applied sciences

MDPI

Article

NeuChain+: A Sharding Permissioned Blockchain System with Ordering-Free Consensus

Yuxiao Gao, Xiaohua Li *, Zeshun Peng, Yanfeng Zhang and Ge Yu

School of Computer Science and Engineering, Northeastern University, Shenyang 110169, China; gaoyuxiao@stumail.neu.edu.cn (Y.G.); pengzeshun@stumail.neu.edu.cn (Z.P.); zhangyf@mail.neu.edu.cn (Y.Z.); yuge@mail.neu.edu.cn (G.Y.)

* Correspondence: lixiaohua@mail.neu.edu.cn

Abstract: Permissioned blockchains are widely used in scenarios such as digital assets, supply chains, government services, and Web 3.0, but their development is hindered by low throughput and scalability. Blockchain sharding addresses these issues by dividing the ledger into disjoint shards that can be processed concurrently. However, since cross-shard transactions require the collaboration of multiple shards, blockchain sharding needs a commit protocol to ensure the atomicity of executing these transactions, significantly impacting system performance. To this end, by exploiting the characteristics of deterministic ordering, we propose a cross-shard transaction processing protocol called *cross-reserve*, which eliminates this costly cross-shard coordination while providing the same consistency and atomicity guarantee. Based on the ordering-free execute–validate (EV) architecture, we implemented a blockchain prototype called NeuChain+, which further reduces the cross-shard transaction processing overhead using the pipelined read sets transmission. Experimental results show that NeuChain+ is scalable and outperforms state-of-the-art blockchain systems with 1.7–75.3× throughput under the SmallBank workload.

Keywords: permissioned blockchain; sharding; scalability; deterministic concurrency control; cross-shard transaction

check for updates

Citation: Gao, Y.; Li, X.; Peng, Z.; Zhang, Y.; Yu, G. NeuChain+: A Sharding Permissioned Blockchain System with Ordering-Free Consensus. *Appl. Sci.* **2024**, *14*, 4897. https://doi.org/10.3390/app14114897

Academic Editor: Gianluca Lax

Received: 30 April 2024
Revised: 31 May 2024
Accepted: 3 June 2024
Published: 5 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The applications of permissioned blockchain in scenarios like digital assets [1], supply chains [2], government services [3], and Web 3.0 [4] are becoming increasingly diverse, but the performance of existing blockchains fails to meet the demands of these high-throughput applications [5–8]. For example, Hyperledger Fabric, a widely adopted permissioned blockchain, can only achieve a throughput of around 3.5 kTPS [9]. In contrast, financial applications like Visa [10] require tens of kTPS to handle peak transaction volumes [11]. Permissioned blockchains typically employ consensus protocols to ensure consistent execution results across nodes. However, most consensus protocols (e.g., PBFT [12], Raft [13]) rely on a single leader node to order all client transactions, making it inherently sequential and impossible to parallelize, limiting the overall throughput.

Existing works [14–19] address the single-node bottleneck by improving the scalability of consensus protocols. GeoBFT [20], RCC [21], and ISS [22] run multiple consensus instances concurrently to distribute the workload of the single leader node. Bidl [23] and the research of Nathan et al. [24] parallelize ordering and transaction execution to hide the ordering latency. NeuChain [25] uses deterministic transaction execution to escape strict total ordering, thereby eliminating the need for explicit transaction ordering consensus. By allowing multiple nodes to independently receive requests, the single-node ordering bottleneck is eliminated. Moreover, NeuChain benefits from deterministic concurrency control, enabling asynchronous block generation. However, all transactions must still achieve consensus and be globally replicated before execution can be completed, each node having to execute all transactions.

As we can see, the above approaches, including NeuChain, do not fundamentally address the scalability issue, as transactions must be globally replicated to all nodes. Firstly, since the consensus protocols ensure that each correct node receives a transaction copy to maintain its local ledger, the total network traffic consumption increases as the number of nodes increases. Secondly, full transaction execution and ledger update are performed on each node exclusively, meaning that the throughput does not scale with increase in the total number of nodes. As shown in Figure 1a, we experimented with NeuChain to show the scalability issue (detailed setup in Section 7). When scaling the number of nodes from 4 to 16, the throughput and latency remain unchanged. Thirdly, having each node maintain the complete blockchain ledger creates significant storage overhead. As shown in Figure 1b, the storage consumption of a node in NeuChain reaches 1561 MB after running for only 60 s.

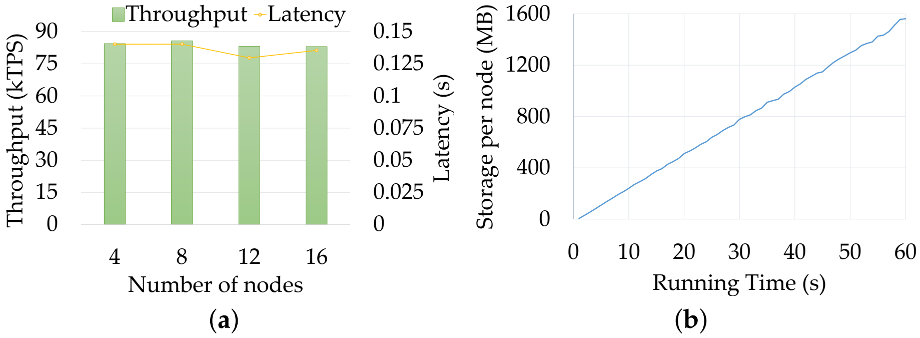


Figure 1. NeuChain's scalability and storage pressure experimental results. (**a**) NeuChain LAN scalability experimental results. (**b**) Single-node storage size during NeuChain running for 60 s.

Blockchain sharding [26] is proposed to solve the above scalability issues, where the ledger is divided into multiple disjoint shards, each maintained by a set of nodes. By distributing workloads across multiple shards, system performance scales linearly with the number of shards. However, to execute cross-shard transactions (i.e., transactions operating on multiple shards), sharding blockchains must employ a cross-shard commit protocol to ensure transaction atomicity, significantly impacting system performance.

A widely adopted [27–31] commit protocol is the two-phase commit [32] (2PC), which requires three rounds of cross-shard communication under the critical path, causing high latency. Moreover, when committing a cross-shard transaction with 2PC, the data record accessed by the transaction cannot be modified by others (e.g., via two-phase locking [33]), further reducing the throughput. Although Monoxide [34] eliminates this cross-shard coordination by using eventual atomicity, it restricts the types of transactions that can be processed, and the latency of cross-shard transactions is also higher than the 2PC approach (detailed in Section 2.1). We propose a deterministic sharding protocol, *cross-reserve* , to address the above scalability issues by exploiting the characteristics of the ordering-free execute–validate (EV) blockchain architecture. We also implement a blockchain prototype called NeuChain+, which leverages deterministic transaction execution to process cross-shard transactions. Since the execution result is deterministic after batching transactions into a block, there is no need to use a commit protocol such as 2PC to determine whether to commit or abort, eliminating the need for multiple rounds of cross-shard coordination. Moreover, remote reads accessing other shards cannot be processed locally when executing cross-shard transactions because each shard only has a partial of the ledger. Although NeuChain+ eliminates the costly cross-shard coordination, the execution of cross-shard transactions cannot start until the cross-shard read sets transmission is complete, causing synchronization waits. To this end, NeuChain+ pipelines the read sets transmission. Transaction execution and read sets transmission can proceed concurrently, further reducing latency.

The main contributions of this paper are as follows:

- We propose a cross-shard transaction processing protocol, *cross-reserve*, which utilizes deterministic concurrency control to avoid costly commit protocols.
- We pipeline and parallelize the read sets transmission during transaction execution to further reduce cross-shard commit latency.
- We propose the sharding—EV architecture, which enhances horizontal scalability while achieving high throughput by eliminating explicit ordering.
- We implement a sharding blockchain prototype, NeuChain+, based on our team's previous work, NeuChain. Experimental results show that the throughput of NeuChain+ is approximately 2.9× that of NeuChain and outperforms other state-of-the-art blockchain systems with 1.7–75.3× throughput under the SmallBank workload.

2. Background and Motivation

In the workflow of blockchain transaction processing, three indispensable phases exist: the ordering phase, the execution phase, and the validation phase. The ordering phase achieves consensus on both intrablock and interblock transaction orders. The interblock order determines which transactions from the transaction pool are included in a specific block, while the intrablock order dictates the sequence in which transactions are executed in this block. The execution phase is responsible for executing transactions and generating read–write sets. The validation phase verifies the validity of blocks to update the local ledger.

Based on the arrangement of these phases, existing blockchain systems can be categorized into four architectures: order–execute–validate (OEV), execute–order–validate (EOV), order–execute–parallel–validate (OEPV), and execute–validate (EV). The following paragraphs provide an introduction and discussion of these architectures.

2.1. OEV Architecture

As shown in Figure 2a, most blockchain systems [26–30,34–39] adopt the OEV architecture, which operates in three steps:

1. After receiving transactions from clients, the consensus leader batches these transactions into a block and determines the execution order of these transactions so that each node can obtain the same results. The consensus leader also pre-executes these transactions to ensure their validity.
2. The block is then replicated to other follower nodes. Since block execution is performed locally, a node must execute the block based on the given order to update its local ledger. Otherwise, if two nodes execute these transactions in a different order, their local ledger copies will be inconsistent.
3. Meanwhile, the node validates the block and its contained transactions to prevent faulty behaviors by the leader node.

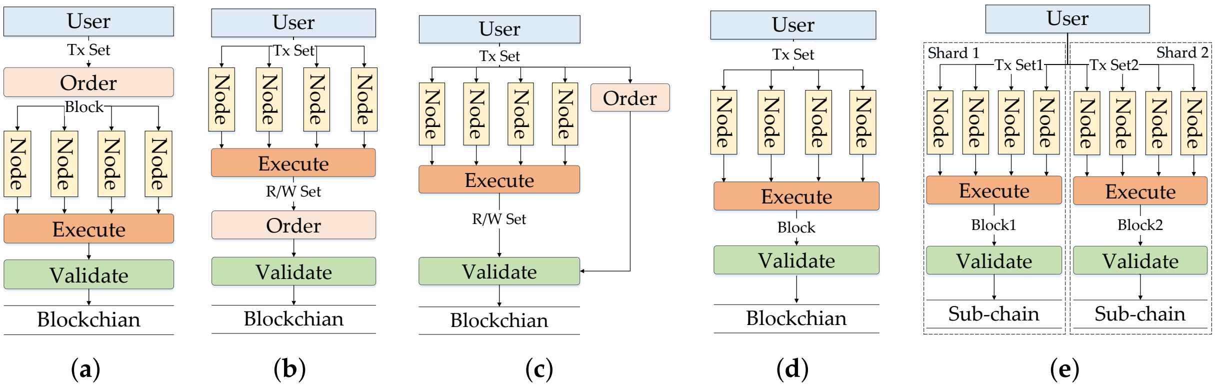


Figure 2. (**a**) Order–execute–validate architecture; (**b**) execute–order–validate architecture; (**c**) order–execute–parallel–validate architecture; (**d**) execute–validate architecture; (**e**) sharding–execute–validate architecture.

Transaction execution in the OEV architecture is performed serially to ensure deterministic results, which could be a performance bottleneck. Blockchains represented by Elastico [26] adopt the sharding technique to solve this bottleneck, allowing parallel transaction execution among shards. However, Elastico does not support cross-shard transactions and state sharding, meaning every node maintains a full copy of the ledger. Other OEV sharding blockchains [27–30,34,37–39] adopt state sharding to solve the above issues. They divide the ledger into shards and use commit protocols represented by 2PC to ensure the atomicity of cross-shard transactions, as shown in Table 1.

Table 1. Comprehensive comparison of blockchain systems.

System	Architecture	Consensus	Shard Fault	Total Fault	Ledger Model	State Sharding	Cross-Shard Transaction	Throughput	Latency
Elastico	OEV	PBFT	33%	25%	UTXO	False	-	48 kTPS	800 s
OmniLedger	OEV	ByzCoinX	33%	25%	UTXO	True	2PC	>100 kTPS	1.38 s
Rapidchain	OEV	50%BFT	50%	33%	UTXO	True	2PC	7.3 kTPS	8 s
Chainspace	OEV	MOD-SmaRt	33%	25%	Object	True	2PC	350 TPS	0.1 s
Monoxide	OEV	Chu-ko-nu	33%	50%	Account/balance	True	Eventual atomicity	11.7 kTPS	15 s
ByShard	OEV	PBFT	33%	33%	UTXO	True	2PC	20 kTPS	1 s
Meepo	OEV	PoA [40]	33%	25%	Account/balance	True	cross-epoc and cross-call	124.6 kTPS	0.52 s
BrokerChain	OEV	PBFT	33%	33%	Account/balance	True	Broker	3 kTPS	14.9 s
PROPHET	OEV	PBFT	33%	33%	Account/balance	True	Reconnaissance coalition	1.2 kTPS	2.5 s
AHL [31]	EOV	AHL	33%	33%	General workload	True	2PC	3 kTPS	18 s
NeuChain	EV	PBFT Raft	-	33%	Account/balance	-	-	85.7 kTPS	0.14 s
NeuChain+	EV	PBFT Raft	33%	33%	Account/balance	True	Cross-reserve	247 kTPS	0.19 s

2PC is widely used as the cross-shard commit protocol, and several variants of 2PC exist [30]. For example, in shard-driven 2PC, cross-shard transaction T involves shard S_1 and S_2. Since T can only be committed if all shards commit T, when S_1 can commit T, it sends the 2PC request to S_2. When receiving the request, S_2 decides whether T can be committed and replies its vote back to S_1. S_1 commits T locally only when all shards can commit T. Otherwise, it aborts T. The decision is also notified to the other shards S_2.

However, this process requires three rounds of communication between shards, and before starting a round, the initializing shard must reach a consensus on its decision, resulting in significant communication overhead. Meanwhile, as the commit decision cannot be changed after achieving consensus, the corresponding data records cannot be modified by another transaction during 2PC [33], significantly limiting the performance of transaction execution.

For other protocols, Monoxide [34] proposes the eventual atomicity to handle cross-shard transactions. When transferring money between accounts, the transaction is guaranteed to be committed after the source account is debited, as adding money to the target account must not violate consistency. Therefore, 2PC is not needed for such types of transactions. However, since faulty shard leaders may refuse to accept these “relay” transactions, cross-shard transactions may require several blocks to be committed, resulting in increased latency. To reduce this unlimited latency, BrokerChain [38] proposes a duration-limited eventual atomicity to ensure that the execution of cross-shard transactions is completed in a known upper bound. However, it is still only applicable to specific types of transactions.

To handle complex smart contract transactions, Meepo [37] proposes cross-epoch and cross-call to execute cross-shard transactions. Regardless of how many shards the cross-shard transaction involves, it can be committed in a single round of consensus. However, its coordination rounds depend on how many shards are involved in the transaction. When executing transactions accessing multiple shards, the execution time of each round increases.

Moreover, if transactions abort during contract execution, the entire block must be rolled back and re-executed, incurring significant overhead.

PROPHET [39] proposes a deterministic ordering algorithm for Byzantine fault-tolerant consensus to handle cross-shard transactions, which is close to our work. It deterministically assigns a global order to transactions based on pre-execution results, thereby eliminating transaction conflicts. However, it requires a costly pre-execution phase that still requires multiple rounds of coordination between shards to generate transaction read–write sets. Furthermore, it still maintains an explicit ordering phase.

2.2. EOV Architecture

The EOV architecture [9] is shown in Figure 2b. By using the optimistic concurrency control [41] (OCC), transaction execution is performed concurrently, significantly improving the throughput. The workflow of EOV is described as follows:

1. A transaction is executed based on a snapshot of the ledger and obtains a read–write set. Since the ledger is not modified during the execution phase, transactions can be executed concurrently and distributed evenly to all nodes (i.e., different nodes execute disjoint transaction sets).
2. The executed transactions are batched into blocks to obtain a global consistent order.
3. Similar to the OEV architecture, transactions are validated in serial order to ensure consistent results. If records read by a transaction have been modified (stale read), the transaction must abort. Otherwise, its write set is applied to the ledger.

Although the EOV architecture addresses the serial execution issue of the OEV architecture through OCC and improves the parallelism, most blockchains based on the EOV architecture use PBFT consensus, which has been proven [42] to be insufficient in scalability due to communication overhead. Thus, sharding technology has also been introduced into EOV architecture, such as AHL. However, the transaction execution results are still nondeterministic, and transactions have to be validated in serial order. Therefore, it also uses 2PC to process cross-shard transactions through costly multiple rounds of coordination.

2.3. OEPV Architecture

As illustrated in Figure 2c, the OEPV architecture [24] utilizes serializable snapshot isolation [43] (SSI) to parallelize the ordering and execution phases of the EOV architecture without affecting the execution results. Transactions are ordered while executing on the ledger snapshot. After that, they are verified and committed based on the ordering results.

Although OEPV improves the parallelism of the blockchain system, it does not change the fact that the ordering phase still exists. Nodes must wait for the ordering server to provide the transaction intrablock order before committing or aborting transactions.

2.4. EV Architecture

In Figure 2d, the ordering-free EV architecture differs from the architectures above. It eliminates the explicit ordering phase through implicit deterministic ordering, thus overcoming the bottleneck of centralized ordering. Its workflow is summarized as follows:

1. All nodes independently receive user transaction requests and then exchange transactions through multiple parallel consensus instances. When a node collects the complete transaction set from all other nodes, it executes these transactions according to the deterministic rule and generates blocks.
2. After completion of execution, the execution results are returned to the user, and signatures verifying the execution results are exchanged among nodes.

Instead of ordering transactions with a single consensus leader, all nodes in the EV architecture can propose blocks concurrently, eliminating the single-node performance bottleneck. Although transactions from different nodes arrive out of order, by using the deterministic concurrency control, executing these transactions out of order can still obtain the same result without internode coordination, eliminating the centralized ordering bottleneck. However, as discussed in Section 1, the EV architecture faces scalability issues in network, storage, and transaction execution: (1) Transactions must be replicated to all nodes via consensus. (2) Each node must execute a complete transaction set. (3) The storage overhead of maintaining a complete ledger is also significant.

Insight. The EV architecture employs deterministic concurrency control, which can determine the order of transactions without additional coordination. Therefore, when employing blockchain sharding to the EV architecture, the order of cross-shard transactions can be determined in advance, and no additional coordination is required during transaction execution, eliminating the costly commit protocols. Therefore, we propose the sharding–EV architecture based on the EV architecture, as shown in Figure 2e.

3. NeuChain+ Design

3.1. System Model

NeuChain+ adopts the account/balance model. Accounts are evenly divided into multiple shards. Nodes within each shard maintain a partial of the ledger. They also store all intrashard and cross-shard transactions that access records belonging to their shard. Like other sharding permissioned blockchains [30,31,37,38], the sharding strategy and organizational structure are known to all nodes.

Network model. NeuChain+ assumes the partial synchrony model [44]. The network is mostly synchronized, but unstable and asynchronous scenarios can also happen. When the network is unstable, an unknown global stabilization time (GST) exists, during which the system operates asynchronously. Once the GST is reached, the system transitions to a synchronous state. At this time, messages are guaranteed to arrive within Δ.

Threat model. NeuChain+ adopts the Byzantine fault-tolerance (BFT) model, where faulty nodes can behave arbitrarily. We denote the number of faulty nodes in a shard as f and the minimum number of nodes in this shard as $3f + 1$. NeuChain+ adopts a public-key infrastructure (PKI), where each node has a public–private key pair for signing and verifying messages.

3.2. System Overview

As shown in Figure 3, NeuChain+ is design based on the sharding–EV architecture. There are three types of components in NeuChain+: client proxy, epoch server, and block server. Client proxy is responsible for batching transactions into blocks and block replication. Epoch server provides a global consistent batch timeout to prevent clock skewing among client proxies. Block server executes blocks deterministically and updates its local ledger. These components are discussed in detail below.

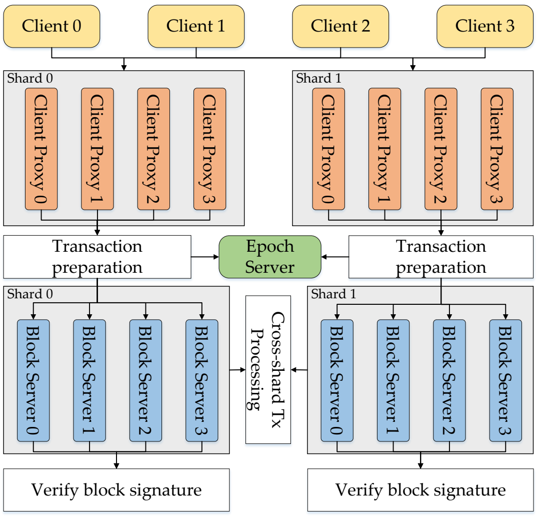


Figure 3. Overall system architecture.

Client proxy. The client proxy receives transactions involving its shard from clients. To determine which block the transactions belong to, it batches them and requests an epoch number from the epoch server. Each transaction is then assigned a globally unique transaction ID *tid* for deterministic intrablock ordering. The client proxy then uses atomic broadcast (e.g., PBFT or Raft) to exchange its transactions with other intrashard client proxies, ensuring the integrity and consistency of transactions. Notably, the client proxy acts as both the leader of the broadcast instance initiated by itself and as the follower of other broadcast instances initiated by other client proxies within the same shard.

Epoch server. The epoch server generates monotonically increasing epoch numbers and assigns them to transaction batches generated by client proxies. The epoch server can be a trusted single node or a fault-tolerant cluster containing multiple epoch servers. Notably, only the increment of the epoch number requires consensus, and the epoch number assignment can be performed concurrently with this consensus to reduce latency.

Block server. The block server only stores transactions that access records on its local shard, forming a subchain. A block server corresponds with a client proxy that accepts transactions. It uses deterministic concurrency control to execute transactions concurrently to generate read–write sets for validation. Each block server maintains a shared read–write reserve table, which only reserves transactions that can be committed in the current *epoch*, ensuring a serializable isolation level. After execution, the cross-shard transactions use *cross-reserve* (detail in Section 5) to guarantee atomicity.

3.3. System Workflow

The workflow of NeuChain+ consists of three phases, the preparation phase, the execution phase, and the validation phase, as shown in Figure 4.

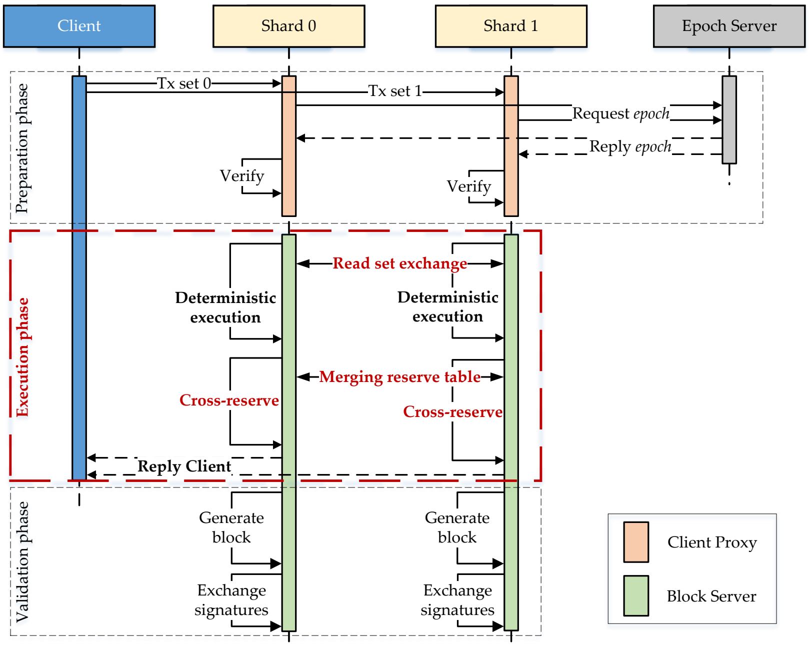


Figure 4. Transaction processing workflow of NeuChain+.

3.3.1. Preparation Phase

Since transactions may involve multiple shards, clients send transactions to the specified client proxies based on records of the transactions accessed. For example, if transaction T modified records in both shard S_1 and shard S_2, T is sent to the two shards for execution. Notably, although NeuChain+, like NeuChain, benefits from deterministic execution without the need to predict the read–write set, to accurately send cross-shard transactions to the involved shards, we assume that the shards involved in a transaction are known. Clients send transactions to client proxies within a shard in a round-robin manner to achieve load balancing.

When enough transactions are received or timeout is triggered, a client proxy batches its received transactions and requests an epoch number from the epoch server. To ensure the validity of the epoch number assignment, the epoch server signs the epoch number and the hash of the transaction batch before returning to the client proxy. Multiple batches with the same epoch number can exist. Therefore, when receiving an epoch number greater than the previous one, a client proxy can determine that the current epoch is finished. Subsequently, it starts exchanging its generated batches with other intrashard client proxies. When receiving all batches within the same epoch from all intrashard client proxies, the client proxy deterministically calculates the IDs of transactions in these batches and forwards the batches to the specified block server for execution. Notably, client proxies only replicate transactions within their respective shards, without any further coordination.

3.3.2. Execution Phase

As shown in the red-marked part in Figure 4, this phase is the key part of sharding based on EV architecture. When receiving the complete transaction set of the current epoch from the specified client proxy, the block server executes these transactions deterministically according to their transaction IDs. To execute cross-shard transactions accessing records from other shards, the block server generates their read–write sets through read sets transmission (detailed in Section 4). We also adopt deterministic rules to handle cross-shard transaction conflicts. In detail, after executing and resolving conflicts of intrashard transactions, cross-shard transactions are committed by using *cross-reserve* (detailed in Section 5) to ensure atomicity. After executing and validating all transactions, the execution results are immediately returned to clients, and the local ledger is also updated with the written sets of the committed transactions.

3.3.3. Validation Phase

To ensure the authenticity of the execution results, block servers within a shard sign their blocks (containing the transaction requests and the execution results) and exchange their signatures to prevent Byzantine nodes from generating faulty blocks. Since the transaction input has already achieved consensus, the output of execution results is valid when it has $f + 1$ signatures.

4. Deterministic Transaction Execution

4.1. Read Sets Transmission

For cross-shard transactions, some of them cannot be executed directly locally but require the read set from remote shards. We denote such transactions as transactions with remote read–write dependencies. This requires the read sets transmission between shards. For state sharding, this communication overhead is inevitable. A straightforward idea is to send a request to the required shard when such a transaction is executed, requesting the required read set. For example, there is transaction T that updates a record in shard S_1, and the update depends on a read in shard S_2. When the block server in S_1 executes T, it will send a read request to S_2. After receiving the request, S_2 will return directly if T has been executed. Otherwise, it will wait for T to be executed and generate a read set to return.

In NeuChain+, when the block server executes a transaction with remote read–write dependencies, it caches its *tid* and the corresponding remote shard, waits for a batch of transactions to process, and then sends a request to each block server in the remote shard. After receiving the request, if all transactions in the batch complete execution, it will directly package all read sets into a batch and return it. If executions have not been completed, it needs to block and wait until the executions are completed. After receiving the read set, it wakes up the pending transaction to complete execution.

4.2. Pipeline Optimization

Firstly, requesting the read set from the remote shard may incur a complete RTT (round-trip time). Secondly, synchronous blocking is required to wait for the receipt of $f + 1$ read sets from the corresponding shard to ensure the validity of the read sets. At this synchronization point, it is necessary to wait for communication between nodes, thereby introducing delays in the overall progress. Therefore, NeuChain+ uses pipeline technology to overlap communication with execution, reducing the communication latency of 0.5 RTT while avoiding blocking waiting.

Algorithm 1 describes the optimized read sets transmission process. During the read sets transmission process, we define the shard that generates a read set as the reading shard and the shard that requires a read set to generate a write set as the writing shard. When executing cross-shard transactions with remote read–write dependencies, the reading shard directly sends the read set to the writing shard (Lines 1–3). Note that epoch n must be attached to the read set when sending to ensure that both the reading and writing shards are synchronized. In the writing shard, a hash table RR (Line 5) is initialized to cache the read sets sent by other shards in the structure of <$account, balance$>. After receiving $f + 1$ of the same read set, the writing shard stores it into RR and wakes up the pending transaction (Lines 6–8). When the writing shard executes the corresponding transaction, it can directly read from the hash table according to the required $account$ (Lines 9–11). When the corresponding $account$ in RR is empty, it proves that no verifiable read set has been received yet, and the transaction will be pending and other transactions will continue to be processed (Lines 12–13).

Algorithm 1 Pipelining read sets transmission.

Input: The reading shard state database snapshot $DB[n-1]$, the cross-shard transaction T with remote read-write dependencies of epoch n

Output: The read-write set $\{T.RS, T.WS\}$

```
 1: function EXECUTE($T$)                                            ▷ In reading shard
 2:     $\{T.RS, T.WS\} \leftarrow$ execute operation of $T$ base on $DB[n-1]$;
 3:     Send $T.RS$ with epoch $n$ to all Block Servers in writing shard;
 4: function READSETRECEIVER($T$)                                    ▷ In writing shard
 5:     Initialize reading result $RR$;
 6:     if receive the same $T.RS$ over $f + 1$ then
 7:         $RR[T.RS.key] = T.RS.value$;
 8:         $T$.notify();
 9: function EXECUTE($T$)                                            ▷ In writing shard
10:     if $RR[account_{req}]$ != NULL then                          ▷ if $T.RS$ received
11:         $T.WS \leftarrow$execute operation of $T$ base on $DB[n-1]$ and $RR[account_{req}]$;
12:     else
13:         $T$.wait();
```

5. Cross-Shard Transaction Processing Protocol

This section introduces the cross-shard transaction processing *cross-reserve* of NeuChain+. It ensures the atomicity of cross-shard transactions. Unlike 2PC, which requires three rounds of coordination, it only takes one round of broadcast to achieve consensus. Furthermore, it also ensures that cross-shard transactions can be committed within a single epoch, distinct from the eventual atomicity. Section 5.1 uses the merging reserve table to enable each block server to obtain a complete reserve table containing accounts involved in cross-shard transactions. Section 5.2 performs deterministic concurrency control on transactions.

5.1. Merging Reserve Table

The reserve table is a crucial data structure for deterministic concurrency control, implemented using a hash table. After executing a transaction, the reserve table is updated in the form of <*account*, *tid*>. When multiple transactions access the same *account*, the reserve table will be updated according to the deterministic rule. This deterministic rule is expressed in detail below.

As illustrated in Algorithm 2, whenever the new epoch n enters the execution phase, the read–write reserve table is first initialized (Lines 1–2) and the cross-shard read–write lists RL and WL implemented using dynamic arrays are initialized (Lines 3–4), used to cache the account involved in the cross-shard transactions. Subsequently, multiple parallel worker threads reserve the transactions executed in Section 4. The worker analyzes the transaction read–write sets to reserve, and the transaction with the minimum *tid* in the reserve table is reserved according to deterministic rules (Lines 6–7, 10–11). Note that the accounts involved in cross-shard transactions need to be stored in the cross-shard read–write lists for *cross-reserve* (Lines 8–9, 12–13).

Algorithm 2 Merging reserve table based on deterministic execution.

```
Input: The read-write set $\{T.RS, T.WS\}$
Output: A complete reserve table containing accounts involved in cross-shard transactions
 1: Initialize read reserve table $RT$;
 2: Initialize write reserve table $WT$;
 3: Initialize cross read list $RL$;
 4: Initialize cross write list $WL$;
 5: function RESERVE($T$)                                  ▷ After $T$ completes execution
 6:     for $w \in T.WS$ do
 7:         $WT[w.key]$ = min($WT[w.key]$, $T.tid$);
 8:         if $T$ is cross-shard transaction then
 9:             $WL$.insert($w.key$);
10:     for $r \in T.RS$ do
11:         $RT[r.key]$ = min($RT[r.key]$, $T.tid$);
12:         if $T$ is cross-shard transaction then
13:             $RL$.insert($r.key$);
14: function MERGINGSENDER( )             ▷ After completing all intra-shard reservations
15:     for $key \in RL$ do
16:         $batch.RT$.insert(<$key$, $RT[key]$>);
17:     for $key \in WL$ do
18:         $batch.WT$.insert(<$key$, $WT[key]$>);
19:     Broadcast $batch$ to all other shard;
20: function MERGINGRECEIVER( )
21:     if receive the same $batch$ of a shard over $f + 1$ then
22:         for $r \in batch.RT$ do
23:             $RT[r.key] = r.value$;
24:         for $w \in batch.WT$ do
25:             $WT[w.key] = w.value$;
```

Although the execution results of cross-shard transactions are deterministic from a global perspective, from the perspective of a block server, it cannot be determined whether the cross-shard transactions can be reserved in other shards. We enable the block server to hold a complete reserve table containing accounts involved in cross-shard transactions by merging reserve tables. This allows block servers among different shards to deterministically commit or abort cross-shard transactions through concurrency control (detailed in Section 5.2). We describe the merging reserve table in detail below.

After completion of the execution of transactions in the current *epoch*, package the rows in the reserve table corresponding to the accounts in a batch and broadcast it to all other shards (Lines 14–19). After receiving $f + 1$ of the same batches from other shards, the accounts involved in cross-shard transactions from remote shards are cached in the local reserve table (Lines 20–25). Since the accounts among shards do not overlap, the keys corresponding to the accounts involved in transactions from other shards in the reserve table of this shard must be empty, so there is no need to judge whether to reserve them based on deterministic rules.

It is not difficult to find that there is cross-shard communication during merging reserve tables, but this is different from the coordination of 2PC. 2PC needs to make decisions through coordination between shards, and decision making means the need for expensive intrashard consensus. Whether cross-shard transactions in NeuChain+ can be committed is deterministic without decision making. Therefore, it only needs a simple exchange of results. Even from the communication overhead perspective, 2PC's three coordinations require 1.5 RTT, while ours with only one broadcast requires 0.5 RTT.

5.2. Transaction Conflict Detection

NeuChain+ transactions are executed based on serializable snapshot isolation [43] (SSI), allowing multithreaded concurrent execution within each block server. To ensure the serializable isolation level, it is necessary to perform concurrency control for potential transaction conflicts. After the above processing, each block server holds a complete reserve table containing accounts involved in cross-shard transactions. NeuChain+ performs conflict detection based on the reserve table and commits or aborts transactions based on deterministic concurrency control rules.

As shown in Table 2, according to the different serializable orders of transaction execution in the same epoch, the potential conflicts are categorized into three types: write-after-write (WAW) dependency, read-after-write (RAW) dependency, and write-after-read (WAR) dependency. There are two transactions T_1 and T_2, which are ordered by *tid* such that T_1 precedes T_2 in a serializable order. As described in Table 2, WAW and RAW may cause errors, since it is necessary to abort a transaction to remove the dependency.

Table 2. Transaction conflict type analysis.

Type	Description	Error Case
WAW	Transactions T_1 and T_2 simultaneously write to *account*.	Concurrent writes by T_1 and T_2 lead to an indeterminate final result.
RAW	T_1 writes to *account*, followed by T_2 reading from *account*.	Due to snapshot-based reading, T_2 reads outdated data.
WAR	T_1 reads from *account*, followed by T_2 writing to *account*.	None.

As illustrated in Algorithm 3, based on the reserve table, NeuChain+ detects conflicts between transactions and deterministically commits or aborts transactions through concurrency control. If the write operation of transaction T involves an account that has been overwritten by a transaction with a smaller *tid* (WAW dependency), then transaction T is aborted (Lines 3–4). If the read operation of transaction T involves an account that has been modified by a transaction with a smaller *tid* (RAW dependency), then T also needs to be aborted (Lines 6–7). All other transactions (WAR dependency and no dependency) are committed (Line 8).

However, for the transactions with RAW dependency but no simultaneous WAR dependency, it can be ensured that there are no cyclic dependencies among transactions. Therefore, such transactions can be committed by reordering [45] transactions. For example, there are three transactions T_{1-3} (the sequence number is the serializable execution order) and accounts A, B, and C. T_1 reads A first and then writes B, T_2 reads B first and then writes C, and T_3 reads B first and then reads C. After detecting conflict (Algorithm 3), T_2 and T_3 have a RAW dependency on T_1, and T_3 also has a RAW dependency on T_2. These three transactions are converted into WAR dependencies by reordering, and all are committed. Another situation is that T_1 reads A first and then writes B, T_2 writes A first and then reads C, and T_3 reads B first and then writes C. After detecting conflict, T_3 has RAW dependence on T_1 and WAR dependence on T_2, and T_2 has WAR dependence on T_1. In this case, if we reorder the transactions to convert T_3's RAW dependence on T_1 into WAR dependence, there are also WAR dependencies between T_3 on T_2 and T_2 on T_1, forming a circular dependency. If executed in the order of T_3, T_2, and T_1 according to reordering, there will be RAW dependencies of T_2 on T_3 and T_1 on T_2. Therefore, this situation cannot be addressed by reordering, and T_3 needs to be deterministically aborted to commit T_1 and T_2.

Algorithm 3 Detect conflict.

```
Input: A complete reserve table containing accounts involved in cross-shard transactions,
    A set of transactions $TS$ of epoch $n$
Output: The commit set and abort set of transactions $TS$ of epoch $n$
for $T \in TS$ do
    for $w \in T.WS$ do
        if $WT\ [w.key] < T.tid$ then
            Abort $T$ and continue;
    for $r \in T.RS$ do
        if $WT\ [r.key] < T.tid$ then
            Abort $T$ and continue;
    Commit $T$;
```

An example of the transaction of account amalgamating. It is convenient for readers to better understand the entire processing of cross-shard transactions. This is a detailed description of the transaction of account amalgamating as an example. In this example, there are two shards: Shard S_0 containing accounts A, B, and C, and Shard S_1 containing accounts D and E. As depicted in Figure 5, the transaction set to be sent by the client includes the transaction amalgamate(C, D) and other intrashard transactions. Amalgamate(C, D) is aimed at merging accounts C and D. It comprises four operations, referred to subsequently as Op_{1-4}, which are to obtain the balance of C, obtain the balance of D, update the balance of C to 0, and update the balance of D to the sum of the original balances of C and D. The client divides the complete transaction set into two subtransaction sets and sends them to S_0 and Shard S_1, respectively. After the preparation phase, amalgamate(C, D) is assigned to the Epoch N and is also assigned a unique *tid*. During the execution phase, the intrashard block server executes this transaction deterministically. It is noteworthy that for cross-shard transactions like amalgamate(C, D), operations are executed based solely on the accounts contained within the local shard. Specifically, S_0 executes only Op_1 and Op_3, while S_1 executes only Op_2 and Op_4. However, since Op_4 depends on the read set of Op_1, it cannot be executed directly, but if Op_1 has completed execution and sent the read set to S_1, then it can directly read the prereceived read set from the local cache for executing (according to Algorithm 1). Otherwise, it is temporarily pended and continues to execute other transactions. After the deterministic execution is completed, *cross-reserve* first generates the intrashard reserve table (according to Algorithm 2, Lines 5–13, the bold words in Figure 5). In this example, S_0 reserves the update of C from amalgamate(C, D), while S_1 reserves the update of D from the other transaction. After merging reserve tables of *cross-reserve* (according to Algorithm 2, Lines 14–25), the reserve tables in both shards have the same row of C and D (the gray words in Figure 5). S_0 and S_1 conduct conflict detection based on the complete reserve table (according to Algorithm 3). According to the deterministic rule, this transaction exists with a WAW dependency. As a result, amalgamate(C, D) ultimately is aborted.

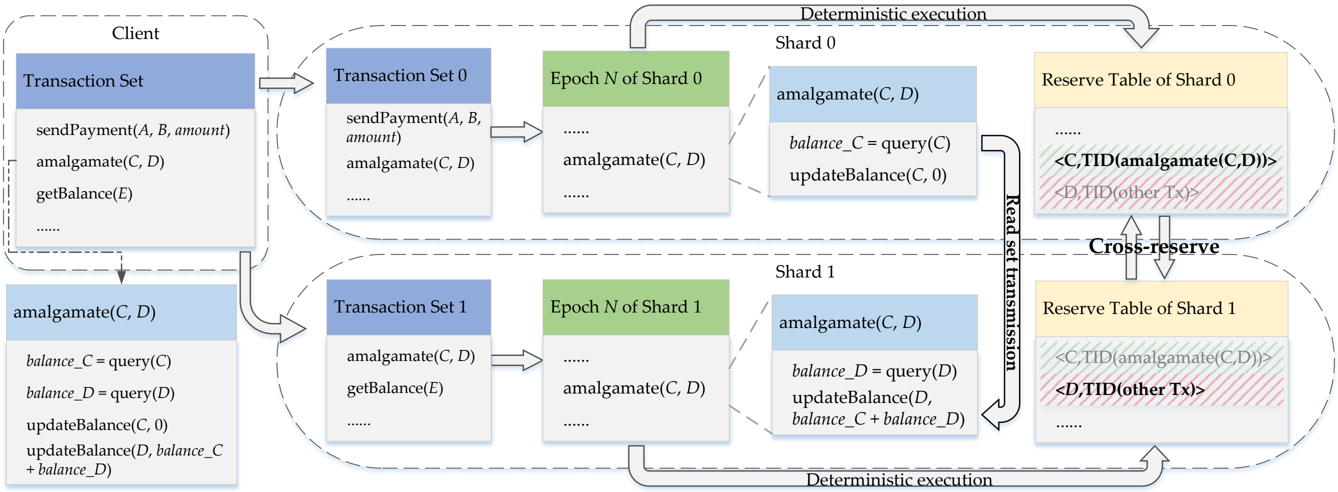


Figure 5. A cross-shard transaction processing with remote read–write dependencies.

6. Analysis

In this section, in order to prove the efficiency of cross-shard transaction processing in NeuChain+ at the theoretical level, we analyze the time complexity of Algorithms 1–3, proposed above, in Section 6.1. In Section 6.2, we analyze the newly proposed cross-shard processing protocol *cross-reserve* and prove its security and liveness.

6.1. Time Complexity Analysis of Algorithms

We define the total number of transactions in the system as n, the number of cross-shard transactions as n_c, the number of cross-shard transactions with remote read–write dependencies as n_d, and the total number of shards as s. On average, each transaction involves m read–write operations, and on average, each transaction involves m_r read operations. In Algorithm 1, the block server needs to perform read sets transfer for all cross-shard transactions with remote read–write dependencies in the system. Therefore, its time complexity is $\mathrm{O}(m_r n_d/s)$. In Algorithm 2, the block server first needs to reserve the read–write sets in the local shard after the transaction is executed. The time complexity of this part is $\mathrm{O}(mn/s)$. The block server then performs the merging of reserve tables among all shards and exchanges reserve tables corresponding to the read–write sets of cross-shard transactions. The time complexity of this part is $\mathrm{O}(mn_c/s)$. Therefore, the time complexity of Algorithm 2 is $\mathrm{O}(mn/s)$. In Algorithm 3, the block server performs conflict detection on the read–write set of the intrashard transactions and the cross-shard transactions involving the local shard to determine whether the transaction can be committed. Therefore, the time complexity of Algorithm 3 is $\mathrm{O}(mn/s)$.

6.2. Cross-Reserve Security and Liveness Analysis

We propose a new cross-shard processing protocol called *cross-reserve*, which could bring potential security and liveness risks. If the block server is a malicious node, *cross-reserve* may be subject to malicious influences. The malicious block server could potentially carry out the following types of attacks through *cross-reserve*:

- Send the tampered read sets or reserve table to other shards to affect security;
- Perform no read sets transmission or merging of reserve tables to some or all block servers to affect liveness;
- Do not follow deterministic rule reserve or malicious conflict detection to tamper with the local blocks to affect security.

Whether in the process of reading sets transmission or merging reserve tables, we require the receipt of at least $f + 1$ of the same read sets or reserve tables to consider them valid. Since each shard contains a total of f malicious nodes, the use of signature verification ensures that malicious nodes cannot launch replay attacks by sending multiple malicious messages. Therefore, the malicious nodes can send at most f malicious messages. If f + 1 consistent messages are received, it can be inferred that at least one honest node has sent the correct message, and if the other f messages are consistent with the correct message, then it can be verified that all f + 1 messages are correct. Therefore, the first type of malicious attack on security is prevented.

Additionally, there are a total number of $3f$ + 1 nodes in each shard. After excluding f malicious nodes and f crash nodes, a node can always receive the correct message from f + 1 honest nodes. Therefore, the second type of attack on liveness can also be prevented.

Intrashard block servers exchange block signatures during the validation phase to ensure the verifiability of the execution results. This means that only blocks with f + 1 signatures are considered valid after verification. Therefore, the blocks tampered with by the last type of attack cannot be validated.

7. Evaluation

Implementation. We implemented a NeuChain+ prototype based on NeuChain [46] with C++14. We leveraged the braft [47] library for intrashard consensus. We employed the RSA digital signature and SHA256 to ensure data integrity and authenticity of messages. We employed key-value database LevelDB [48] as the state database and the file system to store blocks. The state database maintains the ledger view of the current epoch of the shard subchain, and transaction execution is based on this view snapshot. We employed ZeroMQ [49] and protocol buffers [50] to implement message transport between physical nodes. We used Blockbench [42] as the test tool. We also leveraged pipelining and batching optimizations [25] to enhance performance.

Competitors. We compared NeuChain+ to state-of-the-art systems, including NeuChain [25], Fabric [9], Fabric# [51], FastFabric [52], ResilientDB [20], and a BFT-distributed database, Basil [53]. Moreover, to show the efficiency of our sharding strategy, we also compared NeuChain+ with a sharding permissioned blockchain, Meepo [37].

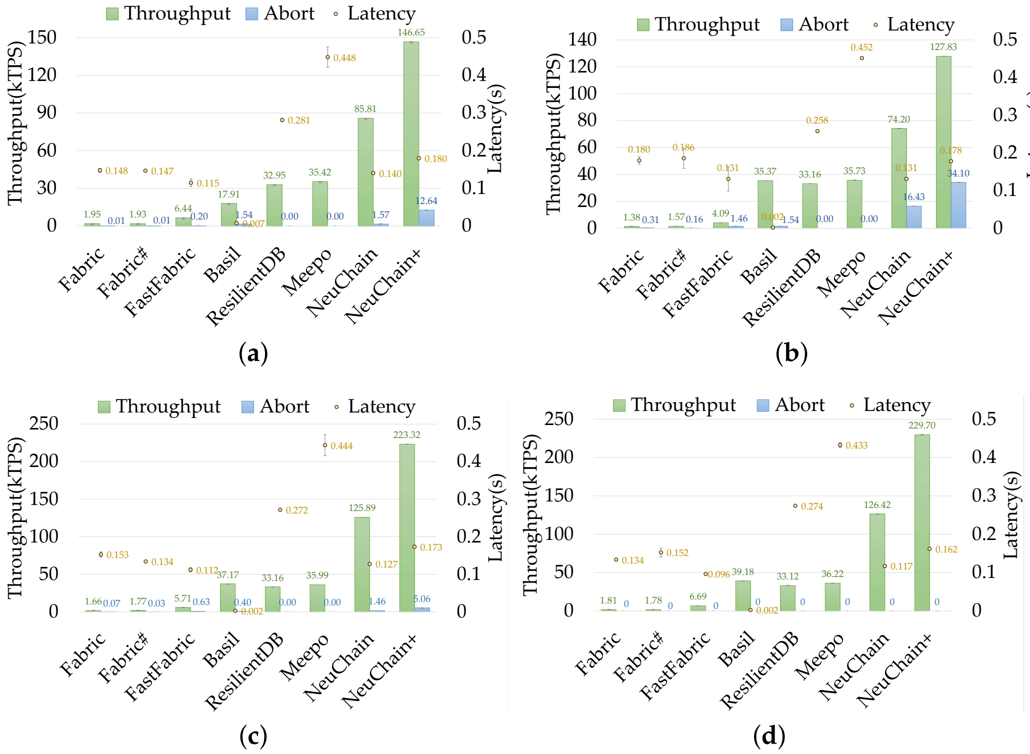
Setup. We deployed 12 nodes on Aliyun [54] in the same data center. Each node is an ecs.hfc6.4xlarge instance with an Intel Xeon Platinum 8269CY 3.1GHz 16-core processor, 32GB RAM, and 10 Gbps bandwidth. In Fabric, Fabric#, and FastFabric, four nodes are orderers, and eight nodes are peers. Basil requires running on $5f + 1$ nodes; hence, we deployed Basil on 12 nodes. For ResilientDB, we deployed it on eight nodes. For Meepo, we deployed it on eight nodes. Due to the trustworthiness within the consortium member, each shard can deploy on a single node. In NeuChain+ and NeuChain, each client proxy corresponds to one block server and runs on the same node. We deployed four nodes for epoch servers and eight for client proxies and block servers. By default, NeuChain+ runs on two shards, and each shard is deployed on four nodes of client proxies and block servers.

Workload. We employed two of the most popular workloads: SmallBank [43] and YCSB [55]. The SmallBank workload is a benchmark for online transaction processing (OLTP) workloads. It consists of three tables (account, checking, and saving). It simulates various fundamental transactions in banking applications, including six types of transactions: transfer, query, account amalgamating, check writing, saving update, and checking update. It is configured with 100,000 accounts, and the transactions follow a uniform distribution. The YCSB workload uses a table containing 1,000,000 records, with each record consisting of 10 attributes, each 100 bytes in size. Read operations access entire records, while write operations access only one attribute of a single record at a time. The experiments used YCSB-A (50% read operations, 50% write operations), YCSB-B (95% read operations, 5% write operations), and YCSB-C (100% read operations). All YCSB transactions followed a Zipf distribution with a skew factor of 0.99. The cross-shard transaction rate depends on the number of shards, which was 16.6%, 22.2%, and 25% under two, three, and four shards, respectively.

7.1. Overall Performance

In order to evaluate the overall improvement of system performance by sharding, NeuChain+ compares the peak throughput, corresponding abort rate, and latency with its competitors under the workloads of SmallBank, YCSB-A, YCSB-B, and YCSB-C. The experimental results are the average of three experiments.

As shown in Figure 6, benefiting from the scalability of sharding and the efficient cross-shard transaction processing protocol *cross-reserve*, the throughput of NeuChain+ reached 146.65 kTPS under SmallBank workload, which is approximately 1.7× that of NeuChain, far exceeding other comparison systems. In state-sharding blockchains [27–31,34,37–39], there are two unavoidable sources of communication overhead resulting in higher latency compared to non-sharding blockchain systems. One is that cross-shard transactions need to obtain the required read sets from remote shards, and the other is that cross-shard transactions require intershard coordination to make decisions on whether to commit. However, NeuChain+ eliminates intershard coordination (Section 5) and optimizes read set transmission (Section 4.2), achieving a latency of approximately 0.18 s under the SmallBank workload, slightly higher than NeuChain but still at the same level as the most state-of-the-art nonsharding blockchains and much lower than the sharding blockchain Meepo. The abort rate of NeuChain+ is higher than that of other blockchains, with approximately 8% of transactions being aborted under the SmallBank workload, due to two reasons. First, in the SmallBank workload, over 80% of operations are write operations. Under deterministic concurrency control, some transactions are deterministically aborted, which will inevitably increase the abort rate while providing excellent throughput. Second, after sharding, the total amount of accounts is divided evenly among shards, which increases the probability of transaction conflicts. Notably, Basil is essentially a BFT-distributed database. Since it does not need to generate blocks, transactions can be returned immediately after execution, unlike blockchain systems that must wait for the entire block's transactions to be executed before returning. As a result, its latency is only 0.007 s under the SmallBank workload.



(**a**) (**b**) (**c**) (**d**)

Figure 6. Throughput and latency performance comparison. (**a**) SmallBank workload; (**b**) YCSB-A workload; (**c**) YCSB-B workload; (**d**) YCSB-C workload.

Under the YCSB-A workload, although the number of records is 10× that of SmallBank and the proportion of write operations is lower than SmallBank, the use of Zipf distribution creates hot records. As a result, the system's abort rate increases. Under the YCSB-B workload, the abort rate significantly decreases due to the mere 5% write proportion of operations. Additionally, because read operations are considerably lighter than write operations, the throughput of each system increases while the latency decreases. Under the YCSB-C workload, since all transactions are read-only, no transaction will be aborted due to conflicts. In NeuChain+, the absence of transactions with remote read–write dependencies eliminates the need for the read set transmission, resulting in significantly reduced latency compared to other workloads.

7.2. Varying Transaction Arrival Rates

To evaluate the peak performance improvement brought by sharding to NeuChain under the same number of nodes, we observe changes in the throughput and latency by varying the transaction arrival rate under the SmallBank workload.

The experimental results are shown in Figure 7. With the increase in the transaction arrival rate, the throughput of the two systems increases linearly. Due to the deterministic parallel execution, the latency only increases slowly with the increase in the transaction arrival rate before reaching the peak performance. NeuChain first reached its peak performance at 87.35 kTPS, corresponding to a throughput of 85.7 kTPS. It can be observed that as the transaction arrival rate continues to increase, transaction accumulation occurs due to slow execution, resulting in a significant increase in latency, with throughput almost plateauing. This is because after reaching the peak performance, a higher transaction arrival rate will cause transactions that exceed the system's processing capacity to accumulate in the pending queue, significantly increasing the latency.

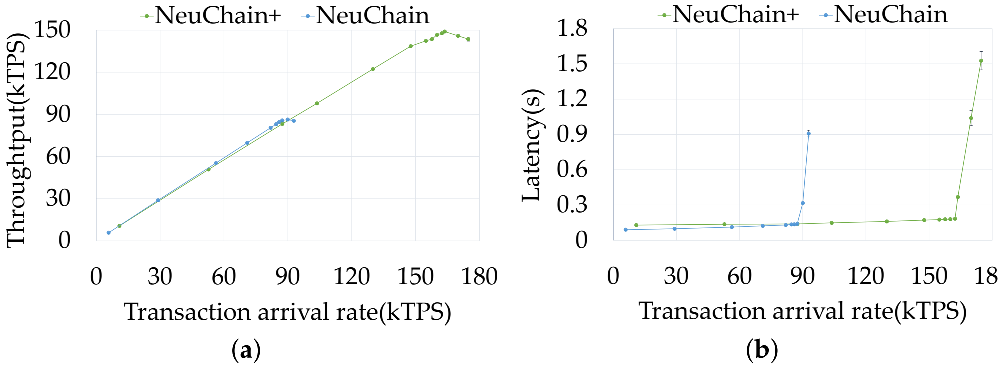


Figure 7. Performance of NeuChain+ and NeuChain under a changing transaction arrival rate (SmallBank). (**a**) Throughput performance when changing the transaction arrival rate. (**b**) Latency performance when changing the transaction arrival rate.

NeuChain+ processes different transaction sets, respectively, through two shards, theoretically improving transaction processing efficiency by 2× compared to NeuChain. The actual peak performance appears at the transaction arrival rate of 160.1 kTPS, and the corresponding throughput is 146.7 kTPS. Due to the overhead of cross-shard transaction processing, the actual peak performance corresponds to approximately 1.8× that of NeuChain at a transaction arrival rate. Before transaction accumulation, NeuChain+ exhibits a latency improvement of approximately 0.04 s compared to NeuChain. We speculate that this is caused by cross-shard transaction processing, which is evaluated and analyzed in detail in Section 7.5.

7.3. Varying Network Latency

In order to evaluate the performance of NeuChain+ in different network environments, we manually changed the latency between nodes from 0.2 to 100 ms using traffic control and recorded the peak performance under the SmallBank workload.

Figure 8 indicates that as the latency between nodes increases, the transaction latency increases significantly. This is due to the growing time overhead of communication for exchanging intrashard transaction batches, transmitting read sets, and merging reserve tables between nodes. Concurrently, the throughput gradually decreases as the latency between nodes increases. NeuChain+ employs pipeline technology to optimize these communications, thereby reducing the impact on throughput.

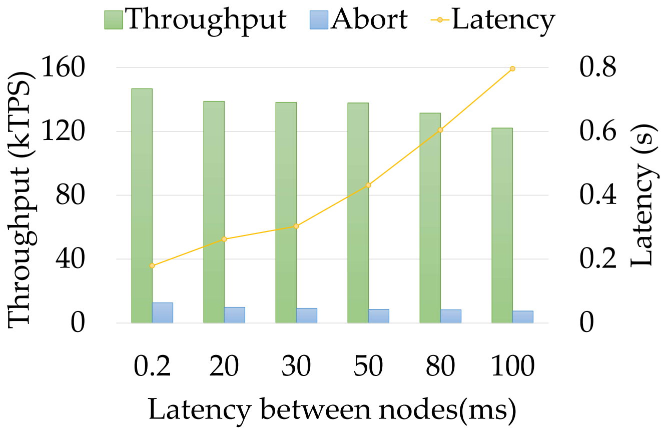


Figure 8. Performance of NeuChain+ under varying network latencies (SmallBank).

7.4. Scalability Comparison

To evaluate the scalability improvement of NeuChain+ using sharding compared to NeuChain, the changes in throughput, abort rate, and latency were tested under the SmallBank workload and configurations of 4, 8, 12, and 16 nodes, respectively. Additionally, the sharding permissioned blockchain Meepo was included as another comparison. Note that NeuChain+ was tested under environments with 1 to 4 shards, respectively, while Meepo operated under environments with 4, 8, 12, and 16 shards.

The experimental results are shown in Figure 9. NeuChain+ achieved excellent scalability, with a peak throughput of 247 kTPS. The throughput under two, three, and four shards was approximately 1.7×, 2.35×, and 2.9× that of NeuChain, respectively. Benefiting from cross-shard transaction processing based on deterministic execution, latency remained at a relatively low level, with the highest latency of just 0.188s for the four-shard setup. However, with the increase in the number of shards, the proportion of cross-shard transactions also increased, coupled with fewer accounts per shard (50,000, 33,000, and 25,000 accounts under two, three, and four shards, respectively), exacerbating transaction conflicts. Compared to NeuChain, the abort rate increased by 11, 25, and 56 kTPS under environments with two, three, and four shards, respectively. Meepo benefits from the trustworthiness within the consortium member and the complete ledger held by each consortium member. Therefore, Meepo maintained remarkable efficiency in processing cross-shard transactions, with a throughput of 65 kTPS under 16 shards. Additionally, Meepo employs a mechanism to redo all failed transactions, resulting in the 0 abort rate, but it also brings higher transaction latency.

In order to evaluate the improvement of storage pressure, we fixed the transaction arrival rate at the peak performance of NeuChain, 87.35 kTPS, and recorded the storage space consumption of a single node during a 5 min runtime of the system under the SmallBank workload. NeuChain+ runs on two, three, and four shards, respectively, and each shard deploys on four nodes of client proxies and block servers.

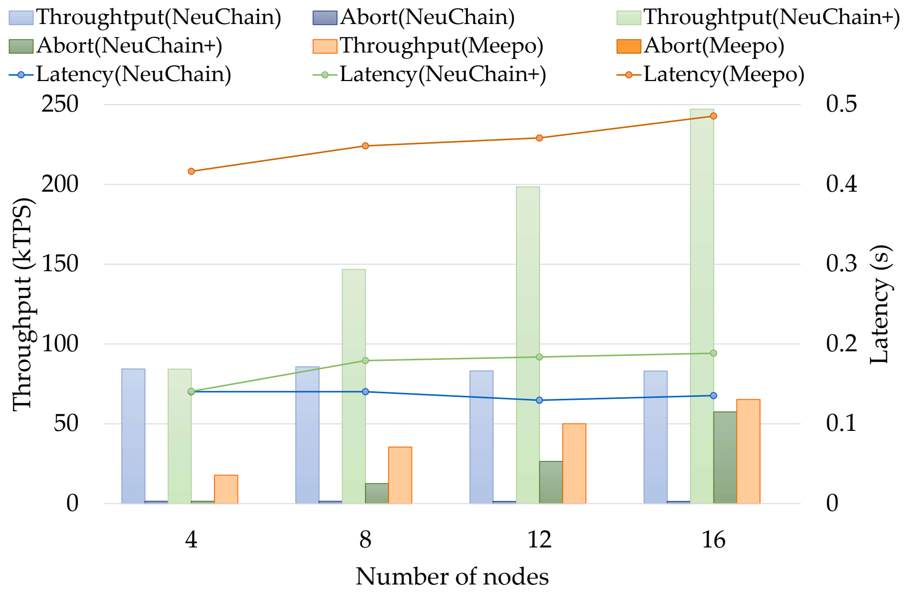


Figure 9. Performance comparison of scaling the number of nodes (SmallBank).

The experimental results, as illustrated in Figure 10, indicate that the baseline NeuChain consumes 7.6 GB of storage space after running for 5 min. NeuChain+ performs excellently in optimizing storage pressure. When running under two, three, and four shards for the 5 min, it only consumed 4.4 GB, 3.1 GB, and 2.4 GB of storage, respectively, which is approximately 58%, 41.2%, and 31.4% of NeuChain. Since cross-shard transactions need to be included in all involved shards, storage optimization is not entirely linear.

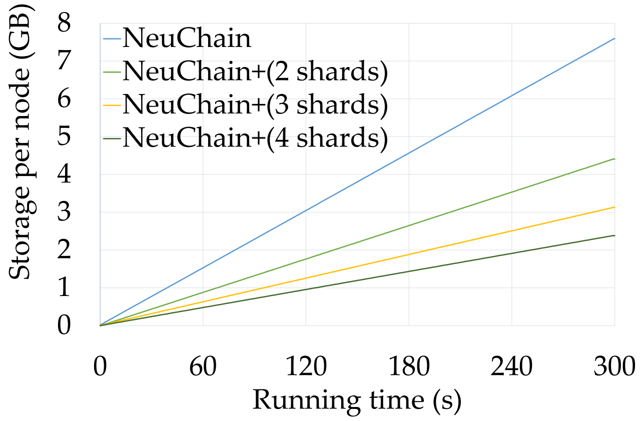


Figure 10. Single-node storage space under different shards (SmallBank).

7.5. *Cross-Shard Transaction Processing Overhead*

In order to evaluate the overhead of cross-shard transaction processing based on the deterministic execution designed in this paper, the time consumption of the transaction execution phase was tested using the peak performance under different numbers of shards (setup is the same as in Section 7.4).

The experimental results are shown in Figure 11. In the no-sharding situation, benefiting from deterministic execution, honest nodes could achieve consensus without communication coordination, resulting in an execution phase overhead of only 39.6 ms. After the introduction of sharding, the execution phase overhead increased by 43.9 ms, reaching 83.5 ms. This increase is attributed to the synchronization point that is difficult to optimize at *cross-reserve*. It is necessary to ensure the receipt of $f + 1$ copies of cross-shard reserve tables from all other shards before detecting conflicts for cross-shard transactions.

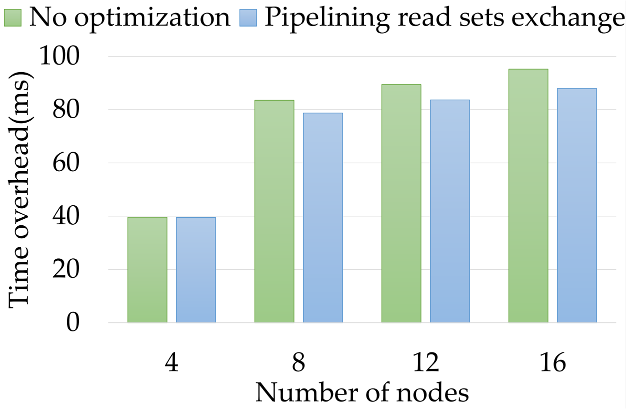


Figure 11. Optimization effect of pipeline technology for read sets transmission (SmallBank).

Furthermore, we also evaluated the impact of pipelined read sets transmission optimization on overhead. Under two, three, and four shards, the overhead was reduced by 4.8 ms, 5.8 ms, and 7.3 ms, respectively. Due to its optimization on generating read–write sets during the execution of cross-shard transactions, the effectiveness of this optimization was more obvious as the proportion of cross-shard transactions increased. Note that there are no cross-shard transactions under no sharding, so the optimization had no effect.

Although cross-shard transaction processing brings unavoidable time overhead, the results of continuing to increase the number of shards indicate that the overhead of cross-shard transaction processing based on deterministic execution is relatively fixed and almost does not increase with the increase in the number of shards and the proportion of cross-shard transactions. We believe that such an increase in time overhead is acceptable given the advantages of high scalability brought by sharding.

Summary of results. Experimental results show that NeuChain+ addresses the scalability issues of nonsharding and previous sharding blockchains. Under 16 nodes, the throughput was 2.9× that of NeuChain, and the storage overhead was only 30.9% of NeuChain. Compared to Meepo, NeuChain+ in the situation of storage sharding in each node and nontrustworthiness among all nodes, achieved 2.8× higher throughput and 61% lower latency. Compared with nonsharding blockchains under eight shards, NeuChain+ outperformed state-of-the-art blockchain systems with 1.7–75.3× throughput.

8. Conclusions

This paper proposes NeuChain+, a permissioned blockchain with the sharding–EV architecture, leveraging sharding technology. The sharding–EV architecture addresses the scalability deficiencies of the EV architecture, bringing significant performance improvements and horizontal scalability. An efficient cross-shard transaction processing protocol called *cross-reserve* is proposed based on deterministic concurrency control; it eliminates multiple rounds of coordination used for decision making in traditional cross-shard commit protocols, greatly reducing the overhead of cross-shard transactions processing. Additionally, the overhead is further reduced through pipelined read sets transmission.

Experimental results demonstrate that the sharding–EV architecture is more promising than other systems in comparison in addressing the performance issues of permissioned blockchains in high-throughput application scenarios. The sharding–EV architecture further improves the throughput through horizontal scalability based on the high throughput of the EV architecture, which is more suitable for large-scale deployment to meet the demands of high-throughput applications. Unlike previous sharding blockchain systems, the *cross-reserve* proposed in this paper eliminates the costly multiround coordination overhead, thereby avoiding the limitation that cross-shard commit protocols like 2PC impose on system performance.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/app14114897/s1, Table S1: NeuChain+ Experimental Data.

Author Contributions: Conceptualization, Y.Z.; data curation, Y.Z.; investigation, X.L.; methodology, X.L.; software, Y.G.; supervision, G.Y.; validation, Z.P.; visualization, Y.G.; writing—original draft, Y.G.; writing—review and editing, Z.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Social Science Foundation of China (21&ZD124), the National Natural Science Foundation of China (grant numbers 61991404, 62072086, and 62372097), and the Special Funds for Basic Scientific Research of Central Universities (N2116005, N2116008, and N2416003).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the Supplementary Materials, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Truong, V.T.; Le, L.; Niyato, D. Blockchain Meets Metaverse and Digital Asset Management: A Comprehensive Survey. *IEEE Access* **2023**, *11*, 26258–26288. [CrossRef]
2. Kamble, S.S.; Gunasekaran, A.; Subramanian, N.; Ghadge, A.; Belhadi, A.; Venkatesh, M. Blockchain technology's impact on supply chain integration and sustainable supply chain performance: Evidence from the automotive industry. *Ann. Oper. Res.* **2023**, *327*, 575–600. [CrossRef]
3. Elisa, N.; Yang, L.; Chao, F.; Cao, Y. A framework of blockchain-based secure and privacy-preserving E-government system. *Wirel. Networks* **2023**, *29*, 1005–1015. [CrossRef]
4. Ray, P.P. Web3: A comprehensive review on background, technologies, applications, zero-trust architectures, challenges and future directions. *Internet Things Cyber Phys. Syst.* **2023**, *3*, 213–248 . [CrossRef]
5. Satija, S.; Mehra, A.; Singanamalla, S.; Grover, K.; Sivathanu, M.; Chandran, N.; Gupta, D.; Lokam, S. Blockene: A high-throughput blockchain over mobile devices. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), Virtual, 4–6 November 2020; pp. 567–582.
6. Ulm, G.; Smith, S.; Nilsson, A.; Gustavsson, E.; Jirstrand, M. OODIDA: On-board/off-board distributed real-time data analytics for connected vehicles. *Data Sci. Eng.* **2021**, *6*, 102–117. [CrossRef]
7. Di Vaio, A.; Varriale, L. Blockchain technology in supply chain management for sustainable performance: Evidence from the airport industry. *Int. J. Inf. Manag.* **2020**, *52*, 102014. [CrossRef]
8. Tapscott, A.; Tapscott, D. How blockchain is changing finance. *Harv. Bus. Rev.* **2017**, *1*, 2–5.
9. Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; pp. 1–15.
10. Visa Fact Sheet: What You Need to Know about One of the World's Largest Payments Companies. 2018. Available online: https://www.visa.co.uk/dam/VCOM/download/corporate/media/visanet-technology/aboutvisafactsheet.pdf (accessed on 17 May 2024).
11. Amiri, M.J.; Agrawal, D.; Abbadi, A.E. CAPER: A cross-application permissioned blockchain. *Proc. VLDB Endow.* **2019**, *12*, 1385–1398. [CrossRef]
12. Castro, M.; Liskov, B. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. (TOCS)* **2002**, *20*, 398–461. [CrossRef]
13. Ongaro, D.; Ousterhout, J. In search of an understandable consensus algorithm. In Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14), Philadelphia, PA, USA, 19–20 June 2014; pp. 305–319.
14. Eyal, I.; Gencer, A.E.; Sirer, E.G.; Van Renesse, R. {Bitcoin-NG}: A scalable blockchain protocol. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), Santa Clara, CA, USA, 16–18 March 2016; pp. 45–59.
15. Bentov, I.; Pass, R.; Shi, E. Snow White: Provably secure proofs of stake. *IACR Cryptol. ePrint Arch.* **2016**, *2016*, 1–62.
16. Kiayias, A.; Russell, A.; David, B.; Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In Proceedings of the Annual International Cryptology Conference, Santa Barbara, CA, USA, 20–24 August 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 357–388.

17. Miller, A.; Xia, Y.; Croman, K.; Shi, E.; Song, D. The honey badger of BFT protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 31–42.
18. Yin, M.; Malkhi, D.; Reiter, M.K.; Gueta, G.G.; Abraham, I. HotStuff: BFT consensus with linearity and responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, Toronto, ON, Canada, 29 July–2 August 2019; pp. 347–356.
19. Gilad, Y.; Hemo, R.; Micali, S.; Vlachos, G.; Zeldovich, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28 October 2017; pp. 51–68.
20. Gupta, S.; Rahnama, S.; Hellings, J.; Sadoghi, M. Resilientdb: Global scale resilient blockchain fabric. *arXiv* **2020**, arXiv:2002.00160.
21. Gupta, S.; Hellings, J.; Sadoghi, M. Rcc: Resilient concurrent consensus for high-throughput secure transaction processing. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; pp. 1392–1403.
22. Stathakopoulou, C.; Pavlovic, M.; Vukolić, M. State machine replication scalability made simple. In Proceedings of the Seventeenth European Conference on Computer Systems, Rennes, France, 5–8 April 2022; pp. 17–33.
23. Qi, J.; Chen, X.; Jiang, Y.; Jiang, J.; Shen, T.; Zhao, S.; Wang, S.; Zhang, G.; Chen, L.; Au, M.H.; et al. Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual, 26–29 October 2021; pp. 18–34.
24. Nathan, S.; Govindarajan, C.; Saraf, A.; Sethi, M.; Jayachandran, P. Blockchain meets database: Design and implementation of a blockchain relational database. *arXiv* **2019**, arXiv:1903.01919.
25. Peng, Z.; Zhang, Y.; Xu, Q.; Liu, H.; Gao, Y.; Li, X.; Yu, G. Neuchain: A fast permissioned blockchain system with deterministic ordering. *Proc. VLDB Endow.* **2022**, *15*, 2585–2598. [CrossRef]
26. Luu, L.; Narayanan, V.; Zheng, C.; Baweja, K.; Gilbert, S.; Saxena, P. A secure sharding protocol for open blockchains. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 17–30.
27. Kokoris-Kogias, E.; Jovanovic, P.; Gasser, L.; Gailly, N.; Syta, E.; Ford, B. Omniledger: A secure, scale-out, decentralized ledger via sharding. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 583–598.
28. Zamani, M.; Movahedi, M.; Raykova, M. Rapidchain: Scaling blockchain via full sharding. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 931–948.
29. Al-Bassam, M.; Sonnino, A.; Bano, S.; Hrycyszyn, D.; Danezis, G. Chainspace: A sharded smart contracts platform. *arXiv* **2017**, arXiv:1708.03778.
30. Hellings, J.; Sadoghi, M. Byshard: Sharding in a byzantine environment. *Proc. VLDB Endow.* **2021**, *14*, 2230–2243. [CrossRef]
31. Dang, H.; Dinh, T.T.A.; Loghin, D.; Chang, E.C.; Lin, Q.; Ooi, B.C. Towards scaling blockchain systems via sharding. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 123–140.
32. Mohan, C.; Lindsay, B.; Obermarck, R. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst. (TODS)* **1986**, *11*, 378–396. [CrossRef]
33. Özsu, M.T.; Valduriez, P. *Principles of Distributed Database Systems*; Springer: Berlin/Heidelberg, Germany, 1999; Volume 2.
34. Wang, J.; Wang, H. Monoxide: Scale out blockchains with asynchronous consensus zones. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), Boston, MA, USA, 26–28 March 2019; pp. 95–112.
35. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://assets.pubpub.org/d8wct41f/31611263538139.pdf (accessed on 20 October 2021).
36. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
37. Zheng, P.; Xu, Q.; Zheng, Z.; Zhou, Z.; Yan, Y.; Zhang, H. Meepo: Sharded consortium blockchain. In Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), Chania, Greece, 19–22 April 2021; pp. 1847–1852.
38. Huang, H.; Peng, X.; Zhan, J.; Zhang, S.; Lin, Y.; Zheng, Z.; Guo, S. Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding. In Proceedings of the IEEE INFOCOM 2022—IEEE Conference on Computer Communications, London, UK, 2–5 May 2022; pp. 1968–1977.
39. Hong, Z.; Guo, S.; Zhou, E.; Zhang, J.; Chen, W.; Liang, J.; Zhang, J.; Zomaya, A. Prophet: Conflict-Free Sharding Blockchain via Byzantine-Tolerant Deterministic Ordering. In Proceedings of the IEEE INFOCOM 2023—IEEE Conference on Computer Communications, New York, NY, USA, 17–20 May 2023; pp. 1–10.
40. De Angelis, S.; Aniello, L.; Baldoni, R.; Lombardi, F.; Margheri, A.; Sassone, V. PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain. In Proceedings of the CEUR Workshop Proceedings. CEUR-WS, Milan, Italy, 7 May 2022; Volume 2058.
41. Kung, H.T.; Robinson, J.T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. (TODS)* **1981**, *6*, 213–226. [CrossRef]
42. Dinh, T.T.A.; Wang, J.; Chen, G.; Liu, R.; Ooi, B.C.; Tan, K.L. Blockbench: A framework for analyzing private blockchains. In Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, IL, USA, 14–19 May 2017; pp. 1085–1100.
43. Cahill, M.J.; Röhm, U.; Fekete, A.D. Serializable isolation for snapshot databases. *ACM Trans. Database Syst. (TODS)* **2009**, *34*, 1–42. [CrossRef]
44. Dwork, C.; Lynch, N.; Stockmeyer, L. Consensus in the presence of partial synchrony. *J. ACM (JACM)* **1988**, *35*, 288–323. [CrossRef]

45. Lu, Y.; Yu, X.; Cao, L.; Madden, S. *Aria: A Fast and Practical Deterministic OLTP Database*; Massachusetts Institute of Technology (MIT): Cambridge, MA, USA, 2020.
46. NeuChain: A Fast Permissioned Blockchain System with Deterministic Ordering. 2023. Available online: https://github.com/iDC-NEU/NeuChain (accessed on 30 June 2022).
47. braft: An Industrial-Grade C++ Implementation of RAFT Consensus Algorithm. 2023. Available online: https://github.com/baidu/braft (accessed on 15 July 2022).
48. LevelDB: A Fast Key-Value Storage Library. 2021. Available online: https://github.com/google/leveldb (accessed on 7 July 2022).
49. ZeroMQ: A High-Performance Asynchronous Messaging Library. 2023. Available online: https://github.com/zeromq/cppzmq (accessed on 4 July 2022).
50. Protocol Buffers: The Language-Neutral, Platform-Neutral Extensible Mechanisms for Serializing Structured Data. 2023. Available online: https://github.com/protocolbuffers/protobuf (accessed on 4 July 2022).
51. Ruan, P.; Loghin, D.; Ta, Q.T.; Zhang, M.; Chen, G.; Ooi, B.C. A transactional perspective on execute-order-validate blockchains. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 543–557.
52. Gorenflo, C.; Lee, S.; Golab, L.; Keshav, S. FastFabric: Scaling hyperledger fabric to 20,000 transactions per second. *Int. J. Netw. Manag.* **2020**, *30*, e2099. [CrossRef]
53. Suri-Payer, F.; Burke, M.; Wang, Z.; Zhang, Y.; Alvisi, L.; Crooks, N. Basil: Breaking up BFT with ACID (transactions). In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual, 26–29 October 2021; pp. 1–17.
54. Alibaba Cloud: Cloud Computing Services. 2024. Available online: https://www.alibabacloud.com/zh?_p_lc=1 (accessed on 15 April 2024).
55. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.