Optimization of High-Concurrency Conflict Issues in Execute-Order-Validate Blockchain

MA Qianli[1], ZHANG Shengli[1], WANG Taotao[1], YANG Qing[1], WANG Jigang[2]

(1. Shenzhen University, Shenzhen 518000, China;
2. ZTE Corporation, Shenzhen 518057, China)

DOI: 10.12142/ZTECOM.202402004

https://kns.cnki.net/kcms/detail/34.1294.TN.20240621.1419.006.html, published online June 21, 2024

Manuscript received: 2024-03-23

Abstract: With the maturation and advancement of blockchain technology, a novel execute-order-validate (EOV) architecture has been proposed, allowing transactions to be executed in parallel during the execution phase. However, parallel execution may lead to multi-version concurrency control (MVCC) conflicts during the validation phase, resulting in transaction invalidation. Based on different causes, we categorize conflicts in the EOV blockchain into two types: within-block conflicts and cross-block conflicts, and propose an optimization solution called FabricMan based on Fabric v2.4. For within-block conflicts, a reordering algorithm is designed to improve the transaction success rate and parallel validation is implemented based on the transaction conflict graph. We also merge transfer transactions to prevent triggering multiple version checks. For cross-block conflicts, a cache-based version validation mechanism is implemented to detect and terminate invalid transactions in advance. Experimental comparisons are conducted between FabricMan and two other systems, Fabric and Fabric++. The results show that FabricMan outperforms the other two systems in terms of throughput, transaction abort rate, algorithm execution time, and other experimental metrics.

Keywords: blockchain; MVCC conflict; reordering; parallel validation; transaction merging

Citation (Format 1): MA Q L, ZHANG S L, WANG T T, et al. Optimization of high-concurrency conflict issues in execute-order-validate blockchain [J]. ZTE Communications, 2024, 22(2): 19–29. DOI: 10.12142/ZTECOM.202402004
Citation (Format 2): Q. L. Ma, S. L. Zhang, T. T. Wang, et al., "Optimization of high-concurrency conflict issues in execute-order-validate blockchain," *ZTE Communications*, vol. 22, no. 2, pp. 19–29, Jun. 2024. doi: 10.12142/ZTECOM.202402004.

1 Introduction

Blockchain is essentially a form of distributed ledger technology, and the popularity of blockchain technology began with the emergence of Bitcoin[1] . The true reason for this popularity is that blockchain enables peer-to-peer transactions without the need for a trusted third party. With the advent of smart contracts in Ethereum[2], blockchain technology has been extensively researched and applied in various fields such as finance[3], healthcare[4–5], supply chain[6], and the Internet of Things[7], leveraging its characteristics of decentralization, immutability, and traceability.

From the perspective of participants, blockchain systems can be divided into permissioned chains and permissionless chains. Permissionless chains, also known as public chains, allow any node to anonymously participate. Due to the unknown identities of the nodes and mutual distrust, such blockchain systems often use proof of work or other consensus mechanisms to solve the Byzantine fault tolerance consensus problem[8]. On the other hand, permissioned chains consist of a group of identity-verified nodes. These systems are often only applied to specific scenarios where the nodes, although not entirely trusting each other, share common goals. Permissioned chains constrain participating nodes and can control the read and write permissions of different nodes, making them more suitable for enterprise-level applications.

However, whether they are permissionless chains like Bitcoin and Ethereum, or permissioned chains like Tendermint and Quorum, most mainstream blockchain systems use active replication[9]: First, transactions are sorted through consensus protocols or atomic broadcast and packaged into blocks for dissemination to nodes; then all nodes execute transactions in sequence, changing their ledger states. We call this system the order-execute (OE) architecture, and its limitation lies in the fact that all nodes must execute all transactions serially in order, which is undoubtedly a limitation on throughput. In order to achieve better parallelism in transaction execution, a new execute-order-validate (EOV) architecture has been proposed. In an EOV system, clients send transaction proposals to multiple nodes for endorsement during the execution phase. The endorsers are only a subset of the nodes in the blockchain network, and different endorsers can endorse different transactions at the same time, enabling the system to execute transactions in parallel. After collecting a sufficient number of endorsements, the client packages all response into a transaction and send it to orderers for block creation. Finally, the orderers send the blocks to all the nodes for validation and synchronization of ledger states. This model utilizes optimistic concurrency control techniques to ensure the consistency of data. However, it may lead to multi-version concurrency control (MVCC)[10] conflicts during the validation phase.

We categorize conflicts in EOV systems into two types: within-block conflicts and cross-block conflicts. Within-block conflicts occur within the same block, where the write set modifications of transactions alter the version numbers of read sets for later-executed transactions, resulting in the invalidation of the latter transactions during the validation phase. Cross-block conflicts occur when the value read by a transaction during the execution phase is invalidated before reaching the validation phase due to modifications made by the submission of other blocks. SHARMA et al.[11] proposed a system called Fabric++ to address within-block conflicts by reordering transactions. However, our testing showed that Fabric++ is inefficient when the transaction conflict rate is high. To address this issue, we made several optimizations to the EOV blockchain based on Fabric v2.4, naming FabricMan. The main contributions of this paper are as follows:

1) We design a reordering algorithm with stable time complexity to reduce within-block conflicts. Experimental results show that our algorithm performs better under high transaction conflict rates compared with Fabric++.

2) Based on the transaction conflict graph generated during reordering, we perform parallel validation of unrelated transactions in the validation phase to leverage the advantages of multi-core CPUs.

3) At the chaincode level, we analyze transactions and merge simple transfer transactions to maximize the validation pass rate.

4) We implement a cache-based version validation mechanism to detect and terminate invalid transactions during the ordering phase, reducing cross-block conflicts.

The rest of the paper is organized as follows: Section 2 introduces the structures of Fabric and Fabric++, as well as other related research. Section 3 provides a theoretical analysis of the problems in Fabric and proposes our findings. Section 4 describes the design of FabricMan. Section 5 presents experimental tests of FabricMan's optimizations and compares them with Fabric and Fabric++. Finally, Section 6 concludes our work.

2 Background and Related Work

2.1 EOV Architecture in Hyperledger Fabric

One of the representative blockchain platforms based on the EOV architecture is Hyperledger Fabric[12], abbreviated as Fabric. All nodes in Fabric are known and authorized at all times and are mainly divided into three types: 1) Clients are responsible for submitting transaction proposals and collecting endorsement responses; 2) peers are responsible for executing and validating transaction proposals, and then committing their write sets to maintaining local ledgers; 3) orderers are responsible for ordering transactions and packaging them into blocks according to predefined rules. The workflow of a transaction consists of three phases: execution, ordering, and validation.

1) Execution phase

During the execution phase, clients send the transaction proposal to a subset of peers (endorsers), according to a predefined policy. Endorsers simulate the execution of transactions in parallel based on the current ledger state, generating the corresponding read and write sets. The read set consists of (key, ver) tuples, and the write set consists of (key, val) tuples, where key is a unique name representing the entry, and ver and val are the latest version number and value of the entity, respectively. After execution, endorsers return the read and write set with their signatures to the client. When a client collects sufficient responses from different endorsers, it can package them into a transaction and send them to the ordering service to enter the next phase.

2) Ordering phase

During the ordering phase, different orderers continuously receive transactions from different clients. The ordering service needs to achieve two goals: a) reaching a consensus on transaction orders, and b) packaging ordered transactions into blocks according to rules and delivering them to all peers. In Fabric v2.4, the Raft protocol is used for achieving crash-fault-tolerant consensus in a). The block creation rules in b) are generally formed by the maximum block interval and the maximum number of transactions included in a block.

3) Validation phase

When a node receives a block from the orderers, it first checks for the presence of signatures and the legality of the block structure. If the check passes, the block is added to a validation queue to ensure it can be added to the blockchain. Then, it goes through the validating state-based endorsement check (VSCC) and MVCC validation stages. In the VSCC stage, the node checks if each transaction in the block meets the specific endorsement policy of the chaincode; if not, the transaction is marked as invalid but remains in the block. In the MVCC stage, all transactions are sequentially checked for multi-version concurrency control. If the version number of a key in the transaction's read set does not match the version number in the current local state, the transaction is marked as invalid. Finally, the node writes the block into its local ledger and modifies the ledger state according to the validity of each transaction.

2.2 Optimization of Fabric++

The vanilla Fabric sorts transactions based on the order in which they arrive at the orderers. While this approach allows for quick ordering, it may lead to unnecessary serialization conflicts. To address the aforementioned issue, SHARMA et al.[11] introduced a reordering algorithm in the ordering phase of Fabric. This algorithm terminates a small number of transactions based on the relationship between the read and write sets of transactions. Subsequently, it constructs a conflict-free ordering for the remaining transactions, thereby increasing the success rate of transactions within a block. The algorithm consists of five main steps as follows. 1) A conflict graph is built based on the read and write sets of all transactions to be sorted. 2) Tarjan's algorithm[13] is used to identify all strongly connected subgraphs and Johnson's algorithm[14] to identify all cycles within these subgraphs. 3) The cycles each transaction is part of are idenified, and the times of each transaction appearing in the cycles are counted. 4) The transactions that appear in the most cycles are sequentially terminated until the conflict graph has no cycles. 5) Finally, a serializable scheduling scheme is established using the remaining transactions.

2.3 Related Work

Currently, optimizations for the EOV blockchain can be broadly categorized into two types:

1) Improving the overall throughput of the system

THAKKAR et al.[15] conducted comprehensive tests on the performance of Fabric v1.0 by configuring parameters such as the block size (BS), endorsement policy, channel, resource allocation, and ledger database. They identified three main performance bottlenecks: endorsement policy validation, validation of the order of transactions in a block, and validation and submission of states in CouchDB. They proposed simple optimizations to address the following issues: a) using a hash map with serialized identities as keys to cache deserialized identities, reducing resource consumption for encryption operations; b) parallel validation of endorsements for multiple transactions to utilize idle CPU resources and improve overall performance; c) batch read and write optimization for CouchDB. These optimizations effectively increase the overall throughput of the system. GORENFLO et al.[16] reengineered Hyperledger Fabric v1.2 by a) passing only transaction IDs instead of entire transactions during ordering, b) actively caching unassembled blocks in committers, parallelizing as many verification steps as possible, c) redesigning the data management layer using an in-memory database instead of the original data storage, and d) separating roles responsible for endorsement and submission. These changes reduce computational and I/O overhead during transaction ordering and validation, increasing transaction throughput from 3 000 transactions per second (TPS) to 20 000 TPS.

2) Reducing read/write conflicts caused by parallel execution

RUAN et al.[17] studied the Fabric++ solution and found that it did not consider dependencies between transactions across blocks, limiting the effectiveness of reordering. They proposed a reordering algorithm based on a more granular concurrency control strategy and verified its safety, resulting in improved reordering effectiveness. SUN et al.[18] analyzed the reordering algorithm implemented in Fabric++ and found issues regarding trust. They proposed a trusted reordering algorithm grounded in a greedy approach.

3 Problem Analysis

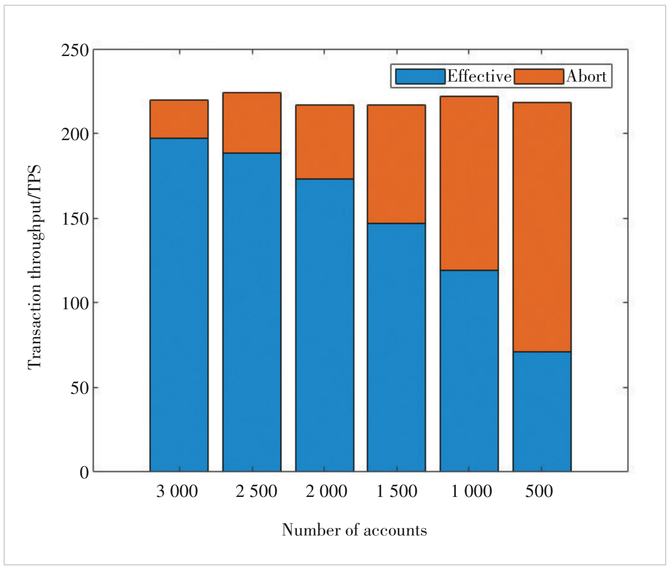
As mentioned in Section 2, Fabric generates read and write sets for transactions while execution. During the validation phase, nodes perform MVCC validation on the read sets based on the current state of the local database. If the versions do not match, the transaction is marked as invalid, and its write set cannot be used to update the ledger state, resulting in an MVCC conflict. To assess the impact of these conflicts on the system, we conducted tests on Fabric using the SmallBank smart contract under the configuration described in Section 5, as shown in Fig. 1.

In our experiments, each block contains 256 transactions. When the total number of accounts is 3 000, the conflict rate is relatively low, resulting in a high TPS for successful transactions, accounting for approximately 90%. However, as the number of accounts decreases, the conflict rate within blocks increases, resulting in a higher rate of transaction abortions. When the total number of accounts is 500, the TPS for successful transactions drops to only 30%. This demonstrates a significant performance decrease when the number of transaction conflicts within blocks increases.

In high-concurrency execution environments, we classify MVCC conflicts in the EOV blockchain into within-block and cross-block conflicts.

3.1 Within-Block Conflicts

Within-block conflicts occur when there are conflicts between different transactions within the same block. When mul-



▲**Figure 1. Transaction throughput of Fabric** tiple transactions that read or write the same key are grouped into the same block, it may lead to this type of conflict. In the example provided in Table 1, transactions T_1 and T_2 are sequentially packaged into a single block. During the validation phase, T_1 updates key k_1, changing its version number to v_1. Next, T_2 is validated, and its read set includes key k_1 with version v_0. During the MVCC validation, it is discovered that $v_0 \neq v_1$, resulting in T_2 being marked as invalid.

Observation 1: It is possible to reduce the number of conflicts within a block by modifying the validation order of transactions. The fundamental reason for within-block conflicts is that two different transactions perform a write-followed-by-read operation on the same key. By adjusting the order of transactions, we can ensure that they perform read operations before write operations. In the given example, if T_2 is validated before T_1, there would be no conflict.

As mentioned in Section 2.2, Fabric++ uses Johnson's algorithm during reordering, with a time complexity of $O((n + e)(c + 1))$, where n is the number of nodes, e is the number of edges, and c is the number of cycles in the graph. While the number of nodes and edges in the conflict graph can be controlled to small values, the number of cycles may be very large. Ref. [19] highlighted a similar issue: when resolving cycles in Fabric++, recalculating the occurrence count of individual transactions in a cycle results in a time complexity of $O(n^3)$ for the entire algorithm. Considering that topological sorting is an algorithm for ordering graph vertices with a stable time complexity of $O(n + e)$, we can propose a new reordering algorithm based on it. This algorithm can rapidly complete reordering even when transaction conflict rates are high.

Observation 2: The conflict graph generated by reordering can reflect dependency information between transactions, which can be utilized for parallel validation. In the Fabric, validation can be divided into two main stages: VSCC and MVCC. VSCC is used to evaluate whether endorsements in transactions comply with the endorsement policy, and this step is already parallelized in the system. MVCC, on the other hand, is executed sequentially. Ref. [15] pointed out that one of the performance bottlenecks of Fabric is the serial MVCC validation of all transactions within a block. If we parallelize the validation of unrelated transactions by leveraging transaction dependency relationships, we can fully harness the advantages of multi-core CPUs to enhance system performance.

Observation 3: Transfers are one of the primary transaction types, characterized by simple logic and fixed parameters, rendering them suitable for merging. As a permissioned blockchain, obtaining block data from Fabric is challenging. Ref. [20] analyzed transactions on Ethereum over a period of time and found that the main types of transactions leading to conflicts are ERC20 token transactions accounting for 60%, decentralized finance (DeFi) transactions accounting for 29%, and gaming transactions accounting for 10%. Therefore, we believe that the merging of transfer transactions holds significance.

▼**Table 1. An example of within-block conflict**

Order	Transaction	Read Set	Write Set	Validity
1	T_1	-	$(k_1, v_0 \rightarrow v_1)$	Valid
2	T_2	$(k_1, v_0), (k_2, v_0)$	$(k_2, v_0 \rightarrow v_1)$	Invalid

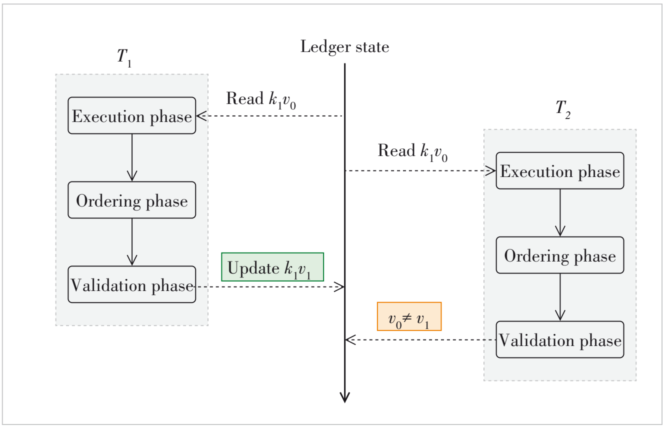
3.2 Cross-Block Conflicts

Due to the nature of the EOV structure, there is a certain delay between the execution and verification. If a transaction in a later block reads a key that was written by a transaction in an earlier block before the earlier block's verification, it can result in a dirty read in the later block, leading to a conflict. As shown in Fig. 2, T_1 and T_2 are two transactions in different blocks. During the execution phase, T_1 reads the current version number v_0 of key k_1. From the verification phase, it can be seen that T_1 modifies key k_1 in its write set, but since this step is a simulated execution, the database state is not altered. Therefore, T_2 still reads version v_0 of k_1. Subsequently, the block containing T_1 enters the verification phase and updates the version number of key k_1 to v_1, resulting in T_2 invalidated. Additionally, under special circumstances such as network congestion, cross-block conflicts can also occur.

Observation 4: Orderers have the opportunity to early abort invalid transactions caused by cross-block conflicts. All transactions arrive at the orderer for block generation. Since the version numbers of keys in the read sets are obtained from the ledger during execution, the version of a key in the ledger at this point must be no lower than the version in the read set. We can utilize a caching mechanism to store versions of keys, thereby filtering out invalid transactions.

4 Design of FabricMan

In the previous section, we have analyzed two types of conflicts in the EOV blockchain and identified four directions for optimization. In this section, we will first introduce our modifications to the ordering phase and then discuss the four modular designs for each direction: transaction reordering, parallel verification, transaction merging, and caching mechanism.



▲**Figure 2. An example of cross-block conflict**

Our optimization efforts primarily focus on the ordering phase. The orderer receives transactions from multiple clients and merges them into a batch until the conditions for block generation are fulfilled. The conditions consist of two parts: When the number of transactions in the batch reaches a predefined threshold, and then when the time taken to construct the batch reaches the maximum block generation time limit.

Once either condition is met, our system starts processing the batch. All transactions are first filtered through a version cache maintained by the orderers. During this process, the system extracts the read sets of transactions and compares them with the cache. Transactions that do not meet the filtering criteria are aborted and feedbacks are provided to the clients. Transactions are then checked to determine if they are transfer transactions. If so, they are moved to the transfer array and merged with other transfer transactions. The remaining transactions in the batch are non-transfer transactions that undergo reordering and subdivision into subgraphs based on dependency relationships. Finally, the system constructs a new block by incorporating merger transactions, transfer transactions, and the reordered batch. It then adds the transaction subgraphs to the block header and distributes the block to all peers.

4.1 Transaction Reordering

When reordering a transaction set S, it is necessary to identify the dependencies between the transactions to construct a transaction conflict graph. In the graph, if a transaction T_i points to another transaction T_j, it means that the write set of T_i intersects with the read set of T_j, denoted as $T_i \rightarrow T_j$. This indicates that during block ordering, T_j needs to precede T_i, otherwise T_j will be invalidated due to reading outdated versions. When constructing the conflict graph, each node represents a transaction. According to the aforementioned definition, the out-degree of a node indicates the number of other transactions whose validity it affects, while the in-degree indicates the number of other transactions that affect it. Additionally, an important issue to address is the presence of cycles in the graph. Circular graphs cannot be serialized, thus necessitating the use of algorithms to remove certain transactions and convert the original conflict graph into an acyclic graph.

Our algorithm is based on topological sorting, but due to the unsuitability of topological sorting for cyclic graphs, we make some improvements. From a high-level perspective, it mainly consists of five steps as follows. 1) The conflict graph is constructed based on the read-write sets of each transaction in S, and the in-degree and out-degree of each node are recorded. 2) A node n with the minimum in-degree is selected for processing. If multiple nodes meet this criterion, the one with the maximum out-degree is prioritized. If there are still multiple options, the one with the smallest index is chosen. 3) Other nodes pointing to n from the graph are removed until the in-degree of n is 0. 4) N is added to the result queue and removed from the graph. 5) Steps 2), 3), and 4) are repeated until there are no remaining nodes in the conflict graph. Finally, the result queue is reversed to obtain a conflict-free serialized ordering. The pseudo-code of Algorithm 1 implements these five steps.

Note that in Step 2), since we need to reduce the in-degree of n to 0, we prioritize selecting the node with the minimum in-degree to retain more transactions. Furthermore, since the out-degree of a node indicates the number of transactions it will affect after ordering, and the final ordering is the reverse of the ordering queue, i.e., transactions that enter the ordering queue first will be placed at later positions in the algorithm, we prioritize selecting transactions with the maximum out-degree for ordering.

Algorithm 1. Reordering algorithm

```
1.  func ReorderSort(Transaction[ ] S) {
2.      // Step 1: Construct a transaction conflict graph and an exit and entry table
3.      Graph cg = buildConflictGraph(S)
4.      Graph incg = invert cg
5.      map[Transaction]int indegree = Calculate in-degrees using cg
6.      map[Transaction]int outdegree = Calculate out-degrees using cg
7.      // Step 2: Select nodes to be sorted
8.      while S is not empty:
9.          for each Transaction tx in S:
10.             if indegree[tx] < minIndegree:
11.                 min = indegree[tx]
12.                 nodeToSort = node
13.             else if indegree[tx] == min and outdegree[tx] > outdegree[nodeToSort]:
14.                 nodeToSort = node
15.         // Step 3: Process nodeToSort so that their degree is 0
16.         for each nodeToRemove in incg[nodeToSort]:
17.             if nodeToRemove not in S:
18.                 continue
19.             remove nodeToRemove from S
20.             for each tx in cg[nodeToRemove]:
21.                 indegree[tx]--
22.             for each tx in incg[nodeToRemove]:
23.                 outdegree[tx]--
24.         // Step 4: Add nodeToSort to the queue and remove from transaction graph
25.         append nodeToSort to result
26.         for each tx in cg[nodeToSort]:
27.             indegree[tx]--
28.         remove nodeToSort from S
29.     // Step 5: Return the reverse order of the ordering queue to obtain the ordering result
30.     return result.invert()
31. }
```

Here is an example to better understand the algorithm. We assume there are six transactions, T_0 to T_5, within a set S, and their read-write sets are as shown in Table 2. These six transactions access 10 different keys, k_0 to k_9, and 0 indicates that a transaction's read or write set does not contain a certain key, while 1 indicates that it does. The reordering process is as follows:

1) The first step is constructing a conflict graph for transactions in S based on the read-write sets, as shown in Fig. 3. If T_i points to T_j, it indicates that the write set of T_i shares common keys with the read set of T_j. At this point, the in-degree and out-degree of each node are shown in Table 3.

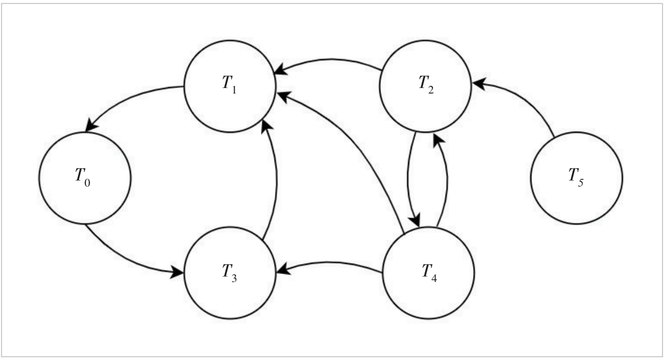
2) In the first iteration, following step 2, the system selects T_5 as n because it currently has the smallest in-degree among the transactions in S. As the in-degree of T_5 is 0, Step 3 is skipped. Next, the system adds T_5 to the result, removes it from S, and then decrements the in-degree of the node pointed to by T_5, which is T_2.

3) In the second iteration, since T_0, T_2 and T_4 all have an in-degree of 1, making them the nodes with the lowest in-degree, the system selects T_4 as n based on Step 2. Then, in Step 3, the system removes the node T_2 from S, reducing the in-degree of T_4 to 0. In Step 4, T_4 is added to the result and removed from S, and then the in-degrees of T_1 and T_3 are each decreased by one.

4) At this point, there remains a cycle consisting of T_0, T_1 and T_3 in the graph. Since their in-degree and out-degree are the same, the system chooses T_0 with the smallest index as n. It removes T_1, adds T_0 to the result, and then adds T_3 to the result. Reversing the result queue, we can get the final sorted result: $T_3 - T_0 - T_4 - T_5$.

The central part of this reordering algorithm exhibits a time complexity comparable to that of topological sorting, which is $O(n + e)$. Moreover, it is not affected by the number of cycles in the graph, eliminating this flaw presented in the Fabric++. It is worth noting that our algorithm does not guarantee the termination of the minimum number of transactions to make the graph acyclic, as this is an NP-hard problem. We merely provide a very lightweight way to terminate a small number of transactions, thereby generating a serializable ordering scheme.

Another point that needs to be clarified is that the reordering algorithm proposed by RUAN et al.[17] resembles a greedy strategy. The system constructs a conflict graph for all pending transactions. When a new transaction is received, its dependencies are added to the conflict graph. If a cycle is detected, the new transaction is directly dropped, ensuring that transactions in the pending queue do not have circular dependencies. In contrast, Fabric++ and our algorithm are more modular. We construct a conflict graph for all the transactions within a block and then eliminate cycles, ultimately achieving a serializable ordering.



▲ **Figure 3. Initial conflict graph formed by the six transactions in a block**

▼**Table 3. Initial in-degree and out-degree of the six transactions**

Transaction	In-Degree	Out-Degree
T_0	1	1
T_1	3	1
T_2	2	2
T_3	2	1
T_4	1	3
T_5	0	1

▼**Table 2. Read and write sets of the six transactions**

Read Set										
T_x	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9
T_0	1	0	0	0	0	0	0	0	0	0
T_1	0	0	0	1	1	1	0	0	0	0
T_2	0	0	0	0	0	0	1	1	0	0
T_3	0	1	1	0	0	0	0	0	1	0
T_4	0	0	0	0	0	0	0	0	0	1
T_5	0	0	0	0	0	0	0	0	0	0
Wirte Set										
T_x	k_0	k_1	k_2	k_3	k_4	k_5	k_6	k_7	k_8	k_9
T_0	0	0	1	0	0	0	0	0	0	0
T_1	1	0	0	0	0	0	0	0	0	0
T_2	0	0	0	1	0	0	0	0	0	1
T_3	0	1	0	0	1	0	0	0	0	0
T_4	0	0	0	0	0	1	1	0	1	0
T_5	0	0	0	0	0	0	0	1	0	0

4.2 Parallel Verification

When conducting reordering, the algorithm generates a conflict graph among transactions, as indicated by lines 3 and 4 in Algorithm 1. Here, cg represents the conflict graph, depicted by a two-dimensional array. Each element cg[i] is a one-dimensional array, where the presence of an element j indicates that transaction T_j must occur before transaction T_i, namely $T_i \rightarrow T_j$.

After reordering, the aborted transactions are removed from the conflict graph. The system then performs a depth-first search (DFS) operation on cg to identify their connected components and further partition them into mutually disconnected subgraphs. When the block is generated, the information regarding these subgraphs will be serialized and appended to the block header. During the validation phase, nodes deserialize the information and utilize goroutines to perform MVCC validations on independent subgraphs in parallel.

4.3 Transaction Merging

During the execution phase, we analyze the parameters of chaincode transactions, which allows us to identify simple transfer transactions. Additionally, we include a value field in the structure of the transaction's read set to represent its initial value.

In the ordering phase, nodes identify transfer transactions under different chaincodes and construct a transfer table for each chaincode. The node adds the initial balance of each account to the Moneymap table and their corresponding version numbers to the Versionmap table. If the sender's balance is less than the transfer amount, the transaction is aborted. Otherwise, the sender's balance is decreased by the transfer amount, and the receiver's balance is increased by the same amount. After processing all transfer transactions, the system utilizes the keys and corresponding version numbers in the Versionmap to form the read set of the merger transaction. Similarly, it utilizes the keys and values in the Moneymap to create the write set of the merger transaction.

Once the merger transaction is constructed, it is positioned at the start of the block during formation, followed by all the merged transfer transactions. Retaining the merged transactions in the block serves two purposes: First, it enables the system to offer feedback to clients concerning the success or failure of the merger transactions during validation; second, it maintains transaction data on the blockchain for future auditing purposes.

4.4 Caching Mechanism

To mitigate the cross-block conflicts mentioned in Section 3.2, we deploy a cache utilizing a hash table at the orderers. This cache is employed to store the keys and version numbers extracted from the read and write sets of received transactions. The cache table consists of three fields: key, version, and updated flag. The key represents the unique entity name in the smart contract chaincode, while the version indicates the latest version number read for the corresponding key in the transaction. The updated flag denotes whether the key may have been updated by a new block.

Upon the arrival of transactions at the orderer, the read sets of each transaction are checked against the cache. If any key is found to have a version lower than that in the cache, the transaction is immediately aborted. This prevents outdated transactions from occupying system resources due to network delays. If a key has a version greater than or not present in the cache, the cache is updated with the latest version. If a key matches the version in the cache, the system checks if it has the updated flag; if so, the transaction is aborted.

After the completion of reordering, the orderer obtains a block containing transactions with no conflicts. Under normal operation, the write sets of this block are applied to update the ledger. At this point, the orderer checks the read sets of each transaction. If the cache contains keys identical to those in the read set, the corresponding values in the cache are marked with the updated flag, indicating that the version is no longer the latest for that key.

However, there is a potential issue with this cache mechanism. If transaction T_1 fails to pass the final validation phase for some unexpected reason (e.g., a signature issue), the version of k_1 in the node's ledger remains v_0. However, in the orderer, the version of k_1 has already been altered to $v_0 - \mathit{updated}$, causing subsequent transactions reading $k_1: v_0$ to fail. To address this issue, we introduce a timer in the cache. If a key is not updated within two block intervals, it is removed from the cache. This also ensures that the cache does not indefinitely increase in size.

5 Experimental Evaluation

To validate the improvements proposed in this paper, we conducted experimental evaluations of key metrics such as throughput, transaction abort rate, and algorithm execution time for Fabric, Fabric++, and FabricMan. Since the original Fabric++ code is implemented on Fabric v1.2 while FabricMan is based on Fabric v2.4, for comparison purposes, we also reimplemented Fabric++ based on Fabric v2.4.

5.1 Setup and Workload

The experimental setup involves a single-channel blockchain system comprising two organizations, each containing two peer nodes deployed via docker containers. The consensus mechanism used is Raft, and LevelDB serves as the state database. The experiments were conducted on a server with a 36-core CPU (Intel Core i9-10980XE 3.0 GHz), 256 GB RAM, running on Ubuntu 20.04.5 LTS.

Two types of workloads were employed in the experiments: Smallbank and custom chaincode, assessed through the caliper-benchmarks framework. The Smallbank contract creates a checking account and a savings account for each user and includes six functions. In the custom chaincode, we defined complex read-write transactions that read the balances of four accounts and modify two of them.

In Section 3, Smallbank is used to test Fabric for measuring the transaction throughput under different numbers of accounts. The transaction conflict rates corresponding to different numbers of accounts are shown in Table 4. In subsequent experiments, we continued to use these account numbers to measure the system's performance under different transaction conflict rates.

5.2 Impact of Block Size

BS is one of the important factors affecting the throughput

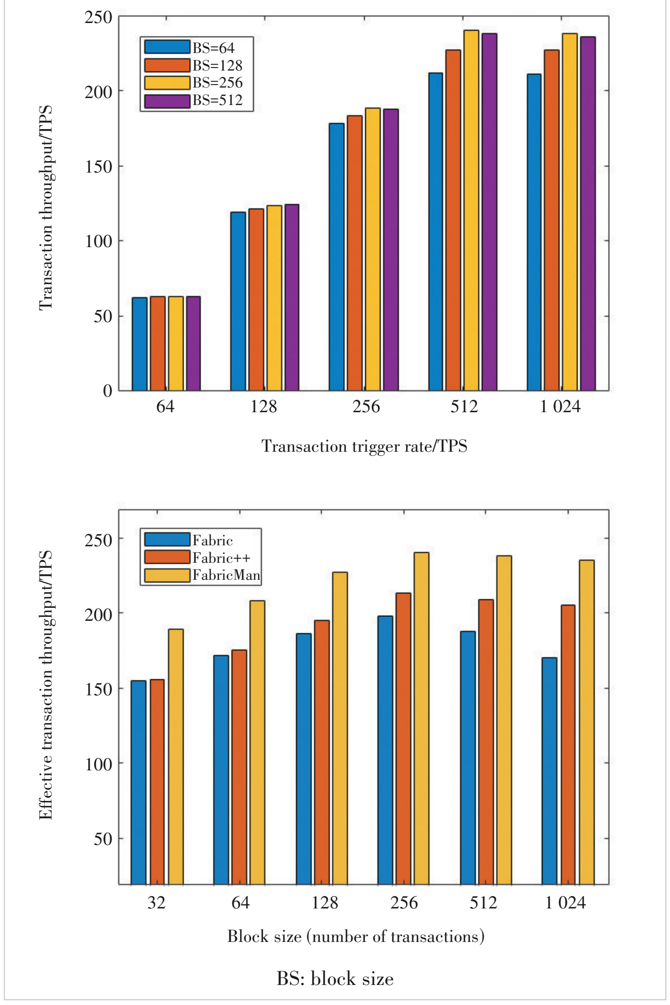
▼Table 4. Number of accounts and corresponding conflict rates

Number of Accounts	3 000	2 500	2 000	1 500	1 000	500
Conflict rate/%	10.5	14	20.3	32.3	46.4	67.8

and latency of a blockchain. We tested the impact of the transaction trigger rate on FabricMan, and the impact of changing the block size on the throughput of Fabric, Fabric++, and FabricMan. We used the Smallbank contract and tested the performance of the systems in a low-conflict environment when the number of accounts was set to 3 000. The results are shown in Fig. 4.

It is found that as the transaction trigger rate increases, the throughput of FabricMan also gradually increases, reaching saturation at around 240 TPS when the transaction trigger rate is 512 TPS. As the block size increases from 64 to 256, the system throughput also increases, but it decreases when the block size is set to 512. This is because increasing the number of transactions in a block leads to more conflicting transactions, reducing the throughput of successful transactions, and larger blocks also increase the transmission time in the system.

We can also observe that as the number of transactions in a block increases, the difference in throughput of successful transactions between FabricMan and Fabric++ compared to Fabric becomes larger. This is because as the number of transactions in the block increases, the probability of conflicts also increases, making the effect of reordering more pronounced. The throughput of all three systems reaches its peak when the block size is set to 256. Therefore, in subsequent experiments, we use 256 as the number of transactions included in a block, and set the maximum block interval to one second, and the transaction trigger rate to 512 TPS.



▲ **Figure 4. Impact of transaction trigger rate of FabricMan (up) and that of block size of three systems (down)**

5.3 Comparison of Reordering Algorithms

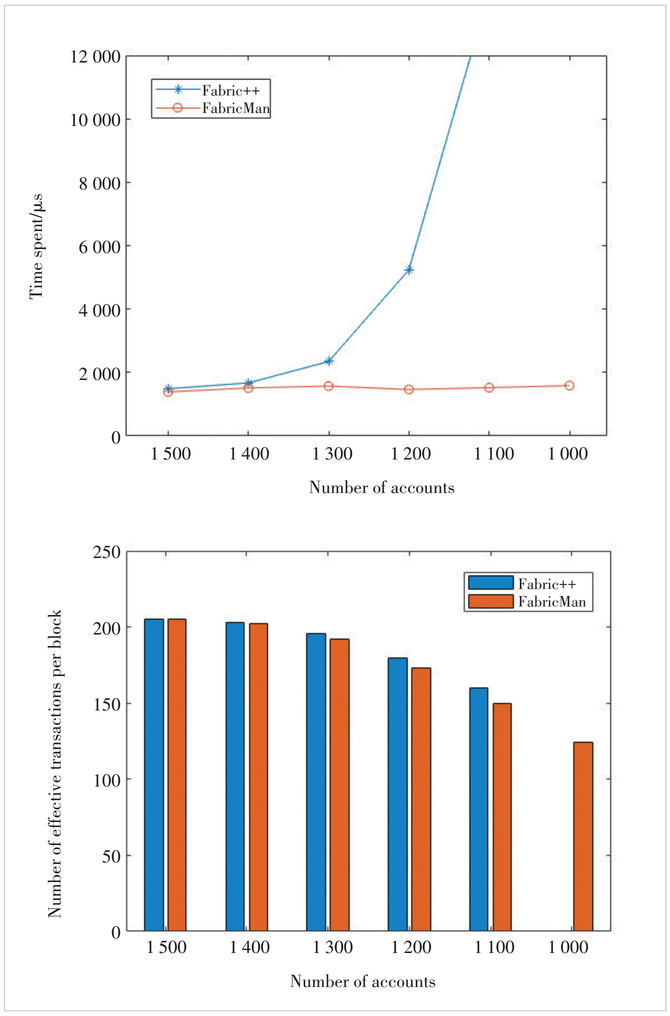
To evaluate the performance of the reordering algorithms, we prepared multiple pre-packaged blocks (each containing 256 transactions) and applied the reordering algorithms of FabricMan and Fabric++ separately. Their execution time, the number of valid transactions in each block, and the throughput of valid transactions were compared. In this experiment, we used complex read-write transactions from custom chaincode to increase the conflict rate between transactions, aiming to better evaluate the performance of both algorithms.

We controlled the number of accounts to gradually decrease from 1 500 to 1 000. Fig. 5 shows the time taken by both the algorithms and the number of valid transactions within each block. It can be observed that the time required by Fabric++ for reordering increases significantly as the conflict rate increases, and it becomes unable to generate blocks when the number of accounts reaches 1 000. In contrast, the algorithm of FabricMan remains stable at around 1 500 us. However, since FabricMan's reordering algorithm does not select nodes with the most cycles in each round like Fabric++, the final number of successful transactions is slightly lower than that in Fabric++. Nevertheless, its stable time complexity allows it to generate blocks normally even in high-conflict environments with 1 000 or fewer accounts.

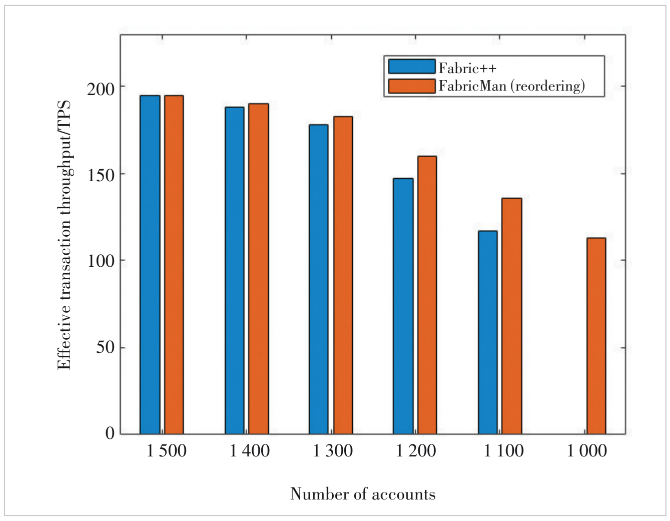
The comparison of throughput between the two systems is shown in Fig. 6, where FabricMan only uses the optimization of the reordering. It can be seen that FabricMan consistently outperforms Fabric++ in throughput across different settings of the number of accounts. This is because, in the range of 1 500 to 1 400 accounts, where the conflict rate is relatively low, both reordering algorithms yield a comparable number of valid transactions within the same block. However, FabricMan's reordering algorithm has shorter execution times. As the number of accounts decreases below 1 400, the conflict rate increases significantly. While Fabric++'s algorithm can produce more valid transactions, the time it takes for execution increases significantly.

5.4 Effect of Parallel Verification

We also tested the impact of assigning different numbers of CPU cores to FabricMan on parallel verification time. The extensive use of CPU resources for identity encryption and decryption operations in permissioned blockchains could affect our experimental results. We used multiple pre-packaged blocks and conducted modular tests on MVCC verification time with validation nodes having CPU core counts ranging from 1 to 16.



▲Figure 5. Comparison of two algorithms in execution time and effective transaction quantity



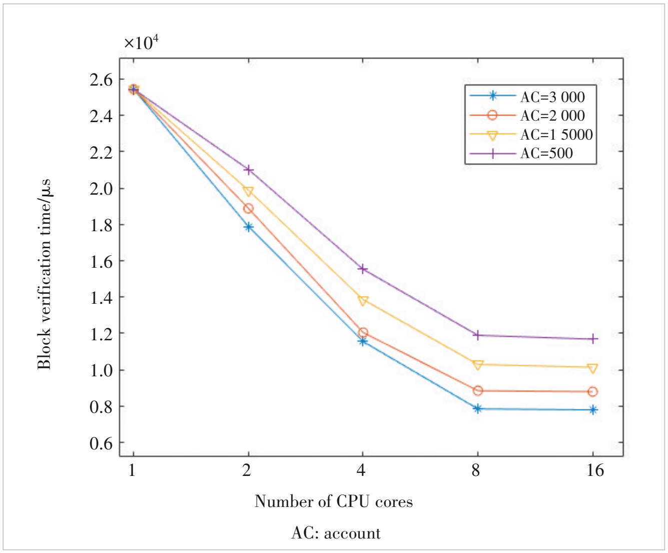
▲Figure 6. Throughput of two systems under different account numbers

The experiments employed the Smallbank contract and varied the number of accounts (AC) to represent different conflict rates. To ensure verification of the same number of transactions at different conflict rates, we only used the subgraph partitioning algorithm instead of reordering during block generation. The test results, as depicted in Fig. 7, indicate a significant reduction in verification time with an increase in the number of cores used. However, beyond 8 cores, the reduction in time becomes less pronounced, as this stage becomes bottlenecked by interactions with the database and the serial verification of the longest transaction conflict chain. Moreover, as the block conflict rate increases, resulting in longer conflict chains, the corresponding verification time also increases.

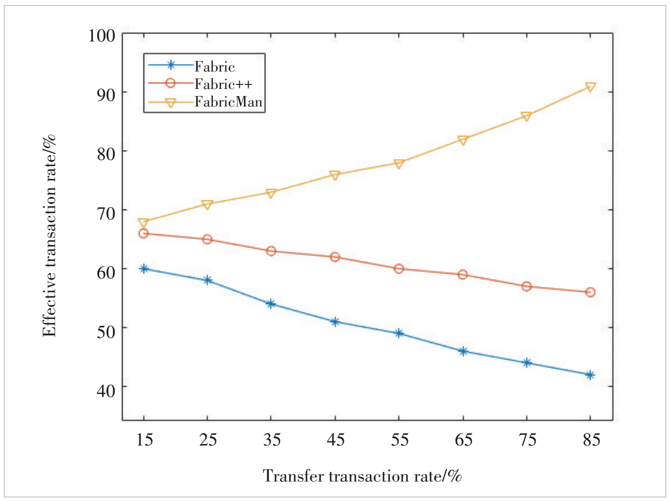
5.5 Effect of Transaction Merging

In this section, we used 1 000 accounts to send non-transfer or transfer transactions in the Smallbank contract. Non-transfer transactions cannot be merged, while transfer transactions can. We gradually increased the proportion of transfer transactions from 15% to 85%. The proportion of successful transactions in the Fabric, Fabric++, and FabricMan systems is shown in Fig. 8.

When the proportion of transfer transactions is low, Fabric++ and FabricMan have a higher successful transaction rate due to reordering. However, as the proportion of transfer transactions increases, the successful transaction rate in FabricMan increases because of the transaction merging mechanism. When the proportion of transfers reaches 85%, over 90% of transactions in FabricMan can be successfully submitted. In contrast, as the read-write set of transfers in Smallbank is more complex than others, the number of MVCC conflicts in Fabric and Fabric++ increases, leading to a decrease in the successful transactions rate.



▲Figure 7. Block verification time under different numbers of CPU cores



▲**Figure 8. Effective transaction rates within the three systems with different transfer transaction rates**

5.6 Combinations of Optimizations

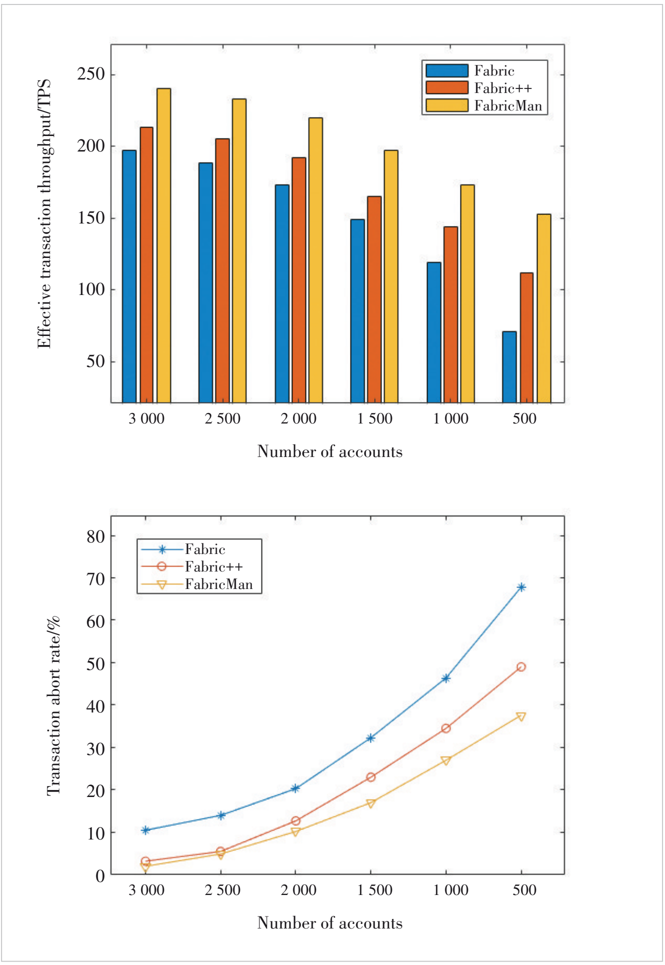
Finally, we conducted comprehensive performance testing of the systems. We used the Smallbank contract as the workload and applied all optimizations mentioned in Section 4 to FabricMan. We tested the system under different numbers of accounts and compared the throughput and transaction abort rate with Fabric and Fabric++, as shown in Fig. 9.

The results demonstrate that when the number of accounts is 3 000 and the transaction abort rate is low, Fabric exhibits the lowest throughput among the three systems, at less than 200 TPS. Fabric++, benefiting from reordering, achieves a slightly higher throughput of around 215 TPS. In contrast, FabricMan, leveraging both reordering and parallel validation along with the merging of transfer transactions, achieves the highest throughput of approximately 240 TPS.

As the number of accounts gradually decreases from 3 000 to 500, resulting in an increasing transaction abort rate, Fabric experiences a significant decline in effective transaction throughput, dropping to less than half of its initial level. Meanwhile, FabricMan experiences a slower decrease in effective transaction throughput, with the lowest transaction abort rate. Even in high-concurrency conflict environments, FabricMan can maintain relatively high throughput.

6 Conclusions and Future Work

We mitigate the performance impact of MVCC conflicts arising from concurrent execution in the innovative EOV blockchain by introducing a comprehensive blockchain architecture named FabricMan. This architecture addresses both within-block and cross-block conflicts while enabling parallel validation and transaction merging. Through testing, FabricMan has demonstrated superior performance in terms of throughput, transaction abort rate, and execution time compared to the baseline schemes. However, there are several areas for future improvement in our work. First, the reordering



▲ **Figure 9. Comparison of throughput and transaction abortion rates of three systems**

of transactions within a block may potentially compromise the fairness of the system. In future work, we plan to analyze this issue and introduce appropriate parameters into the algorithm to address it. Second, our experiments were conducted using Docker on a single server, which could not effectively simulate factors such as communication latency present in real networks. The problem of cross-block conflicts was not adequately addressed, so it was not discussed in the experimental phase. In future work, deploying the system in a multi-node environment can provide a better understanding of this issue and facilitate further discussion.

References

[1] NAKAMOTO S. Bitcoin: a peer-to-peer electronic cash system [EB/OL]. (2008-10-31)[2024-03-15]. https://nakamotoinstitute.org/library/bitcoin
[2] BUTERIN V. A next generation smart contract & decentralized application platform [R]. Ethereum white paper, 2014
[3] QIN K H, ZHOU L Y, GERVAIS A. Quantifying blockchain extractable value: how dark is the forest? [C]//Proc. IEEE Symposium on Security and Privacy (SP). IEEE, 2022: 198 – 214. DOI: 10.1109/SP46214.2022.9833734

[4] AZARIA A, EKBLAW A, VIEIRA T, et al. MedRec: using blockchain for medical data access and permission management [C]//Proc. 2nd International Conference on Open and Big Data (OBD). IEEE, 2016: 25 – 30. DOI: 10.1109/OBD.2016.11

[5] FAN K, WANG S Y, REN Y H, et al. MedBlock: efficient and secure medical data sharing via blockchain [J]. Journal of medical systems, 2018, 42(8): 136. DOI: 10.1007/s10916-018-0993-7

[6] ABEYRATNE S A, MONFARED R P. Blockchain ready manufacturing supply chain using distributed ledger [J]. International journal of research in engineering and technology, 2016, 5(9): 1 – 10. DOI: 10.15623/IJRET.2016.0509001

[7] DAI H N, ZHENG Z B, ZHANG Y. Blockchain for internet of things: a survey [J]. IEEE internet of things journal, 2019, 6(5): 8076 – 8094. DOI: 10.1109/JIOT.2019.2920987

[8] VUKOLIĆ M. The quest for scalable blockchain fabric: proof-of-work vs. BFT replication [M]//Open problems in network security. Cham: Springer International Publishing, 2016: 112 – 125. DOI: 10.1007/978-3-319-39028-4_9

[9] CHARRON-BOST B, PEDONE F, SCHIPER A. Replication: theory and practice [M]. Berlin Heidelberg: Springer, 2010

[10] PAPADIMITRIOU C H, KANELLAKIS P C. On concurrency control by multiple versions [J]. ACM transactions on database systems, 9(1): 89 – 99. DOI: 10.1145/348.318588

[11] SHARMA A, SCHUHKNECHT F M, AGRAWAL D, et al. Blurring the lines between blockchains and database systems: the case of hyperledger fabric [C]//Proc. 2019 International Conference on Management of Data. ACM, 2019: 105 – 122. DOI: 10.1145/3299869.3319883

[12] ANDROULAKI E, BARGER A, BORTNIKOV V, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains [C]//Proc. Thirteenth EuroSys Conference. ACM, 2018: 1 – 15. DOI: 10.1145/3190508.3190538

[13] TARJAN R. Depth-first search and linear graph algorithms [C]//Proc. 12th Annual Symposium on Switching and Automata Theory. IEEE, 1971: 114 – 121. DOI: 10.1109/SWAT.1971.10

[14] JOHNSON D B. Finding all the elementary circuits of a directed graph [J]. SIAM journal on computing, 1975, 4(1): 77 – 84. DOI: 10.1137/0204007

[15] THAKKAR P, NATHAN S, VISWANATHAN B. Performance benchmarking and optimizing hyperledger fabric blockchain platform [C]//Proc IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2018: 264 – 276. DOI: 10.1109/MASCOTS.2018.00034

[16] GORENFLO C, LEE S, GOLAB L, et al. FastFabric: scaling hyperledger fabric to 20000 transactions per second [J]. International journal of network management, 2020, 30(5): e2099. DOI: 10.1002/nem.2099

[17] RUAN P C, LOGHIN D, TA Q T, et al. A transactional perspective on execute-order-validate blockchains [C]//Proc. 2020 ACM SIGMOD International Conference on Management of Data. ACM, 2020: 543 – 557. DOI: 10.1145/3318464.3389693

[18] SUN Q C, YUAN Y Y, GUO T, et al. A trusted solution to hyperledger fabric reordering problem [C]//Proc. 8th International Conference on Dependable Systems and Their Applications (DSA). IEEE, 2021: 202 – 207. DOI: 10.1109/DSA52907.2021.00031

[19] WU H B, LIU H, LI J. FabricETP: a high-throughput blockchain optimization solution for resolving concurrent conflicting transactions [J]. Peer-to-peer networking and applications, 2023, 16(2): 858 – 875. DOI: 10.1007/s12083-022-01401-9

[20] GARAMVOLGYI P, LIU Y X, ZHOU D, et al. Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts [C]//Proc. IEEE/ACM 44th International Conference on Software Engineering (ICSE). IEEE, 2022: 2315 – 2326. DOI: 10.1145/3510003.3510086

Biographies

MA Qianli (maqianli@foxmail.com) received his BE degree in software engineering from University of Electronic Science and Technology of China in 2020. He is currently working toward his ME degree in electronic information engineering from Shenzhen University, China. His research focuses on blockchain.

ZHANG Shengli received his BE degree in electronic engineering and ME degree in communication and information engineering from University of Science and Technology of China in 2002 and 2005, respectively, and PhD degree with the Department of Information Engineering, The Chinese University of Hong Kong, China in 2008. After that, he joined Communication Engineering Department, Shenzhen University, China, where he is currently a full professor. He has authored or coauthored more than 20 IEEE top journal papers and ACM top conference papers, including *IEEE Journal on Selected Areas in Communications*, *IEEE Transactions on Wireless Communications*, *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Communications*, and *ACM Mobicom*. His research interests include blockchain, physical layer network coding, and wireless networks.

WANG Taotao received his PhD degree in information engineering from The Chinese University of Hong Kong (CUHK), China in 2015, MS degree in information and signal processing from Beijing University of Posts and Telecommunications, China in 2011, and BS degree in electrical engineering from University of Electronic Science and Technology of China in 2008. He joined the College of Information Engineering, Shenzhen University, China, as a tenure-track assistant professor in 2016 and was promoted as a tenured associate professor in 2021.

YANG Qing received his BE degree (Hons.) from Huazhong University of Science and Technology, China and PhD degree from The Chinese University of Hong Kong, China. In 2018, he joined as an assistant professor at the College of Electronics and Information Engineering, Shenzhen University, China and the Principal Researcher at the Blockchain Technology Research Center, Shenzhen University.

WANG Jigang received his PhD degree in computer science from Harbin Engineering University, China in 2007. From May 2007 to June 2009, he held a postdoctoral position in Institute of Computer Science, Tsinghua University. From August 2009, Dr. WANG has been with Cyber Security Product Line, ZTE Corporation as general manager. His recent research interests include operating systems, network and information security, and artificial intelligence.