NeuChain: A Fast Permissioned Blockchain System with Deterministic Ordering

Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, Ge Yu
Northeastern University, China
{pengzs, qianxu, liuhaixu, gaoyuxiao}@stumail.neu.edu.cn, {zhangyf, lixiaohua, yuge}@mail.neu.edu.cn

ABSTRACT

Blockchain serves as a replicated transactional processing system in a trustless distributed environment. Existing blockchain systems all rely on an explicit ordering step to determine the global order of transactions that are collected from multiple peers. The ordering consensus can be the bottleneck since it must be Byzantine-fault tolerant and can scarcely benefit from parallel execution. In this paper, we propose an ordering-free architecture that makes ordering implicit through deterministic execution. Based on this novel architecture, we develop a permissioned blockchain system `NeuChain`. A number of key optimizations such as asynchronous block generation and pipelining are leveraged for high throughput and low latency. Several security mechanisms are also designed to make our system robust to malicious attacks. Our geo-distributed experimental results show that `NeuChain` can achieve 47.2-64.1× throughput improvement over HyperLedger Fabric and 1.6-12.2× throughput improvement over the state-of-the-art high performance blockchains.

PVLDB Reference Format:
Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, Ge Yu. NeuChain: A Fast Permissioned Blockchain System with Deterministic Ordering. PVLDB, 15(11): 2585 - 2598, 2022.
doi:10.14778/3551793.3551816

1 INTRODUCTION

Blockchain has evolved into a technology conducting transactions in a secure and verifiable manner without the need for a trusted third party. To enhance its viability in practice, blockchain systems must support transaction rates comparable to those supported by existing database management systems, which can provide the same transactional guarantees. Great efforts have been put on developing blockchain systems to support high-throughput trusted transactions [14, 30, 52, 55].

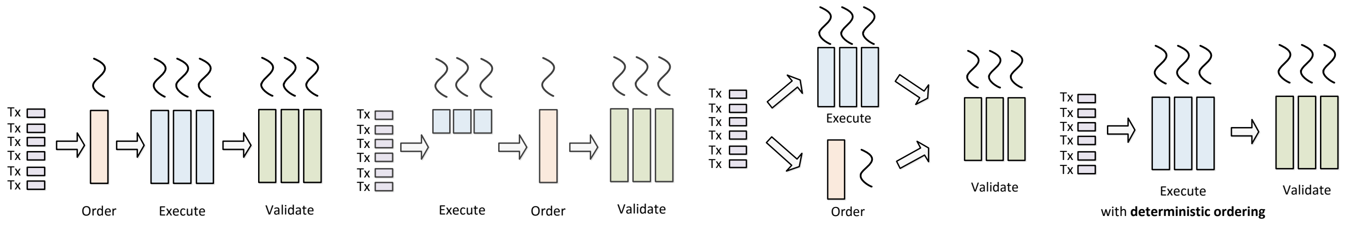
In a trustless distributed environment, the keys to meet transactional requirements lie in how to validate the correctness of transaction contents and how to achieve consensus on the order of transactions. Permissionless (public) blockchains, such as Bitcoin and Ethereum, follow an **order-execute-validate** (**`OEV`**) architecture [14], running a *proof-of-work* (PoW) consensus to determine an order of transactions. All peers are involved in solving a puzzle which must be hard enough to prevent Byzantine attacks. The order is determined by a lucky peer that offers the solution first. The possibility of solving puzzles depends on the computing power of that peer. This order is broadcast to all peers, and each peer executes these transactions in the same order and validates their contents. PoW provides eventual consistency and could take a long time for each peer to reach a consensus.

On the other hand, permissioned (consortium or private) blockchains, such as Hyperledger Fabric [14], running among a set of known, identified participants, follow an **execute-order-validate** (**`EOV`**) architecture. First, transactions from different clients are executed concurrently by a set of endorsement peers. The effects of execution in terms of multiple read-write sets are sent to an *ordering* service which may consist of a cluster of ordering peers. By relying on the identities of the peers, the traditional Byzantine-fault tolerant (BFT) consensus can be used to produce a totally ordered sequence of the endorsed transactions grouped in blocks. Then, the block is broadcast to all validation peers. Each validation peer validates the state changes from the endorsed transactions with respect to the endorsement policy and serializability. The `EOV` architecture can greatly increase system throughput [30] by introducing parallelism in the execution phase. However, this is still much slower than traditional databases and far from meeting the demand of high throughput applications, such as financial applications [57], internet of things [53, 61], and industrial supply chain [25].

The reason of performance degradation is attributed to the expensive ordering phase that requires all peers to make a consensus on the serial order of transactions. Determining the serial order can scarcely benefit from parallel computation and usually becomes performance bottleneck. Figure 1a shows the `OEV` architecture where the serial order is determined by a single peer through expensive PoW process. Figure 1b illustrates the `EOV` architecture where the serial order is determined by an independent ordering service, which requires a single orderer or multiple orderers to reach consensus on the serial order. The single node that proposes a complete block of ordered transactions could be the bottleneck, limited by its upstream bandwidth or computation resources. Nathan et al. [40] propose an **order-execute-parallel-validate** (**`OEPV`**) architecture (as shown in Figure 1c) that hides the ordering cost by parallelizing the execution phase and the ordering phase. After the consensus on the global order is reached, the executed transactions that conflict with the order will be aborted and the state is rolled back. Though various concurrency control techniques [12, 19, 33, 62] have been exploited to improve throughput, all of these architectures involve an expensive ordering step that must be executed sequentially.

Essentially, blockchain is a multi-master replicated ledger system that provides the consensus on transaction contents, transaction order, and block boundaries. The execution process generates the

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551816



(a) OEV (Bitcoin [39], Etherum [65]) (b) EOV (Fabric [14], Fabric# [52]) (c) OEPV (Fabric SSI [40], BIDL [45]) (d) EV (ours)

Figure 1: Blockchain architectures.

updated state of database (*i.e.,* read-write sets). The validation process is performed on all peers to ensure the validity and legality of transaction contents. Since any peer can accept write requests independently, the ordering step is necessary to determine a global order of transactions, *i.e.,* determine the block boundaries (inter-block transaction order) and the transaction order within each block (intra-block transaction order). However, it is not necessary to launch the ordering step explicitly with consensus cost.

In this paper, we leverage ***deterministic*** ordering [11, 35, 48, 58] to define the intra-block transaction order and leverage *epoch* to define the inter-block transaction order, so that the explicit ordering step with single point bottleneck is eliminated to realize an ordering-free **execute-validate** (**EV**) architecture. Each transaction is assigned with an epoch number that helps divide transactions into block, which is assigned by a single trusted epoch server or a cluster of epoch servers that reach a consensus through PBFT. Different from existing architectures where a single node (or the single leader during consensus) proposes a complete set of ordered transactions to other nodes, in EV architecture each node can propose a subset of transactions independently, and multiple nodes are running concurrently for proposing transactions of the same epoch. Thus, the consensus on parts of transactions and the block generation can be parallelized. With a complete set of unordered transactions as input, by deterministic ordering all peer nodes will produce the same results without coordination. The inconsistency of orders or contents of a block will be detected and avoided during the validation phase which makes EV architecture robust to Byzantine attacks.

Based on the EV architecture, we develop a blockchain system **NeuChain**. The system does not only include clients and block servers but also introduces new components such as client proxies and epoch servers. NeuChain further improves throughput through asynchronous block generation and reduces latency through pipelining. NeuChain also provides comprehensive security support to deal with malicious attacks from user clients, block servers, client proxies, and epoch servers.

In summary, we have made the following contributions.

- **Ordering-Free EV Architecture**. We introduce the deterministic ordering technique to blockchain and propose the EV architecture. This design brings the flexibility in ordering intra-block transactions and further reduces the number of transaction conflicts through transaction reordering.
- **NeuChain System**. We design and implement a permissioned blockchain NeuChain based on the EV architecture. A number of key optimizations such as asynchronous block generation and pipelining are leveraged for high throughput and low latency.
- **Mechanisms to Defend Malicious Attacks**. As a novel architecture, NeuChain introduces several new components (*e.g.,* epoch server and client proxy) which could bring potential security risks. We provide comprehensive defence mechanisms to tackle malicious activities conducted by various Byzantine nodes.

We perform experiments on a geo-distributed 8-node Aliyun cluster by comparing with Hyperledger Fabric [14], FastFabric [30], a high-performance transactional blockchain ResilientDB [32], a sharded consortium blockchain Meepo [32], and a recently proposed BFT key-value store Basil [56]. We also conduct comparisons with a series of NeuChain variants (with OEV, EOV, and OEPV) to study the efficiency of our EV architecture under the same implementation framework. The results on YCSB and SmallBank workloads show that NeuChain can achieve 47.2-64.1× throughput improvement over Fabric and 1.6-12.2× throughput improvement over ResilientDB, Meepo, and Basil. It also shows superiority over other NeuChain variants with different architectures, say 11.2-16.7× over OEV, 7.4-19.0× over EOV, and 3.7-4.8× over OEPV.

2 BACKGROUND OF BLOCKCHAINS

In this section, we review the existing blockchain systems from different dimensions. We first study the mainstream architectures adopted by existing blockchains, including OEV, EOV, and OEPV. Then we discuss how these blockchains reach consensus. Lastly, we summarize the concurrent transaction processing techniques adopted by these systems to improve their transaction throughput.

2.1 Architectures

2.1.1 OEV *Architecture.* The OEV architecture uses the following process to generate blocks. i) In the ordering phase, each node selects a set of transactions, orders and pre-executes them to ensure their validity in that order; ii) After a node computes a valid PoW, it generates a block and broadcasts it to all the other nodes; iii) In the validation phase, after receiving a block, all other nodes need to ensure that all transactions in that block are valid by re-executing the transactions.

The OEV architecture put emphasis on the ordering consensus, which often relies on PoW to achieve consensus (*e.g.,* Bitcoin [39] and Ethereum [65]), and therefore widely used in permissionless blockchains with a wide range of untrusted peer nodes. Under a permissioned environment, recent systems such as Quorum [7] rely on a relatively less expensive consensus protocol (*e.g.,* PBFT [17] or Raft [42]) to achieve the order agreement. However, the OEV architecture suffers from the drawback of sequential execution due to the order-first architecture. That is, it requires all participant nodes to execute the same set of transactions based on the serial order that is agreed by all peers, where no concurrency is allowed.

Table 1: Summary of Blockchain Systems

Systems	Architecture	Type	Consensus			Transaction Processing	
			Content	Protocol	Participant	Conflict type	Concurrency control
Bitcoin [39]	OEV	permissionless	block	PoW	all nodes	intra-block	-
Ethereum [65]	OEV	permissionless	block	PoW PoS	all nodes	intra-block	-
Quorum [7]	OEV	permissioned	block	PBFT Raft	all nodes	intra-block	-
ResilientDB [32]	OEV	permissioned	Tx batch	GEOBFT	all nodes	intra-block	-
PoE [31]	OEV	permissioned	Tx batch	PoE	all nodes	intra-block	-
Monoxide [64]	shard+OEV	permissionless	block	PoW	group of nodes	cross-shard	eventual atomicity
ByShard [34]	shard+OEV	permissioned	block	PBFT	group of nodes	cross-shard	cross-shard 2PC
SharPer [13]	shard+OEV	permissioned	block	PBFT	group of nodes	cross-shard	cross-shard commit protocol
Rivet [22]	shard+OEV	permissioned	block	HotStuff	reference shard	cross-shard	optimistic
Fabric [14]	EOV	permissioned	Tx batch	PBFT Raft	order nodes	inter&intra-block	MVOCC
FastFabric [30]	EOV	permissioned	Tx header	PBFT Raft	order nodes	inter&intra-block	MVOCC
Fabric++ [55]	EOV	permissioned	Tx batch	PBFT Raft	order nodes	inter&intra-block	MVOCC+reorder
Fabric# [52]	EOV	permissioned	Tx batch	PBFT Raft	order nodes	inter&intra-block	OCC+SSI+reorder
SlimChain [69]	EOV	both	block	PoW Raft	consensus nodes	inter&intra-block	OCC+SSI
Basil [56]	EOV	permissioned	KVs	PBFT	clients	inter&intra-block	MVTSO
Fabric SSI [40]	OEPV	permissioned	block	PBFT Raft	order nodes	intra-block	SSI
BIDL [45]	OEPV	permissioned	block	PBFT Raft	order nodes	intra-block	-
NeuChain (ours)	EV	permissioned	block number Tx batch	PBFT Raft	epoch servers client proxies	intra-block	deterministic

To alleviate the consensus overhead and to increase the scalability, researchers introduce the *sharding* technique from distributed database into OEV blockchains. The sharded blockchain groups participant nodes into clusters and lets each cluster of nodes maintain a subset of transactions initiated from its own cluster. However, the sharding solutions bring new challenges. i) It is essential to use a cross-shard commit protocol to ensure the atomicity of transactions, causing additional round-trip communication overhead. ii) The node-to-cluster assignment should be carefully taken to ensure security (*i.e.,* solving the problem of computing power being diluted after partitioning). For example, the sharded blockchains, such as Monoxide [64], ByShard [34], Sharper [13], and Rivet [22], propose different techniques to address these two challenges.

2.1.2 EOV Architecture. The EOV architecture is proposed to adapt to permissioned blockchains with a set of identified participants. It has significantly improved the transaction throughput by introducing parallel execution and optimistic concurrency control (OCC) technology. The process of the EOV architecture is described as follows. i) In the execution phase, a client sends a transaction to a group of endorsing peers for simulation and producing an endorsement. It then collects and forwards the results to the ordering service. ii) In the ordering phase, the ordering service that may be consist of multiple ordering peers uses a consensus protocol to determine the order of transactions in a block. After generating a block, peers pull blocks from the ordering service. iii) In the validation phase, all peers receive the same block and perform a read-write conflict check on each transaction and send committed or aborted notifications back to clients.

Since the execution phase is performed first and no longer depends on the ordering result, the input transactions can be executed in parallel, even though there do exist dependencies among transactions. The OCC method is used to detect conflicts based on the ordering result. In this way, the performance of the EOV system can be greatly improved. Hyperledger Fabric [14] is the first blockchain system that adopts EOV architecture and has been used in many real applications. A series of blockchains that improve Fabric are recently proposed, such as FastFabric [30], Fabric++ [55], and Fabric# [52]. These works improve the parallelism of EOV architecture by relying on various concurrency control techniques. SlimChain follows the EOV architecture, which performs off-chain execution under trusted execution environment and stateless on-chain transaction ordering and validation. Basil [56] also follows EOV architecture which uses local clock to order transactions and relies on NTP's clock to verify the order. For EOV systems, they may suffer from high abort rates due to many inter&intra-block conflicts and as a result low throughput, especially under high contention workload.

2.1.3 OEPV Architecture. To further increase the throughput, the OEPV architecture parallelizes the ordering phase and the execution phase. The OEPV architecture abides by the following process to generate blocks. i) A client sends a transaction to the ordering service and to the execution nodes simultaneously, *i.e.,* the ordering and the execution are running in parallel. ii) Based on the order sequence and execution results, the validation node determines the successfully committed or aborted transactions and responds to client. iii) Multiple execution nodes exchange the signatures of execution results to ensure that the third party can verify the block without additional information.

The authors in [40] propose the OEPV architecture and use Serializable Snapshot Isolation (SSI) [16] to resolve transaction conflicts. We name this system as Fabric SSI. BIDL [45] employs speculative execution that is essentially the same as OEPV to provide fast transaction processing service. Parallelizing the ordering and the execution can hide the consensus cost of distributed BFT ordering. However, the OEPV architecture still needs an explicit ordering phase that slows down the transaction processing speed.

2.2 Consensus

The consensus on transaction contents and transaction order in a block should be reached in blockchain systems. In permissionless blockchains, due to the massive number of untrusted nodes, a Proof-of-Resources based consensus protocol (*e.g.,* Proof-of-Work and Proof-of-Stake [5]) is needed to ensure the security, *e.g.,* Bitcoin [39], Ethereum [65], and Monoxide [64]. Due to the high cost of PoW consensus, it is better to reduce the number of consensus calls. Hence, the consensus unit in permissionless blockchains is usually a whole block. While in permissioned blockchains (*e.g.,* Quorum [7]) where only a limited number of participant nodes are involved in consensus, a PBFT or Raft protocol is more suitable for distributed consensus. To adjust to geo-distributed environment, ResilientDB [32] employs a hierarchical consensus protocol GEOBFT. It first divides participant nodes into multiple clusters according to their locations. Each cluster first makes PBFT consensus on a batch of local transactions and then exchanges data with other clusters to make a global consensus. To hide the consensus cost, the PoE protocol [31] starts execution before reaching consensus (*i.e.,* speculative execution) which reduces the response latency.

The cost of consensus process is proportional to the number of participant nodes. As the number of participating nodes increases, the system performance will drop sharply. In sharded blockchains, the participant nodes are partitioned into clusters/shards. The consensus is established within each cluster, so it is less expensive and can help improve the scalability. However, there could exist cross-shard transactions. It is necessary to exploit a cross-shard consensus protocol or a cross-shard commit protocol to ensure the atomicity. For example, Monoxide [64] proposes eventual atomicity. ByShard [34] applies two-phase commit (2PC) protocol to ensure the atomicity of cross-shard transactions. In Rivet [22], a reference shard is selected to make consensus using HotStuff [71] protocol. These sharded blockchains reduce the network overhead of a single consensus protocol at the cost of increasing the number of consensus rounds for cross-shard transactions.

With EOV architecture, the consensus on transaction order needs to be reached after the parallel execution phase, and the consensus only involves a set of ordering nodes. As EOV architecture is mainly adopted in permissioned blockchains, PBFT and Raft protocols are usually used. Separating ordering nodes from execution nodes and only requiring a small number of ordering nodes to participate in consensus help improve the performance and scalability. The consensus unit is a batch of transactions in Fabric [14], Fabric++ [55], and Fabric# [52]. To reduce the size of consensus message, FastFabric [30] only makes a consensus on the transaction headers. SlimChain [69] establishes a consensus on the order of transactions among all on-chain consensus nodes (Raft consensus in the permissioned setting while PoW consensus in the permissionless setting). Basil [56] is a BFT key-value store so its consensus unit is a set of key-value pairs. Similar to EOV architecture, Fabric SSI [40] and BIDL [45] that are with OEPV architectures both establish consensus on the transaction order, and at the same time launch transaction processing in parallel.

2.3 Concurrent Transaction Processing

The early OEV blockchain systems only support serial execution of transactions because transaction processing is often not the bottleneck [26]. For example, Bitcoin's PoW dominates the block generation process which is more time consuming than transaction processing. On the other hand considering the complexity of parallel execution, serial execution will not lead to anomalous behaviors when the transaction execution is replicated over many nodes, so it is more preferable in early blockchains. However, concurrent transaction processing is supported by more and more recently proposed blockchains to improve transaction processing throughput.

With sharded OEV architecture, the concurrent transactions initiated from different shards may read or write the same records, which leads to conflicts for cross-shard transactions. It is necessary to exploit cross-shard commit protocol to ensure atomicity. ByShard [34] proposes an orchestrate-execute model supporting multiple 2PC variants and multiple isolation levels to commit cross-shard transactions. Sharper [13] establishes a BFT commit protocol for ordering cross-shard transactions among the involved shards. Rivet [22] proposes to use a more optimistic cross-shard commit protocol. Monoxide [64] proposes the idea of asynchronous consensus zone to achieve eventual atomicity, *i.e.,* for a cross-shard transaction initiated from a shard, it will eventually be processed in other shards without real-time guarantee.

With EOV and OEPV architectures, the order of transactions is determined after parallel execution. The parallel execution on different nodes might lead to transaction conflicts (if there exist read-write dependencies among transactions). The key of concurrent transaction processing is to tackle such conflicts efficiently. There are two types of transaction conflicts in blockchains: intra-block and inter-block conflicts. Concurrent transaction execution might form inter-block transaction conflicts, which occurs only in the EOV architecture that performs execution first based on different blocks. The blockchains with EOV and OEPV architectures adopt OCC and various techniques to achieve concurrency control. For example, Fabric# [52] and SilmChain [69] resolve conflicts in the ordering phase. Fabric [14], Fabric++ [55], and Fabric SSI [40] assign a sequence number to each transaction in the ordering phase and resolve conflicts in the validation phase, where they use Multi-Version OCC or SSI to ensure the consistency of transactions. Fabric++ and Fabric# further apply a transaction reordering technique to reduce abort rates. Basil [56] relies on Multi-Version TimeStamp Ordering (MVTSO) to support concurrency control.

Insight. A blockchain system is essentially a fully replicated storage system for storing ledgers. The key in blockchains is to ensure consistency among these replicas under a trustless environment. Besides the encryption cost, the most expensive step is the consensus on the transaction order, which is essential in all architecture variants (OEV, EOV, and OEPV). An effective optimization technique to improve the throughput performance is the concurrent transaction processing with limited abort rate, where an essential contradiction exists between the serial execution order (for correctness) and the concurrent execution (for performance). Therefore, the elimination of explicit ordering phase is desirable to achieve high performance.

3 SYSTEM DESIGN

In this section, we propose an ordering-free EV architecture and a permissioned blockchain prototype NeuChain. We first overview the system design and introduce the key components in Section

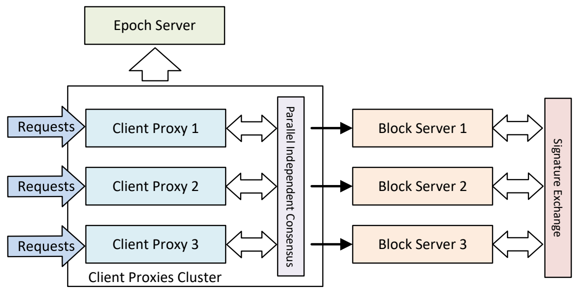


Figure 2: Overall system structure.

3.1 and describe the execution flow details in Section 3.2. Then, we present how the deterministic transaction processing works in Section 3.3. Lastly, we provide some key optimization techniques in our implementation in Section 3.4.

3.1 Overview and Key Components

NeuChain is designed to achieve ordering-free EV architecture as shown in Figure 1d. The explicit ordering consensus phase is eliminated by employing the implicit deterministic ordering. The basic idea can be summarized as follows. All participant nodes accept transaction requests in parallel. These sets of transactions received from multiple nodes are exchanged among nodes through all-to-all communication. After a node collects all transactions from all remote nodes, it executes these transactions in a pre-defined deterministic order and generates a block. The deterministic order can be defined according to the globally unique transaction ID that is generated on each node based on a deterministic and malicious-resilient rule. Due to this deterministic execution, the generated blocks by different nodes will be consistent (*i.e.,* consensus is reached) if no malicious behavior occurs (the trust and security issues will be discussed in Section 4). Following the basic idea, we design a new blockchain system as shown in Figure 2. Different from existing blockchains, several new components are introduced to realize deterministic block generation.

Epoch Server. A blockchain records transactions by dividing them into blocks. A latter block is generated based on the former block's hash for tamper resistance. The blockchain nodes must reach consensus on the following two issues: i) which block a transaction belongs to and ii) in which order the transactions execute within a block. With deterministic execution, the execution order is fixed, but it is still necessary to make consensus on the block boundaries. To tackle this problem, we design an *epoch server* to determine which block a transaction belongs to. A batch of transactions are sent to the epoch server, where a monotonically increased epoch number (based on local clock) is assigned to each transaction that corresponds to its belonging block. To avoid Byzantine attacks, multiple epoch servers can be deployed, and they need to make consensus when the epoch number increases.

Client Proxy. In our system, multiple *client proxies* are deployed. Client proxy has two functionalities. i) A client proxy is responsible for accepting user client's transaction requests and groups them into batches. It then applies an epoch number from the epoch server for a batch of transactions. A client proxy may generate several transaction batches in an epoch. ii) After the epoch is over, the client proxy packages the transactions within the same epoch and broadcasts them to all other client proxies. Because all client proxies need to ensure the integrity and consistency of the transactions,

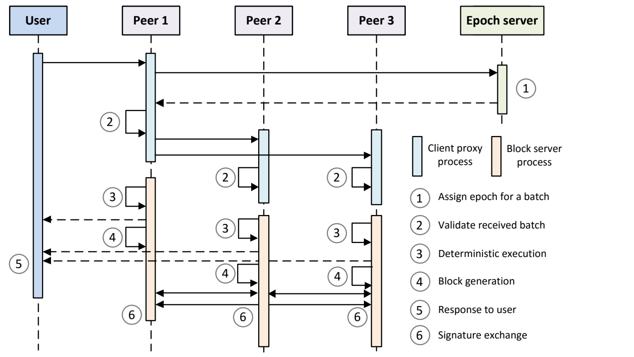


Figure 3: Execution flow of a single transaction in NeuChain. It consists of three phases: preparation phase (step 1-2), execution phase (step 3-5), and validation phase (step 6).

PBFT or Raft can be used as the broadcast protocol. Unlike other blockchain systems that perform centralized consensus, multiple broadcast processes initialized from different client proxies can be performed independently and concurrently, which facilitates parallelism and will not hurt performance.

Block Server. The *block server* keeps the complete ledger and is associated with a state database, *e.g.,* using LevelDB [3] or an in-memory hash table. The block server is responsible for transaction execution, block generation and validation, and updating the state database. It consists of a group of worker threads for executing transactions concurrently, and relies on a *reserve table* to detect concurrency conflicts, which will be discussed in Section 3.3. Each block server corresponds a client proxy for receiving transactions. The block server and the client proxy can be allocated on the same physical node for the sake of sharing transaction data through a pipeline. To provide protection from malicious behaviours, the block server broadcasts the block signature after generating a block. Only when the number of received and verified block signatures reaches a threshold, the validation succeeds. In case a BFT model with $3f + 1$ nodes, this threshold value is set as $f + 1$.

3.2 Execution Flow

Figure 3 shows the execution flow, which consists of three phases: preparation phase, execution phase, and validation phase.

3.2.1 *Preparation Phase.*

The transaction request sent from user client is accepted by a client proxy on a local peer node, where each transaction is associated with an epoch number and a transaction ID (tid) before sending to the block server for processing. After a client proxy collects a batch of local transactions, it calculates a hash value for this batch of transactions and sends the hash value to the epoch server for applying an epoch number with a signature (*i.e.,* this batch of transactions share the same epoch number). The signature is generated by the epoch server based on i) its private key, ii) the hash of the transaction batch, iii) the epoch number, and iv) a nonce. The epoch servers only need to make consensus on the increments of epoch numbers. After receiving an increased epoch number, each client proxy broadcasts the signed transactions of the last epoch to all other client proxies. It is worth noting that an epoch server instantly returns the *current* epoch number to client proxy without waiting for the completion of consensus.

Each client proxy uses a consensus protocol (PBFT or Raft) to ensure atomic broadcast. Suppose there are n peers in our system, each peer initiates an independent consensus instance for broadcasting its collected transactions set, and there are total n consensus instances running concurrently. In other words, each peer is also participating in n consensus instances. It works as a leader in one consensus instance, and works as followers in the other $n-1$ consensus instances. In order to ensure that all peers received a complete set of transactions of epoch i, each peer is required to receive the transactions set of epoch i proposed by all the other $n-1$ peers (recall that each peer initiating an independent consensus instance). If a peer does not receive the message from a certain peer for a period of time (timeout), it will initiate a "view-change" request to other peers and make consensus on whether stop waiting for the message (due to leader failure) or replicate the message from other peers (due to link failure between it and the leader). Note that, the consensus among epoch servers and that among client proxies are running independently. They do not depend on each other. The consensus on the increments of epoch numbers will not stop producing blocks.

After receiving the remote transactions, each client proxy verifies their validity and generates *tid*s for them. The client proxies generate *tid*s independently without causing extra communication. In the deterministic execution algorithm, the *tid* determines the transaction execution order. To ensure fairness, the randomness of *tid* is of vital importance, so we use hash value as *tid*. Specially, the *tid* is generated based on i) the content of the transaction and ii) the content of the whole transaction batch. In a trusted environment, only using the transaction to generate hash value is sufficient. However, since our deterministic execution tends to schedule smaller *tid* first, it will prompt malicious clients to fake transaction contents for gaining a smaller *tid* (as a result prevent other transactions' commitment). Therefore, we generate *tid* based on not only the transaction itself but also the entire batch containing other transactions. The security issues will be discussed in Section 4.

3.2.2 Execution Phase. In the execution phase, the block server first calls chaincode to execute a transaction and generate a read-write set. The read-write set is represented as a set of key-value pairs, which record a set of read or write operations. All the transactions of the same epoch are executed by multiple *worker threads* in parallel. To support concurrency control, a shared *reserve table* that reserves write operations in <*key*, *tid*> format is designed. As multiple threads concurrently update the same row (with the same *key*), the reserve table will be updated according to a deterministic rule that will be used in the deterministic commit protocol. The details of deterministic transaction processing will be further presented in Section 3.3.

Next, the successfully committed transactions with the same epoch number are used to update the state database, and both the committed and aborted transactions are used to generate a new block. The committed/aborted status of each individual transaction is immediately returned to users. Similar to other blockchain systems, a block consists of three parts: header, body, and metadata. The block header contains some verifiable data such as the previous block hash value and the hash of the root of the Merkle tree of this block's transactions. The body contains all submitted transactions and users' signatures. The metadata contains the committed/aborted status of each transaction and a set of block signatures

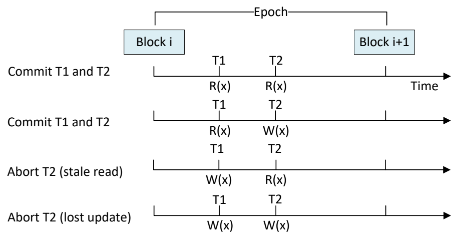


Figure 4: Possible read-write transaction orders and possible conflicts. $T1$ and $T2$ are two transactions (with read/write operations) ordered deterministically according to their *tid*s.

collected from other block servers in the validation phase. We accelerate the block generation by an asynchronous block generation technique that will be presented in Section 3.4.1.

3.2.3 Validation Phase. The validation phase is launched to validate the generated block and provide the block self-verification property. That is, a user client can verify a block's integrity and correctness based on a set of signatures provided by the local block server. After generating a block, each block server is required to generate a signature based on its private key and the hash value of the generated block. These signatures generated by different block servers are exchanged among each other. If there is no malicious behavior, the calculated block hash values should be identical. After $f+1$ signatures of the current block are collected and verified to be identical, the block server marks the block as verified.

3.3 Deterministic Transaction Processing

Blockchain is essentially a replicated transactional processing system. Deterministic database [11, 37, 41, 60] is also a typical multi-master replicated data system that synchronously replicates batches of transactions to multiple replica servers. Each replica runs the same set of ordered transactions deterministically and converts the database from the same initial state to the same final state. Inspired by deterministic database, `NeuChain` is designed to eliminate the explicit ordering phase by deterministic transaction processing.

In `NeuChain`, after the preparation phase is finished, a complete set of transactions of epoch i from all remote client proxies are collected. The set of transactions might be received in different orders, but each transaction is associated with a globally unique *tid*, which can be used to determine the execution order. With the same set of input transactions and a pre-defined deterministic order, the same output is warranted. Nevertheless, multi-thread processing is usually implemented to improve the transaction processing performance, resulting in that two replicas may process the input in different serial orders due to different thread scheduling. To address this problem, we define a deterministic conflict resolution rule.

Algorithm 1 describes the deterministic transaction processing. For each epoch, a reserve table $Table$ and a committed transactions set S_{cmt} are initialized (Line 1-2). These transactions are executed by multiple worker threads in parallel (Line 3-4). A worker thread executes a transaction based on last epoch's database snapshot and produces the transaction's read set $T.RS$ and write set $T.WS$ (Line 11). If it is a read-only transaction that has empty write set, it will not conflict with other transactions, so it is directly added to the committed set (Line 12-13). Otherwise, we update the reserve table's corresponding row to be the smallest *tid* that has ever met (Line 14-15). In other words, only the write operation with the smallest *tid* is recorded for conflict resolution.

After all transactions have been executed (Line 5), the possible conflicts are detected in parallel. As a blockchain system, the snapshot isolation naturally adapts to our transaction processing. That is, all transactions should read and write based on the last block as a database snapshot. Between two snapshots, these transactions collected in an epoch are concurrently executed in a deterministic way. Under such environment, the possible read-write transaction orders and possible conflicts are illustrated in Figure 4. i) If $T1$ and $T2$ are both read operations, there is no conflict, so both of them can be committed. ii) If $T1$ reads an item x and $T2$ updates x (*i.e.,* WAR dependency), both of them can be committed. $T1$ first reads the snapshot before $T2$ updates it, which does not impact the correctness. iii) If $T1$ updates an item x and $T2$ reads x (*i.e.,* RAW dependency), $T2$ should be aborted. This is because that $T2$ will read the data in the snapshot which is updated by an early scheduled transaction $T1$, which leads to stale read anomalies. iv) If $T1$ and $T2$ both update an item x (*i.e.,* WAW dependency), $T2$ should be aborted, since concurrent update may lead to anomalies, such as lost update. In Algorithm 1, we realize the conflict resolution by relying on the reserve table. If a transaction T's write operation is overwritten by another transaction (WAW), T is aborted (Line 17-19). If T's read data is updated by another transaction (RAW), T is aborted (Line 20-22). The remaining transactions are committed and used to generate a new block and update database (Line 8-9). It is worth noting that NeuChain does not allow to update a value multiple times in a block, which may lead to higher abort rate under high-contention workload. However, as NeuChain is very fast (only 30-75 ms to produce a block), this limitation will not impact user experience too much by resubmitting the aborted transactions.

Determinstic Reordering. Given an RAW conflict that may lead to stale read (as shown in Figure 4), we have the opportunity to commit the transaction by deterministic reordering. Inspired by [37], we can deterministically change the order from $T1 - T2$ (RAW dependency) to $T2 - T1$ (WAR dependency), such that both of them can commit. However, for a transaction that contains both WAR and RAW dependencies, the reordering algorithm cannot guarantee no loop after the reordering, so the transaction must be discarded. In short, when the dependency graph has a circle, it is not possible to commit all transactions by changing commit order, but it can ensure serializability by aborting a transaction to break the loop.

Algorithm 1: Deterministic Transaction Processing

```
Input: A set of transactions $TS$ of epoch $i$, and state database
       snapshot $DB[i-1]$
Output: A new block $B[i]$ and updated database snapshot $DB[i]$
Initialize reserve table $Table$;
Initialize committed transactions set $S_{cmt}$;
foreach $T$ in $TS$ parallel do
    ExecuteTx($T$); // execute transactions in parallel
Synchronize with all worker threads;
foreach $T$ in $TS$ parallel do
    DetectConflict($T$); // detect conflicts in parallel
Update state database to a new snapshot $DB[i]$ based on $S_{cmt}$;
Generate new block $B[i]$ based on $TS$ and execution result;
Function ExecuteTx ($T$):
    $\{T.RS, T.WS\} \leftarrow$ execute transaction $T$ based on $DB[i-1]$;
    if $T.WS == \emptyset$ then
        Add $T$ to $S_{cmt}$ and return; // read-only transaction
    foreach $rec$ in $T.WS$ do
        $Table[rec.key] = min(Table[rec.key], T.tid)$;
Function DetectConflict ($T$):
    for each $rec$ in $T.WS$ do
        if $Table[rec.key] < T.tid$ then
            Abort $T$ and return; // $T$'s write is overwritten
    for each $rec$ in $T.RS$ do
        if $Table[rec.key] < T.tid$ then
            Abort $T$ and return; // $T$'s read data is updated
    Add $T$ to $S_{cmt}$ and return;
```

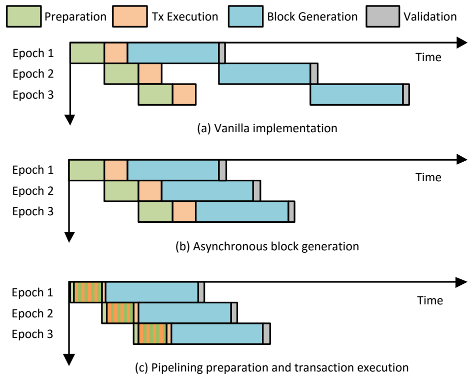


Figure 5: Asynchronous block generation and pipelining.

3.4 Implementation

We implement a prototype of NeuChain with ~20,000 lines C++ code. It uses pluggable LevelDB or in-memory key-value store as the state database and stores blocks in files. We present two key optimization techniques in the following.

3.4.1 Asynchronous Block Generation. In NeuChain, the preparation phase does not depend on any previous result, so it can start as soon as the next epoch starts according to physical time. Under high contention, it is possible that the block of the previous epoch has not been generated yet before the next epoch starts. Thus, the preparation phase may be overlapped with previous epoch's execution phase in vanilla implementation as shown in Figure 5a.

In existing blockchain systems, a block is usually generated during the ordering phase (*e.g.,* by a single node in Ethereum [65] or by the ordering service in Fabric [14]). The generation of next block does not start before the previous block is generated. In NeuChain, the block generation is the most time consuming step (see Figure 9b). We optimize it with *asynchronous block generation.* The next block generation can start before the previous block generation finishes. This process is visually illustrated in Figure 5b. Multiple block generation threads of different epochs are running concurrently if high contention persists. The block headers (containing previous block's hash) are generated serially to support tamper resistance.

3.4.2 Pipelining Preparation and Execution. Recall that the client proxies exchange their collected local transactions and send them to block server for further execution. There is a synchronization point that requires to receive the transactions of the same epoch from all client proxies before entering to the transaction execution phase. However, since deterministic execution is leveraged, within the same epoch the execution result is deterministic despite the transactions arrive in arbitrary order. This nice property brings us new opportunities to improve performance. Specifically, as shown in Figure 5c, we pipeline the preparation phase and the transaction execution phase by feeding mini-batches for transaction execution. The early arrived transactions are immediately used to update the reserve table (Line 3-4 in Algorithm 1). In this way, the preparation phase and the execution phase can be overlapped. Note that there is still a synchronization point at Line 5 in Algorithm 1 to receive all remote transactions before performing conflict detection.

4 TRUST AND SECURITY

NeuChain introduces several new components such as epoch server and client proxy, which could bring potential security risks. This section discusses the possible security risks caused by malicious threats and how to make it robust to these threats. Before discussing specific risks, we first introduce our trust model and assumptions.

Trust Model and Assumptions. NeuChain can be deployed in multiple organizations, each running one or more physical servers. A client proxy and a block server can be setup on a single physical server for performance or on separate physical servers for isolation. In each organization, the client proxies serve the transaction requests from local users, and the block servers maintain all blocks. A server is selected by all organizations as the single epoch server, or else multiple servers contributed by multiple organizations form a cluster epoch servers, where the consensus on the epoch number increment among all epoch servers is achieved through PBFT or Raft protocol. Under such deployment, our trust model assumptions can be summarized as follows. i) The number of client proxies/epoch servers is more than $3f + 1$ where f is the number of Byzantine client proxies/epoch servers. ii) Servers are connected with partial asynchronous networks. iii) Servers within the same organization can trust each other, while servers in other organizations have possibilities to carry out malicious intents.

4.1 Malicious Client Proxy

A malicious client proxy can prevent other client proxies from collecting the complete set of transactions in the following ways:

- No response to some of other client proxies;
- No response to all other client proxies;
- Send different sets of transactions to different client proxies;
- Tamper with epoch server's reply and user's transaction content.

The first three malicious behaviours can be prevented by *atomic broadcast* [24], *i.e.,* forcing all correct nodes to receive the same set of messages, which can be achieved by PBFT consensus protocol [17]. In NeuChain, each client proxy broadcasts transaction batches in an independent PBFT cluster, and multiple overlapped PBFT clusters led by different client proxies run concurrently (see Section 3.2.1). If a client proxy that broadcasts transactions working as leader fails or acts as a malicious node, it would send inconsistent transaction batches, which will be detected by honest followers. This will trigger view change in this PBFT cluster. Regarding the fourth problem, it can be prevented by associating each transaction with the signature of the user that initiates the transaction.

4.2 Malicious Epoch Server

Setting up a single epoch server is fragile. The epoch server may perform the following malicious activities.

- No response to client proxy;
- Hinder growth of global epoch;
- Respond client proxy a future epoch number;
- Respond client proxy a stale epoch number.

To make the epoch service robust to crash failures and Byzantine attacks (*i.e.,* the first two issues), we use multiple epoch servers to construct a cluster that runs PBFT protocol [17]. The PBFT protocol prevents the Byzantine nodes from hindering the growth of the global epoch. However, malicious nodes can still provide arbitrary epoch number to a client proxy (the third and fourth malicious activities). Under normal circumstances, the epoch number returned by the same epoch server should be increased monotonically. Suppose a client proxy receives an epoch number e from an epoch server, any successive responses received from that epoch server will never be smaller than e. Thus, the key is to guarantee the monotonically increased epoch number on all client proxies. We force each client proxy to collect at least $f + 1$ identical epoch numbers. For a client proxy, if the received $f + 1$ identical epoch numbers are smaller than its last one (this may happen when the PBFT consensus is in progress but not completed), it keeps acquiring the up-to-date epoch numbers from epoch servers until collecting $f + 1$ identical increased epoch numbers.

4.3 Malicious User

A malicious user may conduct the following malicious activities.

- Replay attack;
- Tampering with transaction ID.

To prevent replay attack, we require user client to add a *nonce* for each transaction message, which restricts user client to send the same transaction only once [39]. In addition, our deterministic conflict resolution algorithm aborts a transaction with a larger *tid* if it reads or writes an item that is updated by a transaction with a smaller *tid*. This will prompt users to apply a smaller *tid*. We let client proxy generate *tid*s and add randomness to the *tid* generation as follows. A *tid* is generated as the content hash value of not only this transaction but also the corresponding transaction batch (which is unpredictable). The integrity of the transaction and the corresponding batch are ensured by user's signature and epoch server's signature, respectively. In this way, the client cannot predict the hash value *tid* (large or small), as a result there is no motivation to tamper with transaction IDs.

4.4 Malicious Block Server

Since the client proxy has already established a consensus on the input transactions (that are not executed yet and will be contained in a block), the block server cannot tamper with the transaction contents of the transactions. However, the execution result generated by a single block server is untrusted as it might be a Byzantine node. The execution results consist of the read-write sets and the committed or aborted status. A malicious block server may conduct the following activities.

- Tampering with state database;
- Tampering with block contents;
- Return fake committed/aborted notifications to users.

To ensure the correctness and verifiability of the execution results, we introduce the blockchain *self-verification* feature [15], *i.e.,* acquiring from a single node is enough to obtain information that can prove the validity of block and restore the whole state database. All honest nodes exchange signatures on a specific block to make it self-verifiable. If a malicious node generates an invalid block by tampering with the state database or block contents, other nodes cannot verify the signature of this block, and the block server cannot collect $f + 1$ valid signatures to achieve self-verification. In addition, a user client sends requests to multiple block servers and must receive at least $f + 1$ identical committed or aborted responses to ensure the trusted result.

5 EXPERIMENTAL RESULTS

In this section, we evaluate NeuChain from different dimensions. The experiment setup is briefly described as follows.

Physical Environment. We build a geo-distributed cluster and a local cluster on Aliyun. Our geo-distributed cluster contains 8 nodes, including 2 nodes in Zhangjiakou city (north China), 2 nodes in Chengdu city (west China), 2 nodes in Hangzhou city (east China), and 2 nodes in Shenzhen city (south China). For Fabric and FastFabric, one node in each region works as orderer, and the other one works as peer node. While in NeuChain and its variants, one node in each region works as epoch server, and the other runs client proxy/block server. Our local cluster contains 12 locally connected nodes. In Fabric and its variants, 4 nodes work as orderers, and the other 8 work as peers. In NeuChain and its variants, 4 nodes work as epoch servers, and the other 8 work as client proxies/block servers. In both clusters, each node (ecs.hfc6.4xlarge instance) is equipped with 16 vCPUs, 32GB DRAM, running Ubuntu 20.04 LTS. The network bandwidth among cross-region nodes is about 100 Mbps, and that among local nodes is about 5 Gbps.

Competitors and Configurations. NeuChain is compared with four state-of-the-art blockchain systems, including Hyperledger Fabric v2.2 [14], FastFabric [30], Meepo [75] (a sharded consortium blockchain), ResilientDB [32] (a highly scalable blockchain with OEV architecture), and Basil [56] (a leaderless BFT key-value store that relies on EOV architecture). The consensus protocols used in NeuChain, Fabric, and FastFabric are all Raft. Meepo relies on Proof of authority (PoA) [23] consensus protocol. FastFabric, ResilientDB and Basil are optimized for high-throughput transactions. They use in-memory hash table to maintain database state. Fabric and NeuChain both rely on LevelDB as state database, while Meepo uses RocksDB [10]. Note that, Basil does not generate blocks which is not a blockchain. The other systems all generate blocks and store them on disk. To ensure fairness, all these systems are configured with memory storage to maintain database and blocks. For other settings, we use their default configurations.

NeuChain Variants. To study the performance difference of various architectures without being affected by implementation differences, we also implement an OEV architecture, an EOV architecture, and an OEPV architecture under the same implementation framework of NeuChain, which originally supports EV architecture. The default epoch length in NeuChain is 50ms for local cluster experiments and 75ms for geo-distributed experiments.

Workload. We use two popular benchmarks, YCSB [20] and SmallBank [8]. For YCSB workload, we use one table with 10 columns and 1,000,000 rows, with each column size as 100 bytes. We choose YCSB-A (50% read and 50% write), YCSB-B (95% read and 5% write), and YCSB-C (100% read) in our experiments. All YCSB requests from clients are subject to Zipf distribution with a skew factor of 0.99. SmallBank simulates basic bank transfer operations. We configure the SmallBank benchmark with 100,000 accounts. The access pattern to these accounts follows a uniform distribution. We use Blockbench [26] as the test tool.

5.1 Overall Performance

We compare NeuChain with the competitor systems and our system's variants under YCSB-A, YCSB-B, YCSB-C, and SmallBank workload. We measure their peak throughput results and corresponding latencies. All experiments are performed three times on our geo-distributed cluster and local cluster.

Figure 6 and Figure 7 report the performance results (*w.r.t.* throughput, latency, and abort rate) with error bars on geo-distributed cluster and local cluster, respectively. We can see that NeuChain exhibits the highest throughput than the others under all workloads. The main reason for good performance can be attributed to the elimination of the ordering phase. The ordering phase, no matter in OEV, EOV or OEPV architectures, requires a single node to propose a block of ordered transactions to other followers, which could be a bottleneck due to its limited upstream bandwidth. While in the EV architecture, each node initiates an independent consensus instance for its own transactions subset, and multiple consensus instances are running concurrently, so the throughput of EV is not limited by the upstream bandwidth of a single node. In addition, NeuChain only transfers the original transaction requests without the generated read/write set and the corresponding signatures in the œ and the EOV architectures. This reduces the message size, which also helps improve the throughput. These two factors make NeuChain faster under these workloads.

In Figure 6, NeuChain shows a bit higher latency than Fabric, *e.g.,* 230 ms vs. 202 ms under SmallBank workload. In NeuChain, a client proxy asks multiple remote epoch servers for epoch numbers. This means that it requires one more round-trip time (RTT). This is the reason why NeuChain shows higher latency than Fabric and its variants. Meepo shows high latency due to the high latency for processing the cross-shard transactions. OEV shows the highest latency because it generates blocks synchronously and sequentially. Basil exhibits the lowest latency because it does not generate blocks and does not maintain the ledger's history.

NeuChain shows higher abort rate than others, especially under write-intensive workloads. This can be attributed to the deterministic transaction processing method. The transactions with RAW and WAW conflicts are aborted for parallel execution, which is another reason for high performance. While in Fabric, the transactions are validated sequentially, so the abort rate is low. The NeuChain variants use the same concurrency control algorithm as NeuChain, so they show higher abort rates than the competitors. In addition, NeuChain shows higher abort rate under YCSB than SmallBank, because YCSB follows Zipf distribution (resulting in higher probability of conflicts) and SmallBank follows uniform distribution.

5.2 Bottleneck Analysis

To study the reason of significant performance improvement over the widely used Fabric, we perform bottleneck analysis in this experiment. As our analysis in Section 1, the ordering phase requires a single node to propose a block of ordered transactions, which

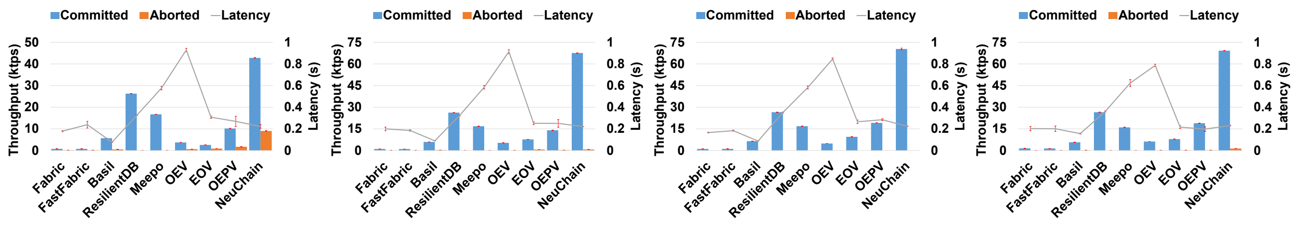

(a) YCSB-A (b) YCSB-B (c) YCSB-C (d) SmallBank

Figure 6: Performance comparison of different blockchains (on geo-distributed cluster).

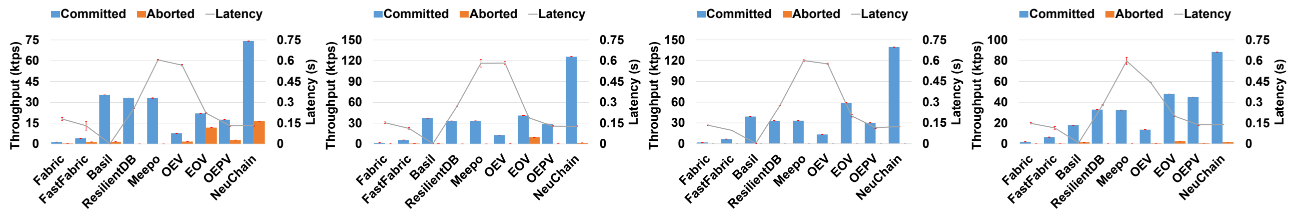

(a) YCSB-A throughput (b) YCSB-B throughput (c) YCSB-C throughput (d) SmallBank throughput

Figure 7: Performance comparison of different blockchains (on local cluster).

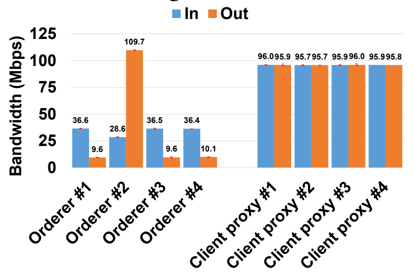


Figure 8: Bandwidth consumption in Fabric and NeuChain on geo-distributed cluster (SmallBank).

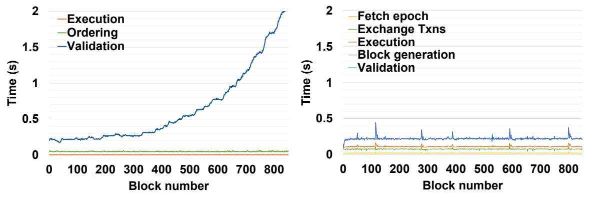

(a) Fabric (b) NeuChain

Figure 9: Runtime Breakdown on local cluster (SmallBank).

could be the bottleneck. We conduct experiments to measure the bandwidth in NeuChain and Fabric on our geo-distributed cluster. Figure 8 shows the (in and out) bandwidth of each client proxy in NeuChain and each orderer in Fabric. In the consensus group of orderers, a single orderer (orderer #2) proposes a block of ordered transactions to other follower orderers. The system throughput is limited by the maximum out bandwidth (100 Mbps) of the leader (i.e., orderer #2) as shown in Figure 8. While in NeuChain, each client proxy accepts its own local transactions and independently proposes them to other client proxies. Multiple client proxies propose transactions and receive them at the same time concurrently. The workload is evenly distributed among client proxies.

Another reason is also attributed to the elimination of the explicit ordering phase. The *arbitrary* order defined by the orderer could limit the parallelism. For example in Fabric, given a block of executed transactions and the order defined by the ordering service, each peer should validate these transactions in terms of the serial order. It is non-trivial to validate them in terms of the serial order in a parallel manner and at the same time guarantee the consistency among nodes. Fabric validates these transactions

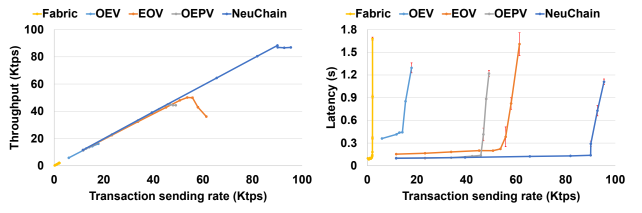

(a) Throughput (b) Latency

Figure 10: Throughput and latency when varying transaction arrival rate (SmallBank).

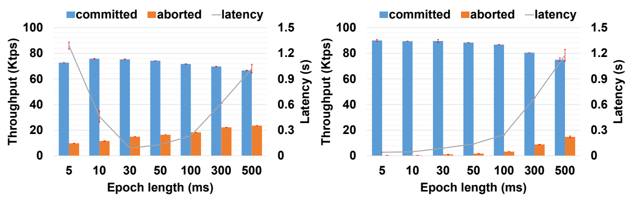
in a sequential manner, which could be a bottleneck under high-contention workload. We verify this through a runtime breakdown experiment on the local cluster where network bandwidth is high enough. As shown in Figure 9a, as blocks are generated continuously, the execution and the ordering time are stable, but the validation time is longer and longer. The sequential validation becomes the bottleneck limiting throughput. In NeuChain, We rely on a deterministic transaction processing rule to execute the out-of-order transactions, which also guarantees consistency. To allow for parallel execution, the deterministic execution rule (that is followed by all block servers) aborts some transactions (with RAW and WAW dependencies). As shown in Figure 9b, the runtime results of all phases in NeuChain are stable without obvious bottlenecks.

5.3 Varying Transaction Arrival Rate

In this experiment, we vary the transaction arrival rate and see how the throughput and latency change accordingly. The arrival rate can be adjusted by configuring the client nodes. We test NeuChain and its variants to measure how much is gained from our novel architecture excluding the implementation effects. ResilientDB, Basil, and Meepo do not support adjusting the arrival rate. We also test Fabric as a baseline.

Figure 10a shows the throughput results of SmallBank. With an increase in transaction arrival rate, the throughput of any system increases linearly as expected till it flattens out or drops, which is the peak throughput (system utilization close to 100%).

Fabric first reaches its peak throughput at around 1.99 Ktps. Among NeuChain variants, OEV first reaches its peak throughput at around 13.6 Ktps for SmallBank due to the high redundant execution cost. The throughput of EOV drops more sharply than others



(a) YCSB-A (b) SmallBank

Figure 11: Performance when varying epoch length.

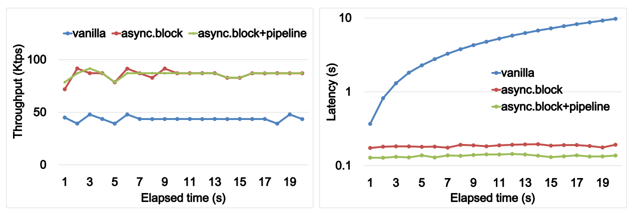
when the arrival rate is above the peak throughput 53.67 Ktps. The reason is that the OCC in EOV tends to abort more transactions under high contention. OEPV gains peak throughput at around 44.91 Ktps. We observe the highest peak throughput when running EV. It flattens out at 88.31 Ktps. Figure 10b shows the latency results of SmallBank, respectively. When the arrival rate is below the peak throughput, the latency increases gently since it takes longer time to processing more and more transactions in a block. When the arrival rate is above the peak throughput, the latency increases significantly, because the transactions are overstocked and waiting for being served in the queue. In addition, before the latency goes abnormally high, most of the variants show similar latency around 0.15 second, but the latency of OEV is a little bit higher.

5.4 Varying Epoch Length

NeuChain uses epochs to divide the transactions into transaction batches. Each epoch of transactions are used to generate a block. In other words, the epoch length corresponds to the block size, which is critical to the system throughput and latency. The epoch length can be adjusted by the epoch servers. In this experiment, we fix the transaction arrival rate and vary the epoch length from 5 ms to 500 ms to see the effect of epoch length. Figure 11 shows the committed and aborted throughput and latency results under YCSB-A and SmallBank workloads with different epoch lengths. With a short epoch (5 ms), the latency is the highest. The reason is that the frequent data exchanges (due to short epoch) are expensive and prolong the overall runtime. The abort rate is the lowest since the transactions in a small block (due to short epoch) is less likely to conflict with others. With the longest epoch (500 ms), the latency is still very high. Because the cost of Merkle tree generation is exponentially increased with the increase of block size, which prolongs the block generation time and latency. In addition, the abort rate is high with long epoch due to more intra-block conflicts.

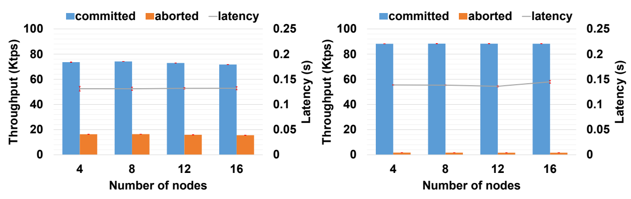
5.5 Effect of Optimizations

As shown in Figure 9b, the block generation is the most time consuming step. It takes even longer time for write-intensive workload (*e.g.,* YCSB-A workload). This can slow down the processing speed and may accumulate more and more blocked transactions with high arrival rate. To address this problem, we propose asynchronous block generation to speedup this process in Section 3.4.1. We also propose the pipelining technique to overlap the preparation phase and the transaction execution phase in Section 3.4.2, which can further help decrease the latency. Figure 12a shows the throughput results with or without optimizations during the running process. The asynchronous block generation has greatly improved the throughput of vanilla implementation without optimizations, say about 2 times on SmallBank, but the pipelining does not improve the throughput. Figure 12b shows the latency results. Without asynchronous block generation, the latency is increased significantly as time passes (note that y-axis is log plot) since the server node is overloaded under high contention. By asynchronous



(a) Throughput (b) Latency (log-scaled)

Figure 12: Effect of optimizations (SmallBank).



(a) YCSB-A (b) SmallBank

Figure 13: Scaling performance.

block generation, the latency can be made stable with the same transaction arrival rate. Furthermore, the latency can be reduced with pipelining.

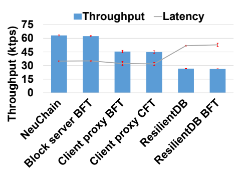
5.6 Scalability

The blockchain systems suffer from the drawback of scalability. Due to its replicated property and verifiable requirement, the redundant storage and execution overhead can be inevitable. Another key step that limits scalability is the consensus on transaction order. Though sharding can improve the scalability by reducing the consensus cost, it also results in additional coordination overhead among shards. NeuChain eliminates the explicit ordering phase where a single orderer to propose a block of ordered transactions and relies on deterministic transaction processing to avoid the single orderer bottoleneck, which helps improve scaling performance. In this experiment, we increase the number consensus nodes from 4 to 16 and see scaling performance. Figure 13a and Figure 13b show the throughput and latency results under YCSB-A and SmallBank, respectively. For YCSB-A workload, when node number scales from 4 to 16, the throughput decreases from 74.2 Ktps to 71.7 Ktps. For SmallBank workload, the throughput keeps stable at around 88.3 Ktps. Similarly, the latency results do not change much.

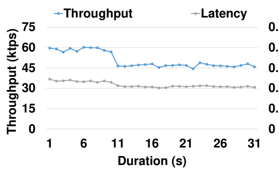
5.7 Performance under Failure Cases

We conduct experiments with Byzantine and crash failures, and compare the performance with ResilientDB under failure cases. NeuChain provides both Byzantine failure tolerance (NeuChain-BFT) and crash failure tolerance (NeuChain-CFT), while ResilientDB provides non-leader Byzantine failure tolerance (ResilientDB-BFT). We kill a client proxy to simulate crash failure and let a client proxy send fake messages to simulate Byzantine failure. The performance results (with respect to throughput and latency) under SmallBank workload are reported in Figure 14a. We observe that under these failure cases, NeuChain still outperforms ResilientDB.

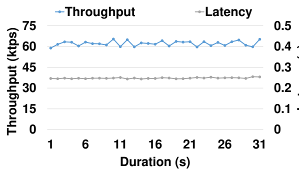
Figure 14b shows the change of throughput and latency under a malicious attack (tampering with the content of a transaction) which is manually injected by a client proxy after 10 seconds. After the other peers receive this message, they will identify the fake transaction by checking the user's signature, followed by starting a view-change process to abandon the consensus group leaded by the Byzantine client proxy. Since then, the Byzantine client proxy is forbidden to submit transactions, so the overall throughput is



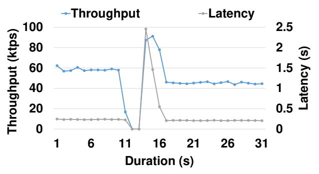
(a) Failure comparison



(b) Client proxy BFT



(c) Block server BFT



(d) Client proxy CFT

Figure 14: Performance under failure cases (SmallBank).

lower. Figure 14c shows the results under attacks by a Byzantine block server, which provides other block servers with a fake block signature. The other block servers will reject the following verification requests from the Byzantine block server. This will not impact the throughput and latency since the notification of commit/abort has already been responded to before the validation phase. Figure 14d shows the results under a crash failure (a node is manually shut down) occurred at 10 second. The other peers wait for the messages from the failed node for a time period (timeout). Then they make agreement on removing this failed node. The timeout mechanism leads to the drop of throughput and the increase of latency. But the other consensus groups are still running. The transactions are cached locally without commitment. Once they agree on removing the failure node, the cached transactions are processed immediately, which leads to a sharp increase of the throughput. Later, the throughput is a little lower than before and keeps stable because one server node that accepts user requests is removed.

6 RELATED WORK

Database for Blockchain. Blockchain, as a replicated ledger database, has attracted much attention in database community. The transaction processing techniques from traditional database have been borrowed to improve the throughput and latency performance of blockchains. Besides the works mentioned in Section 2, there are other representative works in this field, such as XOXFabric [29] and AHL [21]. The authors in [40] present a comprehensive dichotomy between blockchains and distributed databases. On the other hand, as authenticated query processing has been extensively studied in the database community [18, 66, 67], researchers recently put great efforts on designing new authenticated data structures (ADS) to support verifiable queries on blockchains. Representative works include LineageChain [50, 51], vChain [63, 68], FalconDB [43], P^{2}B-Trace [44], authenticated keyword search [73], and authenticated data structure GEM^2-Tree for range queries [74].

Blockchain-based Database. The blockchain essentially has no querying abilities when compared to traditional database. A number of studies add blockchain features to a traditional database to construct a blockchain-based database, including BigChainDB [38], ModexBCDB [4], Postchain [6], BigChainDB [38], Postchain [6], LedgerDB [70], BRD [1], CovenantSQL [2], and ChainifyDB [54]. These systems rely on their built-in databases for concurrency control support and underlying storage. Thus, they are featured with high throughput and low latency. Furthermore, these systems support both the UTXO and account-based models and support storing various types of objects in database systems. However, since these systems rely on traditional database, they cannot protect their local state with authenticated data structure (ADS) like Merkle trees.

Deterministic Execution. In a deterministic database [27, 28, 37, 46, 47, 59, 60], each replica runs the same set of ordered transactions deterministically, and converts the database from the same initial state to the same final state. The incoming transactions are assigned with unique values before passing to replicas, and the executions on different replicas are based on the serial order of the unique values. In this way, replicas do not need to coordinate with each other to remain consistent. To exploit parallelism when processing each replica, deterministic concurrency control protocol that employs deterministic ordered locks [59, 60] or dependency graphs [27, 28, 37] can be employed. This paper leverages deterministic execution to propose an ordering-free blockchain. Canopus [49] is a hierarchical consensus protocol that also lets each node collect local transactions and exchange them with other nodes. NeuChain, as a transactional blockchain system, can borrow the hierarchical consensus structure of Canopus to improve scalability (but at the expense of latency). Furthermore, NeuChain relies on epoch servers to decide block boundaries, and relies on deterministic transaction processing to resolve conflicts with consistent states.

Sharding Blockchains. The sharding technique has been adopted in many blockchain systems, such as Monoxide [64], OmniLedger [36], RapidChain [72], ByShard [34], QuarkChain [9], and [21]. They partition nodes into multiple groups (*a.k.a.* shards), with each group of nodes running their own consensus instances. The atomicity of cross-shard transactions should be ensured through 2PC or BFT-based replication. Sharding is mostly used in public blockchains for improving scalability. However, since the security of blockchains depends on the assumption that the number of failures is below a certain threshold, the shard formation protocol must ensure the security of each shard. NeuChain improves the performance by running multiple consensus instances concurrently, each on a subset of transactions. To sum up, in sharding blockchains the consensus is parallelized among different groups of nodes, while in NeuChain the consensus is parallelized among different data.

7 CONCLUSIONS AND FUTURE WORK

This paper presents an order-free execute-validate blockchain architecture by leveraging deterministic execution. By eliminating the explicit ordering consensus, the system throughput and latency can be greatly improved. Based on the EV architecture, we further design and implement a system prototype NeuChain. The cost of deterministic execution and block generation is alleviated by pipelining and asynchronous block generation. In addition, a series of mechanisms are integrated to NeuChain to provide trust and security guarantee. Our geo-distributed experimental results show that NeuChain outperforms Fabric by a factor of 47.2-64.1X, ResilientDB by a factor of 1.6-2.7X, and Basil by a factor of 7.6-12.2X. As part of our ongoing work, we are extending NeuChain to support authenticated query processing in an untrusted environment.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (62072082, U1811261, 62072086), the Key R&D Program of Liaoning Province (2020JH2/10100037), the National Science and Technology Major Project (J2019-IV-0002-0069), the National Social Science Foundation of China (21&ZD124), and the Fundamental Research Funds for the Central Universities (N2216015). Yanfeng Zhang is the corresponding author.

REFERENCES

[1] 2021. BRD: bitcoin wallet. https://brd.com/

[2] 2021. CovenantSQL: The Blockchain SQL Database. https://covenantsql.io/

[3] 2021. LevelDB: a fast key-value storage library. https://github.com/google/leveldb

[4] 2021. Modex Blockchain Database (BCDB). https://modex.tech/

[5] 2021. Peercoin: The Pioneer of Proof of Stake. https://www.peercoin.net/

[6] 2021. Postchain. https://postchain-docs.readthedocs.io/en/latest/

[7] 2021. Quorum: A permissioned implementation of Ethereum supporting data privacy. https://github.com/ConsenSys/quorum

[8] 2021. SmallBank Benchmark. http://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/

[9] 2022. QuarkChain: A flexible, scalable, and user-oriented blockchain. https://quarkchain.io/

[10] 2022. RocksDB: A persistent key-value store for fast storage environments. http://rocksdb.org/

[11] Daniel J. Abadi and Jose M. Faleiro. 2018. An Overview of Deterministic Database Systems. *Commun. ACM* 61, 9 (2018), 78–88.

[12] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS 2019)*. 1337–1347.

[13] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021)*. 76–88.

[14] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys 2018)*. Article 30.

[15] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. 2020. From Byzantine Replication to Blockchain: Consensus is Only the Beginning. (2020), 424–436.

[16] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2009. Serializable isolation for snapshot databases. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–42.

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

[18] Qian Chen, Haibo Hu, and Jianliang Xu. 2015. Authenticated online data integration services. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. 167–181.

[19] Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Xiaodong Qi, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2021. SChain: a scalable consortium blockchain exploiting intra-and inter-block concurrency. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2799–2802.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*. 143–154.

[21] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data (SIGMOD 2019)*. 123–140.

[22] Sourav Das, Vinith Krishnan, and Ling Ren. 2020. Efficient Cross-Shard Transaction Execution in Sharded Blockchains. *ArXiv* abs/2007.14521 (2020).

[23] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. 2018. PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain. (2018).

[24] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 36, 4 (2004), 372–421.

[25] Assunta Di Vaio and Luisa Varriale. 2020. Blockchain technology in supply chain management for sustainable performance: Evidence from the airport industry. *International Journal of Information Management* 52 (2020), 102014.

[26] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 2017)*. 1085–1100.

[27] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1190–1201.

[28] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017), 613–624.

[29] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2020)*. 1–9.

[30] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2020. FastFabric: Scaling hyperledger fabric to 20 000 transactions per second. *International Journal of Network Management* 30, 5 (2020), e2099.

[31] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2021. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. *ArXiv* abs/1911.00838 (2021).

[32] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proceedings of the VLDB Endowment* 13, 6 (2020), 868–883.

[33] Suyash Gupta and Mohammad Sadoghi. 2021. Blockchain transaction processing. *arXiv preprint arXiv:2107.11592* (2021).

[34] Jelle Hellings and Mohammad Sadoghi. 2021. ByShard: Sharding in a Byzantine Environment. *Proc. VLDB Endow.* 14, 11 (2021), 2230–2243.

[35] Ricardo Jiménez-Peris, Marta Patino-Martinez, and Sergio Arévalo. 2000. Deterministic scheduling for transactional multithreaded replicas. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems (SRDS 2000)*. 164–173.

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

[37] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.

[38] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. 2016. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB* (2016).

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

[40] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1539–1552.

[41] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015)*. 677–689.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 2014)*. 305–319.

[43] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based collaborative database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020)*. 637–652.

[44] Zhe Peng, Cheng Xu, Haixin Wang, Jinbin Huang, Jianliang Xu, and Xiaowen Chu. 2021. P2B-Trace: Privacy-Preserving Blockchain-based Contact Tracing to Combat Pandemics. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021)*. 2389–2393.

[45] Ji Qi, Xusheng Chen, Yunpeng Jiang, Jianyu Jiang, Tianxiang Shen, Shixiong Zhao, Sen Wang, Gong Zhang, Li Chen, Man Ho Au, and Heming Cui. 2021. Bidl: A High-Throughput, Low-Latency Permissioned Blockchain Framework for Datacenter Networks. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*. 18–34.

[46] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*. 180–194.

[47] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1747–1761.

[48] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment* 7, 10 (2014), 821–832.

[49] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. 2017. Canopus: A scalable and massively parallel consensus protocol. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies (CoNext 2017)*. 426–438.

[50] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment* 12, 9 (2019), 975–988.

[51] Pingcheng Ruan, Tien Tuan Anh Dinh, Qian Lin, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2021. LineageChain: a fine-grained, secure and efficient data provenance system for blockchains. *The VLDB Journal* 30, 1 (2021), 3–24.

[52] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020)*. 543–557.

[53] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. 2020. Blockene: A High-throughput Blockchain Over Mobile Devices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 567–582.

[54] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divya Agrawal. 2021. chainifyDB: How to get rid of your Blockchain and use your DBMS instead.. In *CIDR*.

[55] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: The Case of Hyperledger Fabric. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD 2019)*. 105–122.

[56] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (Transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP 2021)*. 1–17.

[57] Alex Tapscott and Don Tapscott. 2017. How blockchain is changing finance. *Harvard Business Review* 1, 9 (2017), 2–5.

[58] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 70–80.

[59] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 70–80.

[60] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD 2012)*. 1–12.

[61] Gregor Ulm, Simon Smith, Adrian Nilsson, Emil Gustavsson, and Mats Jirstrand. 2021. OODIDA: on-board/off-board distributed real-time data analytics for connected vehicles. *Data Science and Engineering* 6, 1 (2021), 102–117.

[62] Hoang Tam Vo, Ashish Kundu, and Mukesh K Mohania. 2018. Research Directions in Blockchain Data Management and Analytics.. In *EDBT*. 445–448.

[63] Haixin Wang, Cheng Xu, Ce Zhang, and Jianliang Xu. 2020. vChain: A Blockchain System Ensuring Query Integrity. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020)*. 2693–2696.

[64] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchain with Asynchronous Consensus Zones. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI 2019)*. 95–112.

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

[66] Cheng Xu, Qian Chen, Haibo Hu, Jianliang Xu, and Xiaojun Hei. 2017. Authenticating aggregate queries over set-valued data with confidentiality. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 30, 4 (2017), 630–644.

[67] Cheng Xu, Jianliang Xu, Haibo Hu, and Man Ho Au. 2018. When query authentication meets fine-grained access control: A zero-knowledge approach. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD 2018)*. 147–162.

[68] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data (SIGMOD 2019)*. 141–158.

[69] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling Blockchain Transactions through off-Chain Storage and Parallel Processing. (2021).

[70] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. 2020. LedgerDB: a centralized ledger database for universal audit and verification. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3138–3151.

[71] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC 2019)*. 347–356.

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

[73] Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, and Byron Choi. 2021. Authenticated Keyword Search in Scalable Hybrid-Storage Blockchains. In *2021 IEEE 37th International Conference on Data Engineering (ICDE 2021)*. 996–1007.

[74] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. 2019. GEM^2-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain. In *Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE 2019)*. 842–853.

[75] Peilin Zheng, Quanqing Xu, Zibin Zheng, Zhiyuan Zhou, Ying Yan, and Hui Zhang. 2021. Meepo: Sharded consortium blockchain. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 1847–1852.