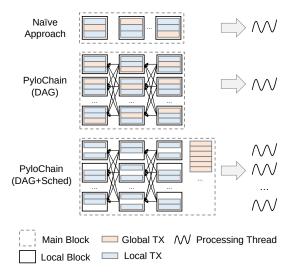
Fig. 4. Comparisons of main block processing. A naive approach simply aggregates a fixed number of local blocks from each local chain into a single main block, resulting in a linearly growing chain with a single processing thread. In contrast, PyloChain (DAG) concurrently appends local blocks from each local chain, and PyloChain (DAG+Sched) further optimizes processing with multiple threads and scheduling.

Pop(), which returns and deletes the lowest value among the stored monotonically increasing sequences, and Next(), which provides the next value following the most recently returned value. Full and local members from zone z are denoted by Π_F^z and Π_L^z, respectively. Π_F denotes all full members across zones. A signed message by participant p is denoted as $\langle \cdot \rangle_{\sigma_p}$. We define DAG as a black-box module that performs DAG mempool inclusion and main block consensus, providing two functions: Include and Consensus. The Include function adds a newly created vertex to the DAG, while the Consensus function interprets the DAG and notifies the topologically ordered committed sub-DAG with a sequence number for the main chain. We first describe the protocol for full members and then explain the protocol for local members.

Initially, each full member maintains an empty mempool, $\mathcal{M}$ (L1). Upon receiving a local block certificate for a local block b_i^z from Π_L^z, a full member $p \in \Pi_F^z$ adds the block to its mempool $\mathcal{M}$ and broadcasts $\langle \text{REPL}, z, i, b_i^z \rangle_{\sigma_p}$ to other full members for block availability (L2-L3). Upon receiving the REPL message, a full member stores the local block from zone z into its z-indexed mempool $\mathcal{M}_z$ and replies with $\langle \text{ACK}, z, i, D(b_i^z) \rangle_{\sigma_p}$ (L4-L5). When the proposing member p has collected $2f_F + 1$ ACK messages including its own (L6), it prepares to propose a vertex containing the set of block digests. The Next() method of $\mathcal{M}_z$ returns the next block number to be proposed from zone z. If this matches the block number i from the ACK messages, member p consecutively executes Pop() from $\mathcal{M}_z$, adding digests $D(b)$ to $candidates$ from block number i up to the latest available block number. The member p then generates a block availability certificate $cert_{avail}$ from the $2f_F + 1$ ACK messages, asynchronously executes DAG.Include with these digests, and broadcasts $\langle \text{AVAIL}, cert_{avail}, z, i \rangle_{\sigma_p}$ to local members in zone z (L6-L15).

Algorithm 1: The Main Chain Protocol

// For Full Members

1   $\mathcal{M} \leftarrow \emptyset$;
2   **on receiving** a local block certificate $cert_{local}$ for b_i^z at $p \in \Pi_F^z$:
3     $\mathcal{M}_z$.Add(b_i^z); broadcast $\langle \text{REPL}, z, i, b_i^z \rangle_{\sigma_p}$ to Π_F
4   **on receiving** $\langle \text{REPL}, z, i, b_i^z \rangle_{\sigma_p}$ at $q \in \Pi_F$:
5     $\mathcal{M}_z$.Add(b_i^z); reply $\langle \text{ACK}, z, i, D(b_i^z) \rangle_{\sigma_q}$ to p
6   **on receiving** $2f_F + 1$ $\langle \text{ACK}, z, i, D(b_i^z) \rangle_{\sigma_p}$ at $p \in \Pi_F^z$:
7   **if** $\mathcal{M}_z$.Next() $= i$ **then**
8       $candidates \leftarrow \emptyset$
9   **while** $b \leftarrow \mathcal{M}_z$.Pop() **is not** nil **do**
10        $candidates \leftarrow candidates \cup \{D(b)\}$
11   **end**
12   **end**
13     $cert_{avail} \leftarrow$ a certificate from $2f_F + 1$ ACK msgs.
14   DAG.Include($candidates$, $cert_{avail}$)
15   broadcast $\langle \text{AVAIL}, cert_{avail}, z, i \rangle_{\sigma_p}$ to Π_L^z
16   **on receiving** $\langle \text{CON}, \tau, k \rangle$ *from* DAG.Consensus() *at* $p \in \Pi_F^z$:
17     $B_k \leftarrow$ CreateMainBlock($\mathcal{M}, \tau, k$)
18     $ents \leftarrow$ ProcessMainBlock(B_k)
19     $\pi_k \leftarrow$ MainBlockProof($ents$,$B_k.Hdrs$, k)
20     $cert_{proc} \leftarrow$ exchange π_k among Π_M
21   broadcast $\langle \text{PROC}, cert_{proc}, k, ents_z \rangle_{\sigma_p}$ to Π_L^z

// For Local Members

22   **on receiving** a local block certificate $cert_{local}$ *for* b_i^z at $p \in \Pi_L^z$:
23   verify $cert_{local}$ and start a timer Δ_{avail}
24   **on receiving** $\langle \text{AVAIL}, cert_{avail}, z, i \rangle_{\sigma_p}$ *at* $p \in \Pi_L^z$:
25   verify $cert_{avail}$, exit the timer Δ_{avail}, and start a timer Δ_{proc}
26   **on receiving** $\langle \text{PROC}, cert_{proc}, k, ents_z \rangle_{\sigma_p}$ *at* $p \in \Pi_L^z$:
27   verify $cert_{proc}$ and exit a timer Δ_{proc}
28   update sync entries in $ents_z$, according to k, to local state DB

The asynchronously executed result from DAG.Consensus() is returned within a CON message as event notifications with ordered digests τ and their sequence number k. Using these, the k-th main block B_k is generated by CreateMainBlock, processed by ProcessMainBlock (which will be explained in 2), returning the updated states as $ents$. A processing proof, which is a signature over inputs including entries $ents$, local block headers $B_k.Hdrs$, and the main block number k, is generated by MainBlockProof and then exchanged among full members to create a processing certificate. Note that $ents$ contain both zone-level hashes and a global hash, and the zone-level hashes can be indexed. Finally, each full member broadcasts the main chain processing results and zone-specific entries ent_z within the message $\langle \text{PROC}, cert_{proc}, k, ents_z \rangle_{\sigma_p}$ to all local members in zone z, with invoking local chain consnesus (L16-L21).

The main block processing is presented in Algorithm 2. The

Algorithm 2: Main Block Processing

---

1 **Function** ProcessMainBlock (B) **:**
2     $globalTXs \leftarrow$ extract global TXs from B
3     $ents \leftarrow \{\}$
4   **foreach** $tx \in B_k$ **do**
5         $res \leftarrow$ Validate(tx)
6   **if** $res = success$ **then** Update(tx); **continue**
7   AppendEntry ($ents, tx.Id, tx.WSet$)
8   **end**
9   **foreach** $tx \in globalTXs$ **do**
10         $rwset \leftarrow$ Execute(tx)
11   AppendEntry ($ents, tx.Id, rwset.WSet$)
12   Update(tx)
13   **end**
14   **return** $ents$
15 **Function** AppendEntry ($ents, txid, wset$) **:**
16   **foreach** $w \in wset$ **do**
17         $z \leftarrow$ Ownership($w.key$)
18         $v_{lat} \leftarrow$ GetState($z, w.key$)
19   Add($ents_z, \{txid, w.key, v_{lat}\}$)
20   **end**

---

algorithm takes the main block B, extracts global transactions through filtering (L2), and initializes sync entries (L3). Then, it performs MVCC-based validation [20] using Validate on the local transactions from the filtered main block. If validation succeeds, the results are updated in the main-chain state DB (Update), and the protocol proceeds to the next iteration. If not, i.e., when speculative local updates are aborted, the latest main-chain states of the involved write keys must be synchronized to the local state, for which we use AppendEntry (L4–7). The AppendEntry operation adds a synchronization entry to the given $ents$. Specifically, for each write w in the write set $wset$, it identifies the ownership z (i.e., shard index) of its key $w.key$, retrieves the latest state v_{lat} from the state DB, and appends it to $ents$ (L15–L20). Subsequently, global transactions are executed sequentially to compute their $rwset$, update the results to the state DB, and append them to $ents$ (L9–12). After processing global transactions, the algorithm finally returns the sync entries $ents$ (L14).

We describe the local member protocol as follows: After a local member p in a zone z identifies the local block certificate $cert_{local}$ for b_i^z, the local member verifies its certificate and starts a timer Δ_{avail} (L22-23). When an AVAIL message is delivered by the full member in z, each local member verifies the $cert_{avail}$, exits the timer Δ_{avail}, and starts another timer Δ_{proc} to measure the delay until b_i^z is processed on the main chain (L24-25). Finally, upon receiving the corresponding PROC message, each local member verifies the $cert_{proc}$, exits the timer Δ_{proc}, and updates the entries in ent_z to the local state database according to k (L26-28). Note that we intentionally omit the local sequence number for simplicity. Local members within zone z consistently update the entries through the local chain consensus. We discuss timer configurations under a partially synchronous network, along with failure handling scenarios, in Section IV-D and Section

IV-E.

D. Auditing Full Member's Trustworthiness

We now describe the mechanism for auditing full members' trustworthiness. Full members, who relay protocol messages between local and main chains, introduce significant security challenges for PyloChain. Malicious relay attacks are classified into two types: (1) Bottom-up attacks occur when a full member withholds local blocks (compromising availability) or broadcasts malicious blocks (compromising integrity). (2) Top-down attacks occur when a full member fails to deliver main block processing certificates to the local chain (compromising liveness) or reports fake certificates (compromising integrity).

Among these, data integrity is straightforward to guarantee, as local blocks and main block processing certificates contain signatures from $2f_L + 1$ local members and $2f_F + 1$ full members, respectively. In other words, even if a malicious full member attempts to inject fake local blocks into the DAG mempool, these blocks will not be included in the honest full members' DAG mempool without valid local block certificates. Similarly, fake entries generated by malicious full members through incorrect main block processing will not be applied to the local chain without a valid full members' certificate. While a full member might reorder local blocks in the DAG mempool against the locally agreed sequence, local members can detect this by verifying the order of local block headers in the subsequent main block processing certificate.

But timing-related faults, such as delaying or dropping relayed messages, require careful handling. Local members should be able to confirm that their local blocks are included in the main chain in a timely manner via the main block processing certificate from their full member. However, because the main chain operates in a partially synchronous network and is managed solely by full members, local members lack direct visibility into its status. Thus, the full member must provide proof of correct behavior to the local members.

PyloChain addresses this by making each full member responsible for externalizing fine-grained main chain operations, along with their corresponding certificates, to its local members, including local block availability, DAG mempool inclusion, and main block consensus and processing. This allows local members to use a timer to check whether each main chain operation is delivered within the expected latency once a local block is agreed upon in their local chain. We first define the expected maximum latency for these operations during the normal synchronous period.

For the local block availability certificate, it takes $3\Delta_{Sync}$ (i.e., Δ_{avail}): $1\Delta_{Sync}$ for broadcasting the local block to the full members, $1\Delta_{Sync}$ for collecting $2f_F + 1$ acknowledgments from other full members, and $1\Delta_{Sync}$ for delivering these acknowledgments to the local members. For the DAG mempool inclusion certificate, it takes $3\Delta_{Sync}$: $1\Delta_{Sync}$ for the full member's vertex proposal, $1\Delta_{Sync}$ for collecting $2f_F + 1$ votes from other full members, and $1\Delta_{Sync}$ for delivering these votes to the local members. For the main block consensus certificate, it takes $4\Delta_{Sync}$ because consensus is reached every two rounds using the partially synchronous Bullshark

algorithm [39], and the next round's vertex must follow the previous two DAG mempool steps. For the main block processing certificate (i.e., Δ_{proc}), it takes $2\Delta_{Sync}$: $1\Delta_{Sync}$ for full members to broadcast their main block processing certificates and gather $f_F + 1$ certificates, and $1\Delta_{Sync}$ for delivering them to the local members. Since the main block consensus is performed locally, it can be combined with the main block processing certificates and delivered in a single batch, with the combined operation expected to take a total of $6\Delta_{Sync}$. Note that we omitted externalizing the DAG mempool inclusion certificate in Algorithm 1 for simplicity, as this phase's certificate can be incorporated into the main block processing certificate.

By using timers based on the expected confirmation latency of externalized operations, local members gain visibility into main chain operations conducted by full members, allowing them to audit their trustworthiness. However, in a partially synchronous network, unexpected latency periods can occur, during which necessary certificates might not be delivered within the expected confirmation latency. In such cases, local members may ask, *"Is this due to being in the GST period, or is it our full member's fault?"* For this, they can either conservatively wait up to Δ_{GST} for the message to arrive, concluding a fault by the full member if it still doesn't arrive, or proactively determine a fault as soon as the synchronous timer expires. In PyloChain, we adopt the conservative approach, assuming Byzantine attacks by full members are infrequent in a permissioned consortium blockchain.

We describe PyloChain's behavior under asynchronous network conditions during the GST period. In the first case, where messages are eventually delivered after the GST period, local members record uncommitted local blocks in a pending block list during the GST period. Once the GST period ends and the synchronous period begins, missing certificates for the pending local blocks are delivered by the full members, and the corresponding local blocks are removed from the pending list, after which normal operations resume. In the second case, where messages are not received even after the GST period ends, local members also record uncommitted local blocks in the pending block list during the GST period. However, if the expected messages are still not received after the GST period ends and the synchronous period begins, local members conclude that their full member is malicious.

E. Faulty Full Member Handling

Since the main chain of PyloChain leverages a DAG mempool, $2f_F + 1$ local block availability is supported as long as the local blocks are included in the DAG mempool. However, full members play a critical role, as they participate in both local chain and main chain consensus. Therefore, if a full member is malicious, it can impact the system's performance and security.

We first explain the impact of a faulty full member on the main chain. DAG-based main chain consensus inherently does not require a traditional view change mechanism due to the nature of DAG construction [39], [42]. This is because, in DAG mempool construction, no entity is assigned a special role. Even in the presence of a faulty full member, honest full

members can proceed completely asynchronously at network speed. Specifically, for local block distribution that ensures data availability, only acknowledgments from $2f_F + 1$ members are required to proceed. On the other hand, the Bullshark consensus proceeds by interpreting DAG construction on a round-by-round basis. In this scheme, every even-numbered round relies on a predetermined anchor vertex (known to all via a deterministic mapping function, e.g., round-robin) to maintain consistency in DAG ordering logic. In scenarios where the anchor vertex belongs to a faulty full member, this member can either: 1) fail to propose the vertex entirely or 2) propose it selectively to only a subset of members. Nevertheless, Bullshark safely skips these faulty vertices by leveraging the $f_F + 1$-commit rule. By committing the next honest anchor vertex in a subsequent round, Bullshark can retroactively and amortizedly commit the path from that honest anchor vertex back to the previous committed anchor vertex. Consequently, although a faulty full member can slightly delay the consensus speed, it does not negatively impact throughput. This characteristic is a primary advantage of introducing DAG-based consensus into the upper layer of PyloChain, significantly minimizing overhead from malicious actions by full members.

We now explain how PyloChain recovers a faulty full member. Within a zone, a traditional PBFT-like view change mechanism [40] is leveraged to replace the faulty full member with a new full member. This mechanism enables the correct migration of the local chain from the previous view to the new view within the zone. However, in a hierarchical blockchain, such a traditional view change alone is insufficient, as the main chain must also be considered. We describe the recovery from a cross-zone perspective as follows: First, the new full member joins the main chain network. For this, the new full member must obtain participation permission, which can be managed via a consensus-based membership change protocol [43]. Next, the new full member catches up with the main chain by contacting honest members and downloading main blocks (i.e., DAG mempool and sub-DAGs). To avoid costly re-execution, the main states are downloaded as well. For fault tolerance, a new full member first verifies state integrity by querying digests from $f_F + 1$ members, and then downloads the actual large-volume data from only one member to verify it.

Then, the new full member identifies the highest local block number whose availability has been ensured on the main chain. Next, the new full member identifies any pending local blocks that have been locally committed but not yet included in the DAG mempool. Finally, the new full member resumes the main chain protocol by starting to submit the pending local blocks to the DAG mempool. Simultaneously, the new full member resumes normal local chain operations by processing incoming client transactions. By applying this recovery procedure, PyloChain ensures robust fault tolerance and maintains system liveness, even in the presence of malicious full members.

F. Correctness Analysis

We define one liveness and two safety properties of PyloChain in the context of a hierarchical sharded blockchain for proving its correctness. We assume the BFT algorithms operating within each local chain and the main chain are correct. Additionally, for simplicity in proving correctness, we address a full member failure instead of an entire zone failure. Then, we define the following properties that PyloChain should guarantee:

- **Validity** If a local block is agreed upon in a local chain, then that local block will eventually be confirmed on the main chain.
- **Bottom-up Consistency** If two local blocks b_x^i and b_x^j are agreed upon within the same x-th local chain where $i < j$, then b_x^i is ordered before b_x^j within the main chain.
- **Top-down Consistency** A local chain does not finalize any transactions that are inconsistent with the processing results on the main chain.

We then prove that PyloChain guarantees Validity, Bottom-up Consistency, and Top-down Consistency.

Theorem IV.1. *PyloChain guarantees Validity.*

Proof. Assume that a local block is agreed upon in a local chain and its full member is malicious enough not to follow the main chain protocol. Then, the malicious full member will cause a failure in at least one phase of the main chain protocol for that local block. This failure will be detected by a fine-grained auditing mechanism carried out by local members from the same local chain. Specifically, the malicious full member might fail to deliver the local block availability certificate within $3\Delta_{Sync}$ during the local block distribution phase, or fail to deliver the main block processing certificate within $9\Delta_{Sync}$ (including $3\Delta_{Sync}$ for DAG inclusion, $4\Delta_{Sync}$ for main block consensus and $2\Delta_{Sync}$ for main block processing certificate). If a failure occurs in any of these phases, the local members record the local block as pending, start a GST timer, and, upon expiration, the local members recover the full member through a PBFT-like view change. Then, a new honest full member will execute the recovery process as in Sec. IV-E to resume main chain protocol again on the pending local block.   □

Theorem IV.2. *PyloChain guarantees Bottom-up Consistency.*

Proof. We first mention that PyloChain is resilient to a full member's fork attack— that is, generating different local blocks for the same sequence number. This is because, in the main chain protocol, full members consider only local blocks with $2f_L + 1$ local block certificates to be valid. As a result, maliciously forked local blocks are ignored by honest full members in the main chain protocol. Let b_x^i and b_x^j be two agreed-upon blocks from the x-th local chain, where $i < j$. These blocks can be safely submitted to the main chain, and during the local block distribution phase, they can be completed in any order. However, in the DAG mempool inclusion phase, an honest full member in the x-th local chain proposes the digest of b_x^i first, followed by b_x^j. If a malicious full member violates this order—e.g., by proposing b_x^j before b_x^i—the other honest full members can easily detect this by checking its monotonically increasing round number and its block number and will refrain from voting. As a result, the malicious full member will fail to deliver the DAG mempool inclusion certificate to its local members within the expected time. Eventually, this violation will be detected by a fine-grained auditing mechanism and, after GST, the malicious full member will be replaced. The newly recovered honest full member will then re-propose those pending local blocks, ensuring that PyloChain guarantees Bottom-up Consistency.   □

Theorem IV.3. *PyloChain guarantees Top-down Consistency.*

Proof. Assume that a global transaction has been committed on the main chain. Then, all local transactions that were issued on the involved local chains, until the sync entry of this global transaction is committed to those chains, are subject to being aborted. There are two cases. First, there may be subsequent local transactions included in some main blocks, which are dependent on that global transaction, are doomed to be aborted. Second, there may be inflight local transactions that are speculatively executed on the local chain, updating the local states and preparing to be included in a main block. Despite these potential inconsistencies, PyloChain ultimately finalizes all transactions on the main chain. Additionally, since involved local chains finalize their local transactions only if the main block processing certificate is confirmed, if a speculatively executed local transaction is aborted on the main chain, it deterministically synchronizes the latest main states based on the sync entries of the aborted local states. Consequently, a local chain never finalizes a result that contradicts the main chain. If a full member is malicious and incorrectly processes a main block, this can be validated through the main block processing certificate, which contains the execution results from the other $2f_F + 1$ honest full members. Therefore, PyloChain guarantees Top-down Consistency.   □

V. EVALUATION

A. Implementation

To evaluate the feasibility of the PyloChain designs, we prototyped PyloChain in GoLang [25] and integrated it into Hyperledger Fabric [26] to support smart contracts, incorporating necessary modifications for the PyloChain implementation. Local members are implemented as peer nodes, while full members consist of both peer nodes and orderer nodes for block ordering and cross-zone communications. We configured the peer nodes to execute smart contracts locally. We use LevelDB [44] for state management database. For DAG BFT, we utilize the Rust-based implementation of Narwhal/Bullshark [27], [45] and extend it to support the main chain of PyloChain. Specifically, we configure its number of block distributors (called Workers) to match the number of shards, which directly affects the performance of each full member in concurrently sending and receiving local blocks [24]. We use a partially synchronous version of Bullshark

[39] for the consensus logic, rather than its fully asynchronous version, to align with the network assumptions of PyloChain.

We also implemented other sharding schemes, i.e., availability sharding and performance sharding, within our environment for comparison with PyloChain. For availability sharding, we simply configured full members to propagate the collected local blocks from the DAG mempool to their respective local members within the same zone. For performance sharding, we removed the main chain and implemented a 2PC protocol to handle cross-shard transactions instead. In the 2PC protocol, a designated full member acts as the global coordinator. All cross-shard transactions issued within zones are forwarded to this coordinator, which then carries out the 2PC protocol, i.e., the prepare phase for locking and aggregating the involved states, and the commit phase for finalizing the transaction (i.e., commit or abort) and delivering the finalized results to the involved local chains.

B. Experimental Setup

We conducted our experiments on an in-lab cluster consisting of 24 machines with three different configurations: 6 high-end machines with AMD Ryzen CPU 3990x (2.9GHz) and 256GB RAM, 3 medium-end machines with AMD Ryzen CPU 3970x and 128GB RAM, and 12 low-end machines with AMD Ryzen CPU 5950x (3.4GHz) and 32GB RAM. All machines feature Samsung SSD 970, run Ubuntu 20.04, and are connected via a 10 Gbps Ethernet network. We evaluated PyloChain in an environment with up to 18 zones, where in each zone, we deployed one full member, four local members, and four clients, which sums up to a total of 18 full members for the main chain, and 72 local members for 18 local chains, respectively. All members in our experiment run as Docker containers [46], orchestrated by Docker Swarm, and deployed on its overlay network.

We set up a micro-payment application using SmallBank [47] with 300k users equally distributed and statically initialized for each zone. The average size of each transaction and a local block are 2.89KB and 1.4MB, respectively. The experimental parameters of the main chain using DAG BFT [27] are as follows: each worker batch is set to include one local block. A primary can propose up to 40 digests per vertex. The maximum delay for a primary to propose a vertex is 800ms. We measured PyloChain's throughput and latency on the client side, with each client asynchronously submitting transaction requests at a send rate ranging from 10k to 110k, in steps of 10k, evenly distributed across the clients. Latency was measured as the time between the client submitting a transaction and receiving a commit event from the local members, while throughput was calculated by summing the number of transactions committed per second as measured by all clients.

C. Overall Performance

We demonstrate PyloChain's scalability in throughput and latency by varying the number of zones and the global transaction ratio, comparing it with performance and availability sharding (Fig. 5). We distinguish two PyloChain versions:

TABLE II
PERFORMANCE ANALYSIS IN TERMS OF MAIN BLOCK

	%.Global TX	Proc. Delay(ms) (#Zones: 6 / 12 / 18)
PyloChain (DAG)	0%	119 / 94 / 190
	20%	212 / 193 / 299
	40%	241 / 225 / 331
PyloChain (DAG+Sched)	0%	53 / 48 / 81
	20%	105 / 94 / 136
	40%	145 / 123 / 172

PyloChain (DAG), using only DAG-based BFT consensus, and PyloChain (DAG+Sched), adding scheduling technique. We first compare PyloChain with DyloChain, then with other sharding schemes.

1) Zone Scalability: We first examine all local workloads (0% global transactions). Notably, PyloChain (DAG) consistently outperforms DyloChain, especially in latency. With 9 zones, PyloChain (DAG)'s latency (2.67s) is 1.47x faster than DyloChain (3.92s), though throughput is similar (PyloChain: 37,432 TPS, DyloChain: 35,148 TPS). With 18 zones, PyloChain (DAG) achieves greater improvements (37,090 TPS, 4.84s latency) compared to DyloChain (30,672 TPS, 8.13s latency). This difference arises from main block consensus methods. DyloChain synchronously includes a fixed number of local blocks, increasing latency and reducing throughput as zones scale. PyloChain (DAG)'s asynchronous DAG-based consensus allows flexible inclusion based on varying workloads, improving scalability and efficiency. Performance peaks at 12 zones, then declines beyond 15 zones.

PyloChain (DAG+Sched) shows minimal performance gains over PyloChain (DAG) with zero global transactions. This is because, despite multi-threaded parallel processing in PyloChain (DAG+Sched), validating already-executed local transactions incurs minimal overhead, thanks to the O-X-O-V model. Note that at higher zones with low send rates, latency increases as clients send fewer transactions, not enough to fill each local block batch size in a timely manner.

2) Global Transaction Scalability: With 20% and 40% global transactions in Fig. 5, saturation occurs earlier due to heavier main block processing overhead from global transactions (i.e., smart contract execution, transaction conflicts, and synchronization). In these cases, PyloChain (DAG+Sched) consistently outperforms others. For instance, at 12 zones (20% global transactions), PyloChain (DAG+Sched) achieves 37,642 TPS, 2.80s latency, outperforming PyloChain (DAG) by 1.17x throughput, 1.83x latency, and DyloChain by 1.25x throughput, 1.95x latency. At a send rate of 90,000, PyloChain (DAG+Sched) achieves 57,263 TPS with 3.10s latency, offering 1.39x and 1.49x higher throughput, and 2.76x and 2.63x faster speed compared to PyloChain (DAG) and DyloChain, respectively. This is because PyloChain (DAG+Sched) improves by scheduling global transactions at the end of each main block, significantly reducing local transaction interferences and synchronization overhead. Nevertheless, when the proportion of global transactions increases, all systems experience performance degradation because the execution of global
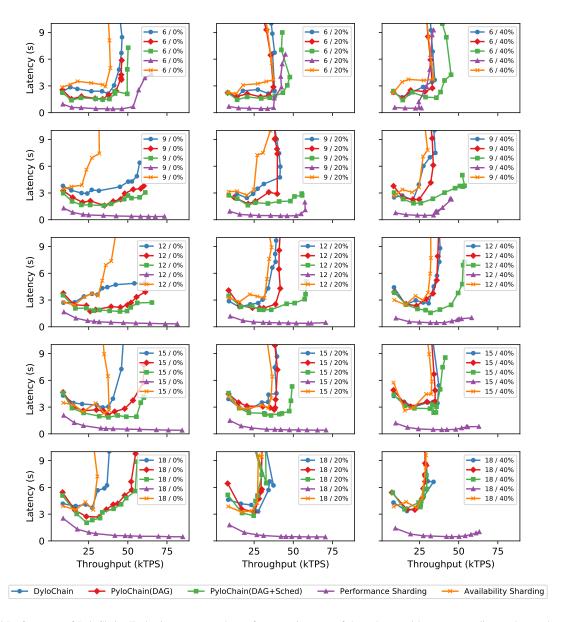
Fig. 5. Overall Performance of PyloChain: Each plot represents the performance in terms of throughput and latency, according to the number of zones and the percentage of global transactions, denoted by #Zones / %GlobalTX. For example, "12 / 20%" indicates that the experiment was conducted in 12 zones with 20% of the global transactions.

transactions significantly slows down main block processing.

3) Comparisons with Other Sharding Schemes: Comparing PyloChain's balanced sharding with availability and performance sharding, balanced sharding outperforms availability sharding but trails performance sharding. Availability sharding incurs significant overhead due to local block distribution to all members. Performance sharding horizontally scales due to no main chain overhead, even under global transactions. PyloChain's scalability is limited by its main chain's all-to-all local block broadcasting, constraining throughput due to higher network bandwidth usage.

D. Performance Analysis

We analyzed PyloChain's performance in terms of main block processing delay across varying zones (6, 12, 18) and
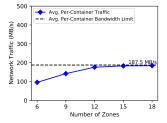

Fig. 6. Bandwidth consumption.

global transaction percentages (0%, 20%, 40%), as shown in Table II. Notably, as the percentage of global transactions and the number of zones increase, the main block processing delay rises. This occurs because a higher num-
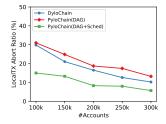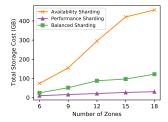
Fig. 7. Interference Analysis

Fig. 8. Storage Cost

ber of global transactions is executed, and each main block processing involves more interfered local transactions, introducing additional overhead. However, scheduling notably reduces main block processing delays. For example, PyloChain (DAG+Sched) processes main blocks in 94 ms, compared to 193 ms for PyloChain (DAG) with 12 zones, demonstrating a 2.05x speedup. This performance improvement suggests that scheduling reduces the overhead associated with handling interfered local transactions, even though the absolute number of global transactions remains constant.

We also measured bandwidth consumption as zones scales as in Fig. 6. The analysis reveals a network bottleneck at 12 zones due to container bandwidth limits in full members, driven by significant local block broadcasts under balanced sharding. Note that although the physical machine's theoretical bandwidth is 10 Gbps (1,250 MB/s), the actual bandwidth utilization of containers on the Docker Swarm overlay network is observed to be much lower.

E. Interference Analysis

We examined transaction interference at 15 zones, 40% global transactions, and an 80k send rate as in Fig. 7. PyloChain (DAG) experienced higher abort rates than DyloChain due to larger main blocks. However, PyloChain (DAG+Sched) significantly reduced interference via scheduling. With 300k accounts, PyloChain (DAG+Sched)'s abort ratios (10.20%, 13.19%, 5.66%) were 1.8x and 2.3x lower than DyloChain and PyloChain (DAG).

F. Storage Cost

Fig. 8 compares storage consumption across sharding schemes. In this experiment, clients in each zone submitted transactions at a rate of 4000 per seconds, with all transactions being local, and the experiment lasting one minute. As expected, storage costs were highest for availability sharding (up to 457 GB), as local members, along with full members, also store local blocks across zones. Balanced sharding, which requires only full members to store local blocks across zones (up to 122 GB), was closer to the performance of performance sharding, where each member holds only their respective local chain (up to 30.7 GB).

VI. DISCUSSION AND FUTURE WORK

Sharding: Performance and Availability Trade-Off: It is obvious that, assuming the total number of participants remains fixed, increasing the number of shards in a sharded system can lead to higher performance due to enhanced parallelism and reduced consensus overhead. Yet, as the size of each consensus group shrinks proportionally as the number of shards increases, the availability of shard data inherently decreases, making the system highly susceptible to disastrous events [21]–[23],compared to fully replicated, non-sharded systems. Nevertheless, there are approaches to improve shard availability in performance sharding. For instance, geographically distributing shard servers can partially mitigate regional disasters [22], [23]. However, such geographical distribution incurs costly cross-region communications, particularly detrimental to message-passing-based BFT consensus protocols. Additionally, even geographically distributing shard servers across regions but deploying them to a single cloud vendor could lead to data unavailability due to vendor-specific failures (e.g., single-point management failures from bad software upgrades [21]). Considering these scenarios, a practical mitigation strategies from distributed system practices involves deploying shard servers across multiple geographic regions and multiple cloud vendors to address performance sharding's inherent availability issues. Despite these standard practices, fundamentally, improving availability in performance sharding involves a trade-off with performance, requiring increased replication. PyloChain aims to balance this trade-off effectively by introducing a hierarchical sharded blockchain architecture.

Optimizations: As demonstrated in our evaluation, reducing the network and storage overhead on full members in PyloChain is necessary to achieve further scalability as the number of zones increases. For this, erasure coding techniques (e.g., Reed-Solomon$(3f + 1, 2f + 1)$) can be utilized for full members [48]. By adopting erasure coding, only a fraction of each encoded local block—rather than the entire block—needs to be propagated, significantly reducing network communication and storage overhead. However, one challenge remains: erasure coding requires costly reconstruction operations to access block contents. Nevertheless, PyloChain is expected to mitigate this cost by performing selective reconstruction only when processing global transactions. Additionally, global transactions in PyloChain are executed sequentially on the main chain, which could significantly increase main processing delays, especially as their percentage grows or application computation becomes more expensive. Fortunately, our main chain can easily integrate existing parallel transaction execution algorithms [34]–[38], [49] to accelerate the processing of global transactions. We consider these optimizations as future work.

Asynchronous Network and Flexible Consensus: While partially synchronous models, as assumed in PyloChain, are common, prolonged asynchronous periods or delayed GST can occur in practice, potentially limiting PyloChain's applicability in environments with highly variable network conditions (e.g., fully asynchronous settings). Currently, PyloChain leverages a DAG-based BFT protocol for the higher-level main chain, in which the mempool (for data distribution) and the consensus (for total ordering) are fully separated, supporting fully asynchronous and partially synchronous networks, respectively. Meanwhile, Tusk [24] is a fully asynchronous consensus protocol known to achieve higher throughput, but at the cost of increased latency compared to Bullshark [39], [42], which is currently used in PyloChain. Based on this moudlarity of DAG BFT, we consider adaptively switching the consensus algorithm in DAG-based BFT according to changing network conditions. For example, the system could initially employ Bullshark in partially synchronous environments, then dynamically switch to Tusk upon detecting prolonged asynchronous periods. We leave this adaptive, condition-based consensus switching as future work, aiming to enhance the overall applicability of PyloChain.

VII. CONCLUSION

In this paper, we propose PyloChain, a hierarchical sharded blockchain that strikes a balance between performance and availability. PyloChain effectively leverages a recent DAG-based mempool to provide local block availability and to achieve efficient main block consensus via locally performed BFT, across sharding zones. It accelerates transaction processing through parallel sharded local chains and a simple scheduling technique to further reduce global transaction interferences. We conduct a systematic evaluation to demonstrate PyloChain's effectiveness.

REFERENCES

[1] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD/PODS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 76–88. [Online]. Available: https://doi.org/10.1145/3448016.3452807

[2] M. J. Amiri, Z. Lai, L. Patel, B. T. Loo, E. Lo, and W. Zhou, "Saguaro: An edge computing-enabled hierarchical permissioned blockchain," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 259–272.

[3] M. J. Amiri, D. Shu, S. Maiyya, D. Agrawal, and A. El Abbadi, "Ziziphus: Scalable data management across byzantine edge servers," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 490–502.

[4] H. Dang, T. T. A. Dinh, D. Loghin, E. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 123–140. [Online]. Available: https://doi.org/10.1145/3299869.3319889

[5] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 95–112. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping

[6] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1968–1977.

[7] 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)*. IEEE, 2018, pp. 583–598.

[8] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 931–948. [Online]. Available: https://doi.org/10.1145/3243734.3243853

[9] F. Cheng, J. Xiao, C. Liu, S. Zhang, Y. Zhou, B. Li, B. Li, and H. Jin, "Shardag: Scaling dag-based blockchains via adaptive sharding," *2024 40th IEEE International Conference on Data Engineering (ICDE)*, 2024.

[10] Z. Cai, J. Liang, W. Chen, Z. Hong, H.-N. Dai, J. Zhang, and Z. Zheng, "Benzene: Scaling blockchain with cooperation-based sharding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 639–654, 2023.

[11] H. Tang, H. Zhang, Z. Zhang, Z. Zhang, C. Jin, and A. Zhou, "Towards high-performance transactions via hierarchical blockchain sharding," in *Euro-Par 2024: Parallel Processing: 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, August 26–30, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 373–388. [Online]. Available: https://doi.org/10.1007/978-3-031-69577-3_26

[12] Y. Liu, X. Xing, H. Cheng, D. Li, Z. Guan, J. Liu, and Q. Wu, "A flexible sharding blockchain protocol based on cross-shard byzantine fault tolerance," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 2276–2291, 2023.

[13] I. A. Obiri, J. Gao, Q. Xia, H. Xia, and C. N. A. Cobblah, " Hiba: Hierarchical High-Performance Blockchain Architecture ," *IEEE/ACM Transactions on Networking*, no. 01, pp. 1–16, Nov. 5555. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TNET.2024.3481488

[14] Z. Hong, S. Guo, E. Zhou, W. Chen, H. Huang, and A. Zomaya, "Gridb: Scaling blockchain database via sharding and off-chain cross-shard mechanism," *Proc. VLDB Endow.*, vol. 16, no. 7, p. 1685–1698, mar 2023. [Online]. Available: https://doi.org/10.14778/3587136.3587143

[15] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-2_Al-Bassam_paper.pdf

[16] P. Zheng, Q. Xu, Z. Zheng, Z. Zhou, Y. Yan, and H. Zhang, "Meepo: Sharded consortium blockchain," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1847–1852.

[17] ——, "Meepo: Multiple execution environments per organization in sharded consortium blockchain," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 12, pp. 3562–3574, 2022.

[18] J. Gao, J. Zhang, Y. Li, J. Hao, K. Wang, Z. Guan, and Z. Chen, "Pshard: A practical sharding protocol for enterprise blockchain," in *Proceedings of the 2022 5th International Conference on Blockchain Technology and Applications*, ser. ICBTA '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 110–116. [Online]. Available: https://doi.org/10.1145/3581971.3581987

[19] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "Ohie: Blockchain scaling made simple," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 90–105.

[20] Y. Jo and C. Park, "A hierarchical blockchain supporting dynamic locality by extending execute-order-validate architecture," *Distrib. Ledger Technol.*, aug 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3688811

[21] GT. (2024, Jan) Cybersecurity update causes worldwide microsoft outages. [Online]. Available: https://www.govtech.com/security/cybersecurity-update-causes-worldwide-microsoft-outages

[22] P. Lipscombe. (2024, Jan) More than 170 base stations offline in taiwan following earthquake. [Online]. Available: https://www.datacenterdynamics.com/en/news/more-than-170-base-stations-offline-in-taiwan-following-earthquake/

[23] Reuters. (2022, Oct.) Pangyo data center fire. [Online]. Available: https://www.reuters.com/world/asia-pacific/fire-knocks-out-services-south-korea-tech-giants-kakao-naver-2022-10-15/

[24] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 34–50. [Online]. Available: https://doi.org/10.1145/3492321.3519594

[25] Google. (2025, Jan.) The go programming language. [Online]. Available: https://go.dev/

[26] (2020) Hyperledger fabric v2.1.0 branch. [Online]. Available: https://github.com/hyperledger/fabric/tree/v2.1.0/

[27] (2021) narwhal. [Online]. Available: https://github.com/facebookresearch/narwhal

[28] Z. Hong, S. Guo, and P. Li, "Scaling blockchain via layered sharding," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 12, pp. 3575–3588, 2022.

[29] 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," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190538

[30] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "Resilientdb: Global scale resilient blockchain fabric," *Proc. VLDB Endow.*, vol. 13, no. 6, p. 868–883, Feb. 2020. [Online]. Available: https://doi.org/10.14778/3380750.3380757

[31] L. N. Nguyen, T. T. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: Optimal transactions placement for scalable blockchain sharding," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2019, pp. 525–535. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDCS.2019.00059

[32] Y. Zhang, S. Pan, and J. Yu, "Txallo: Dynamic transaction allocation in sharded blockchain systems," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2023, pp. 721–733. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDE55515.2023.00390

[33] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, *Shard Scheduler: Object Placement and Migration in Sharded Account-Based Blockchains*. New York, NY, USA: Association for Computing Machinery, 2021, p. 43–56. [Online]. Available: https://doi.org/10.1145/3479722.3480989

[34] X. Qi, J. Jiao, and Y. Li, "Smart contract parallel execution with fine-grained state accesses," in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2023, pp. 841–852. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDCS57875.2023.00068

[35] D. Ryu and C. Park, " Toward High-Performance Blockchain System by Blurring the Line between Ordering and Execution ," in *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2024, pp. 391–406. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41406.2024.00033

[36] Q. Kniep, L. Kokoris-Kogias, A. Sonnino, I. Zablotchi, and N. Zhang, "Pilotfish: Distributed execution for scalable blockchains," 2025. [Online]. Available: https://arxiv.org/abs/2401.16292

[37] Y. Fang, Z. Zhou, S. Dai, J. Yang, H. Zhang, and Y. Lu, "Pavm: A parallel virtual machine for smart contract execution and validation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 1, pp. 186–202, 2024.

[38] 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," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 232–244. [Online]. Available: https://doi.org/10.1145/3572848.3577524

[39] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2705–2718. [Online]. Available: https://doi.org/10.1145/3548606.3559361

[40] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[41] J. Ma, Y. Jo, and C. Park, "Peerbft: Making hyperledger fabric's ordering service withstand byzantine faults," *IEEE Access*, vol. 8, pp. 217 255–217 267, 2020.

[42] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: The partially synchronous version," 2022. [Online]. Available: https://arxiv.org/abs/2209.05633

[43] S. Duan and H. Zhang, "Foundations of dynamic bft," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 1317–1334.

[44] syndtr. (2025, Jan.) Leveldb keyvalue database in go. [Online]. Available: https://github.com/syndtr/goleveldb/

[45] R. Team. (2025, Jan.) Rust: A language empowering everyone to build reliable and efficient software. [Online]. Available: https://www.rust-lang.org/

[46] (2024) Docker. [Online]. Available: https://www.docker.com/

[47] B. Univ. (2019, Dec.) Smallbank benchmark. [Online]. Available: https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[48] C. Yang, K.-W. Chin, J. Wang, X. Wang, Y. Liu, and Z. Zheng, "Scaling blockchains with error correction codes: A survey on coded blockchains," *ACM Comput. Surv.*, vol. 56, no. 6, jan 2024. [Online]. Available: https://doi.org/10.1145/3637224

[49] 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)*. IEEE, 2019, pp. 1337–1347.

Yongrae Jo received the B.S. degree in computer science and engineering from Pusan National University, Republic of Korea, in 2017. He is currently pursuing the Ph.D. degree with the Pohang University of Science and Technology (POSTECH). His research interests include distributed systems and blockchain.

Chanik Park received the B.S. degree in electronics engineering from Seoul National University, Seoul, Republic of Korea, in 1983, and the M.S. and Ph.D. degrees in electronics and electrical engineering (computer engineering) from KAIST, Daejeon, South Korea, in 1985 and 1988, respectively. He was a Visiting Scholar with the Parallel Systems Group, IBM T. J. Watson Research Center, and a Visiting Professor with the Storage Systems Group, IBM Almaden Research Center. He also visited Northwestern University and Yale University, in 2009 and 2015, respectively. Since 1989, he has been working as a Professor with the Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH). His research interests include storage systems, operating systems, system security, and blockchain. He has served as a Program Committee Member at a number of international conferences and workshops.