Key-Based Transaction Reordering: An Optimized Approach for Concurrency Control in Hyperledger Fabric

Haoliang Ma[1,2(✉)], Peichang Shi[1,2], Xiang Fu[1,2], and Guodong Yi[3]

[1] National Key Laboratory of Parallel and Distributed Computing, College of Computer Science, National University of Defense Technology, Changsha 410073, China
[2] Key Laboratory of Software Engineering for Complex Systems, College of Computer Science, National University of Defense Technology, Changsha 410073, China
{mhliang0640,pcshi}@nudt.edu.cn
[3] Xiangjiang Lab, Changsha 410073, China

Abstract. As blockchain technology garners increased adoption, permissioned blockchains like Hyperledger Fabric emerge as a popular blockchain system for developing scalable decentralized applications. Nonetheless, parallel execution in Fabric leads to concurrent conflicting transactions attempting to read and write the same key in the ledger simultaneously. Such conflicts necessitate the abortion of transactions, thereby impacting performance. The mainstream solution involves constructing a conflict graph to reorder the transactions, thereby reducing the abort rate. However, it experiences considerable overhead during scenarios with a large volume of transactions or high data contention due to capture dependencies between each transaction. Therefore, one critical problem is how to efficiently order conflicting transactions during the ordering phase. In this paper, we introduce an optimized reordering algorithm designed for efficient concurrency control. Initially, we leverage key dependency instead of transaction dependency to build a conflict graph that considers read/write units as vertices and intra-transaction dependency as edges. Subsequently, a key sorting algorithm generates a serializable transaction order for validation. Our empirical results indicate that the proposed key-based reordering method diminishes transaction latency by 36.3% and considerably reduces system memory costs while maintaining a low abort rate compared to benchmark methods.

Keywords: Hyperledger Fabric · Reordering Algorithm · Concurrency Control · Transaction Conflicts

1 Introduction

Originating from Nakamoto's Bitcoin whitepaper [12], Bitcoin only supports cryptocurrency. Ethereum [1] was then developed to facilitate Turing-complete

Z. Tari et al. (Eds.): ICA3PP 2023, LNCS 14493, pp. 272–291, 2024.
https://doi.org/10.1007/978-981-97-0862-8_17

smart contracts, thus enabling arbitrary data processing logic. Consequently, the blockchain evolved from merely a cryptocurrency platform to a distributed transaction system. Traditional blockchain systems, such as Bitcoin and Ethereum, employ an Order-Execute(OE) model, whose sequential transaction execution characteristic restricts performance, as evidenced in an analysis of seven blockchain systems [14]. In contrast, Hyperledger Fabric leverages an Execute-Order-Validate (EOV) model to enhance performance: transactions submitted are first executed by the endorsing peers, then ordered and batched by the ordering services, and finally validated by the validating peers. Fabric exploits today's multi-core architecture to facilitate transaction processing by supporting parallel processing of transactions [2]. It overcomes the limitations of the OE model by providing parallelism of transaction execution on different endorsing peers.

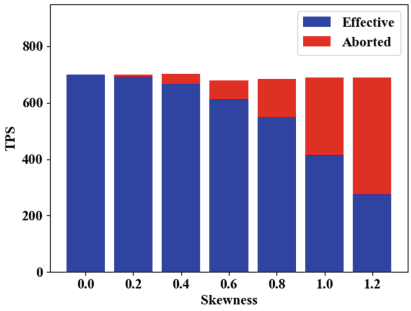

Fig. 1. Effective and aborted throughput under vaying skewness

However, the delay between execution and commitment of a transaction increases the probability of conflicting transactions, which are subsequently rejected by peers during the verification stage, thus creating a scalability bottleneck. The Fabric uses an optimistic concurrency control (OCC) mechanism, terminating conflicting transactions to ensure the consistency of the ledger under concurrent updates. However, this measure comes with a substantial transaction abort rate exceeding 40% [3] due to many inter & intra-block conflicts, amplified particularly under high contention workload characterized by a large number of conflicting transactions. Various degrees of data race conditions can be simulated by adjusting the skew parameters implying Zipfian distribution. It reveals that higher skewness corresponds to an increased percentage of conflicting transactions, e.g., $skew = 0$ represents uniform access, and $skew = 2.0$ represents extremely skewed access. Figure 1 reports Fabric's throughput under varying skewness [15], with its blue and red components, respectively, demonstrating effective and aborted throughput. The raw throughput remains consistent despite the workload type and requests skewness. But with higher skewness, a larger proportion of transactions are aborted for serializability.

Current studies [15,16] employ conflict graph construction, with Tarjan's and Johnson's algorithms [18] used for cycle detection and removal to decrease dis-

carded transactions during the sorting phase of transaction reordering. However, the overhead associated with conflict graph construction is significant due to the need to map dependencies between every transaction pair, especially when large transaction volumes or considerable conflicting transactions are present. As transactions increase, so do blocks that need to be processed, implying more conflict graphs need construction and processing. Heightened data contention amplifies this issue as each transaction potentially conflicts with a larger number of other transactions, leading to an increase in edges that may trigger out-of-memory issues. Additionally, Tarjan's and Johnson's algorithms require complex operations on the graph to identify strongly connected components or cycles, demanding considerable computational and memory resources. This approach can result in substantial delays and potential system failures. Therefore, an efficient algorithm or strategy for conflict graph construction and transaction processing and order is essential to alleviate system resource usage and manage an increased number of conflicting transactions.

We propose an efficient alternative: a key-based conflict graph (KCG) construction method that leverages key dependency to establish a global transaction order instead of capturing conflicting relationships between each pair of transactions in the conflict detection of transaction dependency-based strategies. This key dependency reveals transaction order on different keys, and more dependent transactions can be obtained on each key. Subsequently, a transaction sorting method is adopted to obtain a commit order. The advantage of our solution lies in its efficiency under high data contention. As conflicting transactions increase on each key, more dependent transactions can be detected, thus reducing system resource overhead. Utilizing the key-based reordering method, we can get a submission of non-conflicting transactions and achieve higher performance under concurrency conflicts on Fabric. The contributions of our paper are as follows:

- We present a theoretical classification of various types of concurrency-related transaction conflicts in Fabric and formulate the problem that our study aims to address.
- We introduce a key-based conflict graph construction approach, leveraging key dependency instead of the conventional transaction dependency, to efficiently resolve concurrency update conflicts in Hyperledger Fabric. Our solution proves particularly suitable for large transaction volumes under data contention. Notably, our method remains functional even when the CG method crashes due to Out-of-Memory (OOM) errors.
- We evaluate the performance of our solution and compare it with methods employed by vanilla Fabric and Fabric++/FabricSharp. Additionally, we conduct a sensitivity analysis to study the impact of different workload parameters on performances. Compared to these existing methods, our model diminishes transaction latency by 36.3% and considerably reduces system memory costs while maintaining a low abort rate.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 categorizes transaction conflicts and formulates the problem. In Sect. 4, we present the system model and propose our approach. Performance

evaluations are provided in Sect. 5, followed by a conclusion and future work in Sect. 6.

2 Related Work

Efficient handling of concurrency conflicts is a hot research topic in distributed databases, and conflicting transactions are also existing in Hyperledger Fabric which is a distributed system. Many studies have proposed the optimization of the performance for processing conflicting transactions. In this section, we will introduce these works along three categories according to the Fabric lifecycle: optimization for endorsement, ordering, and validation.

2.1 Endorsement Phase Optimization

Xu et al. [21] propose a lock mechanism to create a temporary database index for conflicting transactions, with the subsequent merging of the newly created index with the original index after the transaction is verified. However, in asynchronous blockchain systems, lock services are required to create and merge database indexes synchronously, resulting in substantial communication costs. Minsu et al. [9] introduce a read and write transactions separating method to accelerate transaction processing. Consequently, the transaction endorsement latency is reduced by 60% compared to the traditional Fabric network. Trabelsi et al. [19] offer a methodology to maintain a cache for conflict transaction detection at this stage and, based on this, compares three different cache storage strategies.

2.2 Ordering Phase Optimization

For the ordering phase, FastFabric [7] redesigns the ordering service to operate only with transaction IDs. By separating the transaction header from the payload, the process for determining transaction order is expedited, thus boosting throughput. Sharma et al. [16] introduce a reordering step immediately before block formation but after consensus, analyzes transaction conflicts by constructing a conflict graph, reorders and selectively discard transactions that cannot be serialized to determine a conflict-free transaction sequence, and eliminates Multi-version Concurrency Control (MVCC) Read Conflicts. Although Fabric++ reduces the number of conflicting transactions in a block, it does not apply a straightforward discarding strategy for cross-block transactions, limiting its reordering effect. Subsequently, Ruan et al. [15] consider transaction cross-block conflicts and varying conflict types based on the work of Fabric++. They proposed FabricSharp, a method capable of handling conflicts in a more fine-grained manner. However, the reordering algorithm has problems in usability and security [17]. In high-concurrency scenarios, the conflict graph becomes complex, and solving it can become a performance bottleneck and potentially even cause system crashes. To mitigate the overhead incurred by cycle detection and removal, Dickerson et al. [4] and FastBlock [11] introduce a happen-before

graph for transaction execution and employ assumptions about software and hardware configurations to detect conflicts. Nevertheless, this reliance is not supported by all blockchain nodes. The transaction reordering method is also adopted in other distributed transaction processing systems. Furthermore, this method is utilized for improving OCC in online transaction processing systems [5]. Xiao et al. [20] employ the key-based concurrency control method to resolve conflicting transactions in directed acyclic graph (DAG)-based blockchains.

2.3 Validation Phase Optimization

Multiple articles propose the parallel execution of the validation process (syntax verification, endorsement policy verification, MVCC validation) to accelerate block validation [6–8]. Gorenflo et al. [6] advocate for the XOX transaction process. He believes that if a transaction is only marked as invalid due to conflicts in the verification phase, there is no trust problem in the entire execution process of the transaction, so conflicts can be found during the verification phase. Then the node executes the transaction locally to get the latest result. However, this method ignores the trust problem that still exists in the alliance chain built by Fabric, and different nodes may maliciously write wrong data, resulting in ledger data errors. FabricCRDT [13] focuses on automatically merging conflicting transactions using CRDT techniques without rejecting them. However, this approach is only suitable for use cases that can be modeled with CRDT. Skipping MVCC verification makes FabricCRDT lose the ability to detect "double spend attacks."

3 Problem Definition

3.1 Types of Transaction Conflicts

There are three categories of transaction conflicts in Fabric [3]:

3.1.1 Endorsement Failure Conflicts: All transactions need to be endorsed in the execution phase. The reasons for endorsement failure include invalid endorsement signatures or other reasons such as configuration or network errors. In this article, we only focus on endorsement policy failures caused by read-write set mismatches. Every peer independently maintains a ledger using a key-value store, which will update independently by each peer in the validation phase. Therefore, transient world state inconsistencies between peers are possible. Moreover, in the execution phase, the tail delay of block propagation makes it impossible for each endorsement node in the organization to obtain the latest ledger status for the first time. Due to the inconsistency of the world state of peers, the error of read/write set mismatch is called endorsement failure conflict.

As illustrated in the Table 1, when two different endorsement nodes Peer1, Peer2 $\in$ P endorse the same transaction $T_i \in T$, the version numbers of the same value Key in the read-write set RWSet generated by Peer1 and Peer2 are inconsistent, and an error occurs in the endorsement phase.

Table 1. Example of Endorsement Failure Conflicts

Peers	Execution Phase		World State	
	Transaction from Client	Generated Read/Write Set	Key	Version
Peer 1	T_1[R(A), W(A)]	R(A, Version 1), W(A)	A	1
Peer 2	T_2[R(A), W(A)]	R(A, Version 2), W(A)	A	2

3.1.2 MVCC Read Conflicts: MVCC Read Conflicts arise in the transaction verification phase. MVCC is a low-cost optimistic concurrent access processing method widely utilized in database systems. Its core principle involves creating historical snapshots for read transactions, and for write transactions, a new version snapshot is created instead of overwriting original data.

During the verification process, each peer node examines the transactions within the current block sequentially, and compares the version number of each transaction's read set with the current world state. The peer ensures that the current ledger state is consistent with the state achieved by transaction simulation. If any key's version number in the read set doesn't match the present world state, the transaction is considered as invalid. MVCC Read Conflicts can happen under two circumstances:

Condition 1: A read-after-write conflict, where a transaction's read operation takes place after another transaction's write operation.

Condition 2: A stale read conflict happens because a node can be either a committing peer or an endorsing peer. During a transaction's transition from the execution to the validation phase, other transactions could get validated and committed to the chain, thereby updating the world state. Therefore, the ledger update turns the data read by the transaction into stale data.

Table 2. Example of an MCVV Read Conflict

Transactions	Validation Phase			World State	
	Transaction from Ordering Service	Read Set Version Matches World State	Status	Key	Version
1	T_1[R(A, Version 1)]	Yes	Success	A	1
2	T_2[W(A, Version 1)]	/	Success	A	1
3	T_3[R(A, Version 1)]	No	Fail	A	2
4	T_4[R(B, Version 1)]	No	Fail	B	2

A typical instance of MVCC read conflict is depicted in Table 2. Transaction 1 (T1) reads key A, whose world state version is the same as the one in the transaction's read set. Therefore, the read set contains the latest value of Key A. Transaction 2 (T2) modifies Key A's value, giving it a new version 2. For Transactions 3 and 4 (T3 and T4), that read Key A and Key B respectively, the world state and read set host different versions. This implies that T3 and T4

are accessing an older key version, hence, they fail. Specifically, T3 fails due to condition 1 and T4 fails owing to condition 2.

3.1.3 Write-Write Conflicts: In traditional databases, "write-write" concurrency conflicts primarily arise when multiple requests attempt to modify the same database index concurrently. Similarly, on blockchain platform like Fabric, when multiple transactions seek to modify the same ledger data simultaneously, it creates a similar concurrency problem. Although the data written later will overwrite the previous ones, these transactions in Fabric will eventually be submitted successfully and not marked as invalid. However, this process consumes system resources.

3.2 Problem Formulation

The problem we are focusing on is solving MVCC Read conflict with lower system overhead and acceptable latency. This is in contrast to the current problem resolved by Fabric++, which uses a CG of consuming a lot of resources. On EOV blockchain systems like Hyperledger Fabric, in the simulation execution phase, multiple peer nodes execute transactions in parallel to obtain read and write sets, which are then sent by the client to the ordering service for sorting and packaging and then verification. For the Hyperledger Fabric blockchain platform, its essence as a distributed database also has concurrency problems. We analyzed the concurrency problems in Sect. 3.1 and defined three types of transactions that cause transaction conflicts. In order to avoid MVCC Read Conflicts caused by the order of transactions in the validate phase, a transaction sequence that satisfies serialized execution can be obtained through a CG reordering method based on transaction dependencies instead of original first-in-first-out (FIFO) ordering, thus can reduce the aborted rate of transactions. CG reordering method is used to guide the ordering of conflicting transactions adopting transactions as vertices and transaction dependencies as edges. However, a transaction dependency only indicates the order between two transactions.

When constructing a conflict graph, its memory usage is closely related to the conflict graph relationship of transactions. As the capacity of transactions within a block or the skewness of transactions escalate, so does the count of conflicting transactions. As the skewness increases, the access pattern tends to concentrate on a small number of hotkeys. Smallbank workload corresponds to frequent asset update operations on a small number of accounts. In this case, the potential conflict between transactions will increase because they may more frequently access the same keys. This will increase the number of nodes and edges of the conflict graph, thereby increasing the complexity of the conflict graph, which in turn increases memory usage. Johnson's algorithm, which is used for identifying cycles in a strongly connected subgraph, can be done in linear time in $O((N + E)(C + 1))$, where N is the number of nodes and E is the number of edges, C is the number of cycles in the graph. Meanwhile, it uses a recursive algorithm called depth-first-search (DFS) for cycle detection, substantially

increasing the system resources overhead required for cycle detection and transaction processing latency. In extreme cases, an Out-Of-Memory (OOM) might occur, which can potentially lead to a system crash. That is why CG is not suitable for data contention situations. Hence, it motivates us to find a more efficient way to generate a commit order.

4 System Design

4.1 System Model

Supposing the client sends two transaction requests (T_u) and (T_v), and the endorsement node gets the transaction read-write set after execution: RS(T_u), WS(T_u), RS(T_v), WS(T_v). Assume that T_u and T_v are executed in parallel, and when any of the following conditions (1)(3) is true, T_u and T_v conflict with each other. The difference is that conflicting transactions that satisfy condition (1) will be marked as invalid transactions during the verification phase, while conflicting transactions that satisfy condition (3) are valid transactions, that is, transactions whose write sets are finally successfully applied to the state database.
(1) $WS(T_u) \cap RS(T_v) \neq \phi$
(2) $RS(T_u) \cap WS(T_v) \neq \phi$
(3) $WS(T_u) \cap WS(T_v) \neq \phi$

Definition 1. ***Transaction Dependency.*** *Given two transactions T_u and $T_v(u < v)$, when T_u is verified before T_v, a transaction dependency $T_u \rightarrow T_v$ exists if condition* (1) *is met, $T_u \xrightarrow{rw} T_v$, that is, read-write dependency, or when* (3) *is satisfied, $T_u \xrightarrow{ww} T_v$, that is, write-write dependency.*

Definition 2. ***Conflict Graph.*** *A conflict graph, denoted as CG, is a directed graph that consists of a set of vertices $V = \{T_1, T_2, ..., T_N\}$ and a set of edges $E = \{(T_u, T_v) | 1 \leq u \neq v \leq N, T_u \rightarrow T_v\}$. In this graph, $|V| = N$.*

Based on the captured transaction dependencies, a conflict graph that takes transactions as vertices and dependencies as edges is build. CG can guide the ordering of transactions to reduce over-aborting transactions that are still serializable.

Definition 3. ***Key Dependency.*** *Let's consider two distinct keys K_i and $K_j (i \neq j)$. We can say that K_i is dependent on K_j (notated as $K_i \dashrightarrow K_j$) if there exists a transaction T_v such that T_v^W belongs to RW_i and T_v^R is part of RW_j. Here, T_v^R and T_v^W denote the read and write units of transaction T_v, respectively.*

Definition 4. ***Key-based Conflict Graph.*** *A key-based conflict graph, denoted as $KCG = (V, E)$, is a directed graph where $V = \{RW_j | j = 1, 2, \ldots, n\}$, and $E = \{(RW_i, RW_j) | 1 \leq i \neq j \leq n, \exists v \in [1, N_e], T_v^W \in RW_i \wedge T_v^R \in RW_j\}$. Here, n represents the number of keys being accessed.*

The key dependency is identified between the read and write units of a transaction. Contrasting with transaction dependency, we methodically map these read and write units to the associated key queues and position them accurately within this sequence. Using the captured key dependency, we build a directed edge between the write-read units of each transaction across different keys. We employ these edges to organize the read/write sets of all keys into a novel conflict graph called KCG.

Table 3. Four concurrent transactions

Transaction	T_1	T_2	T_3		T_4
R/W Operation	R	W	R	W	W
Accessed Key	K_1	K_1	K_2	K_1	K_2
Total order	$T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_4$				

Due to the increased conflicts per key, as shown in Table 3, more dependent transactions can be obtained on each key. Such dependency speeds up the processing of all writes and reads by incorporating a relatively small yet fast solution compared with a transaction dependency.

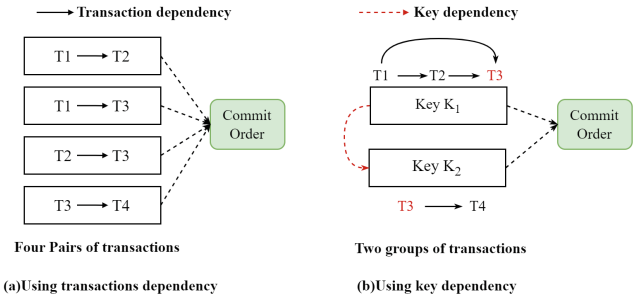

Fig. 2. Example of obtaining a commit order

Figure 2 presents a concrete example. It shows that employing the transaction dependency requires four pairs of dependent transactions to obtain the total order. Instead, it only requires two groups of dependent transactions detected by key K1 and K2 by relying on key dependency. Specifically, there are four transactions from T1 to T4 waiting ordering. In transaction dependency, if there is a dependency between two transactions, an edge is added, and at the same time, the transaction acts as a node. In key dependency, the read and write units of transactions that operate on the same key are stored in the same queue. We can see that employing the transaction dependency requires four pairs of dependent transactions to obtain the total order. Instead, it only requires two groups of dependent transactions detected by K1 and K2 by relying on key

dependency. Hence, comparing to transaction dependency, key dependency is more suitable for large number of transaction conflicts.

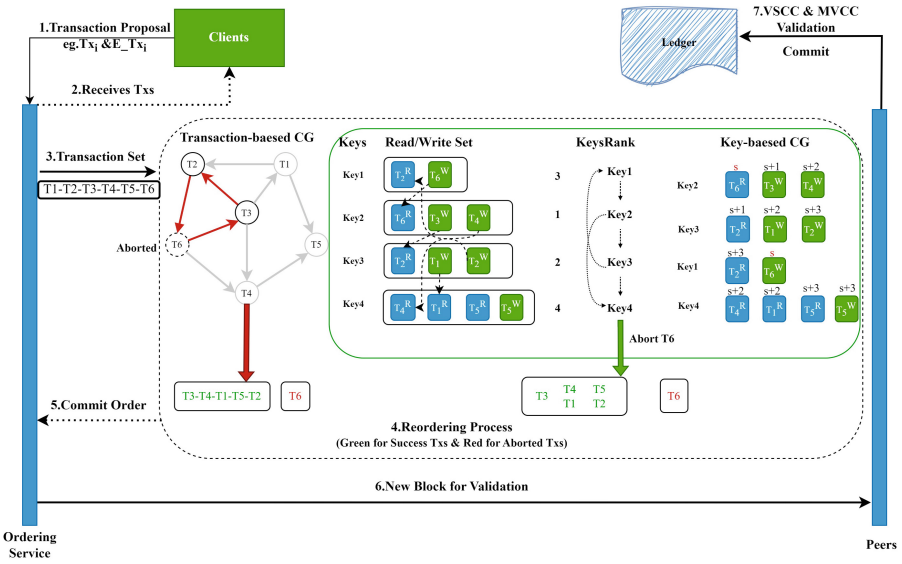

Fig. 3. Overview of System Model

To solve the concurrency conflict problem in Hyperledger Fabric, we propose a new transaction processing optimization method using a key-based transaction reordering algorithm. The main workflow of the system is shown in Fig. 3. The main goal is to get a serialized sorting after ordering service, which produces the least transaction discard and latency under the lowest system resources overhead.

The system workflow is as follows: the proposal is first signed and executed by the endorsing peers before it reaches the client. Then, the client will assemble endorsements into a transaction that contains the read/write sets, the endorsing peers' signatures, and the channel ID. Then send them to the Ordering Service to package and propagate. In the original version, the Ordering Service does not read the transaction details; it simply receives transactions from all channels in the network, orders them chronologically by channel, and creates blocks of transactions per channel. Therefore, you can take advantage of different ordering strategies to order the transactions inside a block to reduce the production of invalid transactions, such as transaction-based CG and key-based CG.

The main goal of the reordering algorithm is to generate a serialized sort in the middle of serialized transactions, ensuring that the least amount of transaction discards occur while consistent state transitions occur. The CG reordering algorithm using transaction dependency mainly includes five steps. They have directed conflict graph construction, subgraph division, cycle detection and

removal, cycle-free conflict graph construction, and topological sorting. With the increase of conflicting transactions, we discover that each key may exhibit more transaction dependencies due to the increased conflicts per key. So, we utilize keys to capture group dependency instead of pair dependency among transactions to alleviate the overhead of detecting all dependent transactions. As shown above, the main steps of the key-based reordering algorithm include graph construction, keys ranking, and transaction sorting, which realizes identifying the dependence among transactions. After the ordering, the block will be generated and delivered to all peers for validation and commitment.

4.2 Algorithm Design

In order to solve the concurrency conflict problems in Hyperledger Fabric, we propose a key-based transactions reordering algorithm to reduce the demand for system resources under high data contention and increase performance. Firstly, the key dependency is used to indicate the order of transactions on different keys so that more dependent transactions can be obtained on each key. Secondly, based on the key-based conflict graph (KCG), we use a transaction sorting method using read/write units in each queue of keys to efficiently obtain a total commit order of transactions. The above methods can effectively improve the performance of Hyperledger Fabric with MVCC Read Conflicts.

Algorithm 1 presents the Key-based Reordering Algorithm. The Procedure CreateGraph creates a graph whose nodes contain queues of read and write units, and the data structure of rwNodes in each node is create to record read/write sets. During the graph construction, the edges of the graph is first created, and then in each read-write node (rwNodes), according to its read-write set, (RWSet) generates queues for the keys and stores these queues in a list. Since each queue of keys maintains all dependent transactions that read and write to it, we can obtain a partial order between transactions on each key. As to transactions that read and write to multiple keys, we need to get their order using key dependency.

After building the KCG, the next step is to determine the specific order of each transaction. Based on key dependency, we can obtain the sorting priority of keys. Procedure KeysRank ranks vertices in a graph based on in-degree and out-degree. There may exist cycles among keys. This phenomenon is caused by unserializable transactions, which will drop in the final sorting process. After that, Procedure TransactionSorting is used to generate a commit order based on KCG. Inspired by Lamport's logical clock [10], we assign a unified sequence number for each read/write unit in the queue to represent their sequence in the total order. After removing unserializable transactions, we got a conflict-free transaction order by switching transactions with the same sequence number to a serial order for deterministic state transfer.

Algorithm 1. Key-based Reordering Algorithm

```
1: procedure CreateGraph
2:     Initialize edges, queueArray, and queues as empty dictionaries
3:     Construct a list of read/write nodes named rwNodes from set of transactions S
4:     for each rw in rwNodes do
5:         Create an edge with rw and an id
6:         Add the new edge to edges
7:         for each node n in rw do
8:             append n to the list of nodes associated with the string key in queueArray
9:         end for
10:    end for
11:    for each key in queueArray do
12:        Sort the nodes associated with the key into rSlice and wSlice
13:        Create a new queue with rSlice and wSlice, and add it to queues
14:    end for
15:    return a new QueueGraph with queues and edges
16: end procedure
17: procedure SortingRankDivision
18:    Initialize a sequence seq to represent sorting ranks
19:    if G.vertices == ∅ then
20:        return
21:    end if
22:    Find the minimum in-degree in G, assign it to min
23:    for each K_j in G do
24:        if A_j.inDegree == min then
25:            Select A_j and append it to seq, then break
26:        end if
27:    end for
28:    if min > 0 then
29:        Find the first keys with maximum out-degree in minAddrs, append it to seq
30:    end if
31:    Remove the vertex and edges of the selected vertex from G
32:    Recursively call SortingRankDivision with the updated G
33: end procedure
34: procedure TransactionSorting
35:    Initialize initialSeq with seq from SortingRankDivision
36:    Find read units with sequence numbers in RW_j, assign it to sortedRSet
37:    if sortedRSet is empty then
38:        Assign initialSeq to sequence in RW_j and update maxRead to initialSeq
39:    else
40:        Find the minimum and maximum sequence numbers in sortedRSet, assign maxRead to maxSeq
       and update sequence of remaining units in RW_j to minSeq
41:    end if
42:    Find write units with sequence numbers in RW_j, assign it to sortedWSet
43:    if sortedWSet contains a unit whose read unit exists in RW_j then
44:        Increment sequence number
45:    end if
46:    for each unit in sortedWSet do
47:        if its sequence is less than maxRead then
48:            Abort the unit
49:        end if
50:    end for
51:    for remaining units in RW_j do
52:        while writeSeq is assigned do
53:            Increment writeSeq and then assign writeSeq to their sequence
54:        end while
55:    end for
56: end procedure
```

5 Experimental Evaluation

In this section, we present a comprehensive evaluation of our key-based reordering method. We first describe the experimental setup for our prototype and the workload in our experiments. Then, we evaluate the performance of KCG against FIFO adopted by vanilla Fabric and CG used by Fabric++.

5.1 Experimental Setup

Table 4. Experiment Configuration

Parameters	Values
Number of users	10,000
World State Database	LevelDB
Number of transactions per block(block size)	100
Probability for picking a read transactions (Pr)	50%
s-value of Zipfian distribution	0.8

Environment: This paper designs a blockchain prototype system implemented in GO 1.19 including simulated execution, sorting and verification of the Fabric. During the simulation phase, we adopt LEVM (Little Ethereum Virtual Machine) to provide an execution environment for our smart contracts written in Solidity. The open source panjf2000/ants library is used to provide the ability to manage and recycle a massive number of goroutines to simulate multiple peer nodes executing transactions in parallel. In the sorting phase, three ordering algorithms, FIFO, CG and KCG, are implemented. Table 4 respectively presents the various parameters that we have configure for system and workload.

Workloads: We use SmallBank as the workload, which simulates typical asset transfer scenarios. It provides 6 types of transactions for operating the data, including 5 update transactions and 1 read transaction. The read transaction is selected with probability P_r, while one of the five update transactions is chosen with a probability $1 - P_r$. The degree of skewness influences the distribution of read/write operations among the 10,000 available accounts. A higher skewness indicates a greater concentration of read/write operations on a smaller subset of accounts, leading to an increased potential for conflicts (Table 5).

Table 5. Transaction Types in SmallBank Workload

Transaction	Implication
CreateAccount	Initialize random funds for each customer's checking and savings accounts
TransactSavings	Add a certain amount of money to a savings account
DepositChecking	Add a certain amount of money to a checking account
WriteCheck	Indicates the removal of an amount from a customer's checking account
SendPayment	Transfer funds between two checking accounts
Amalgamate	Transfer all funds from a savings account to a checking account
Query	Query about the amount of a customer's savings or checking account

5.2 Results

This section presents a comprehensive analysis of the performance metrics of blockchain systems, specifically focusing on the influence of various parameters and different ordering strategies. We parameterize three key factors: 1) the skew parameter of the Zipfian distribution, 2) the block size, and 3) the percentage of read transactions. Our benchmarking analysis primarily considers latency and abort rate as the key metrics. It is worth mentioning that the ordering method employed in Fabric is abbreviated as FIFO, while the conflict graph utilized in Fabric++ and Fabricsharp is referred to as CG. Our proposed solution is denoted as KCG. For all experiments, we conduct ten runs for each parameter, and the reported results are the average values of these sum runs.

5.2.1 Impact of Block Size: To evaluate the impact of block size on performance, we increase the number of transactions in each block from 50 to 200. We set the percentage of read transactions as 50% and the skew parameter as 0.8. For other parameters, refer to Table 4.

Figure 4(a) shows the transaction abort rate under different strategies. The results indicate that all three systems experience a drop in transaction rates ranging from 5% to 15% when the block size is set to 50 and 100. However, the CG strategy encounters memory-related failures when the block size increases from 100 to 150. This is attributed to the increasing number of transactions within a block, leading to a higher likelihood of conflicts and, consequently, a more complex conflict graph. This is attributed to the increasing number of transactions within a block, leading to a higher likelihood of conflicts and, conse-

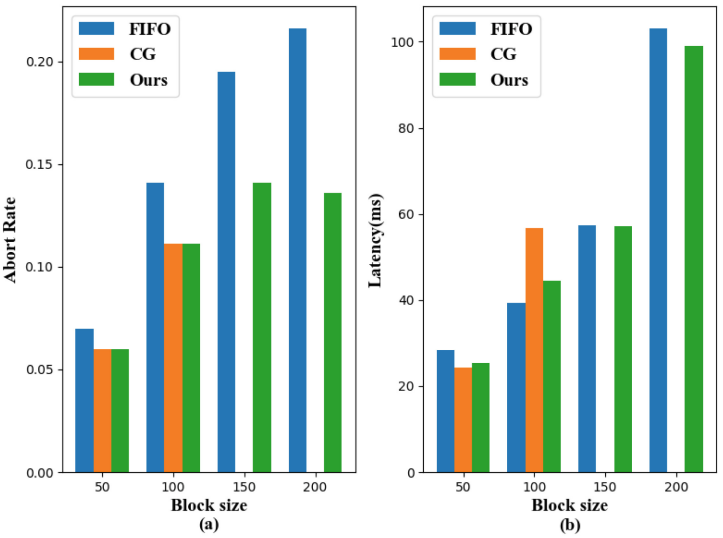

Fig. 4. Impact of the block size on transaction (a) aborting rate, (b) latency of FIFO, CG and ours.

quently, a more complex conflict graph. In the CG strategy, the DFS algorithm is utilized for searching, and when numerous cycles are present, memory consumption becomes significant as a new object is created for each cycle detected in a strongly connected subgraph. Our proposed KCG strategy maintains a transaction abort rate that is lower compared to FIFO and equal to or lower than CG. Moreover, unlike CG, our method can handle larger block sizes such as 150, 200, or even larger. This is possible because KCG resolves conflicting transactions by leveraging key dependency rather than transaction dependency. Key dependency denotes the order of transactions on different keys, allowing for faster processing of all writes and reads through the incorporation of a relatively small yet efficient solution.

Figure 4(b) shows the average latency under different block sizes. It is observable that the average latency escalates with the increase in block size, while FIFO always maintains a low latency. These results are expected since both CG and our KCG require additional reordering processing to resolve conflicting transactions in the ordering phase. Additionally, more transactions included in a block result in longer processing time, thus leading to higher latency. But the latency in KCG is always lower than CG and is comparable with FIFO. This is because the KCG method does not require the time-consuming cycle detection and removal stages under each strongly connected component within the graph. Furthermore, non-serializable transactions are directly discarded. Consequently, compared to FIFO, the number of transactions necessitating verification within KCG is reduced, thereby mitigating the delay. Hence, KCG proves to be a highly efficient method in high contention scenarios with concurrency conflicts.

5.2.2 Impact of Transaction Contention: Next, we conduct a comprehensive examination of transaction contention and its effects on overall performance. We set varying degrees of transaction contention by adjusting the skew parameter of the Zipf distribution from 0.2 to 1.2 in steps of 0.2.

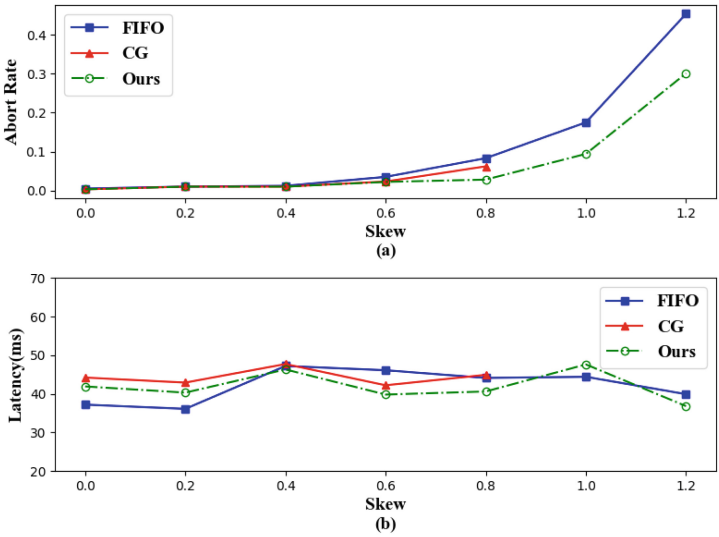

Fig. 5. Impact of skew parameter on transaction (a) aborting rate, (b) latency of FIFO, CG, and ours.

Figure 5(a) summarizes the results of the transaction abort rate under various skew parameters. We can see that for a small skew parameter (< 0.6), the transaction abort rate of all three systems is relatively low because the number of potentially conflicting transactions is small. However, an increase in the transaction abort rate of FIFO is witnessed with higher skew parameters (> 0.6), attributed to the fact that high data skewness in the Smallbank workload leads to a large number of potentially conflicting transactions. What's worse, the CG process fails due to exhausted memory when the skew exceeds 0.8. On the contrary, KCG's abort rate is always lower than FIFO. When the number of conflicting transactions is small (with skew below 0.8), this can be resolved by CG adopted by Fabric++. However, when it is set to 1.0 or higher, CG is prone to failure due to memory exhaustion, and OOM occurs. Contrarily, our KCG allows the system to operate normally for all skew degrees. KCG is not sensitive to the increase of contention degree and demands fewer system resources due to avoid building an edge between each pair of dependent transactions. Specifically, KCG assigns each transaction to the corresponding keys. Unique keys housing dependent transactions construct a novel scheduling graph, known as the key-based conflict graph (KCG), where edges capture key dependencies.

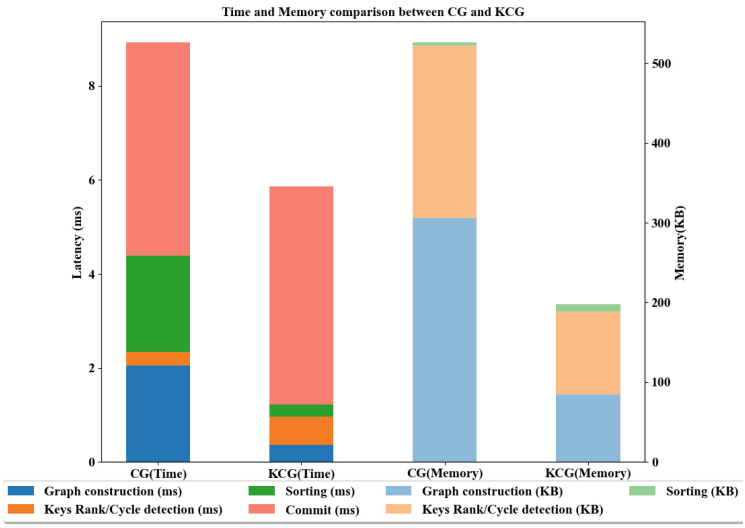

Fig. 6. Latency and Memory Cost of each sub-phase in CG and KCG.

When a skew parameter is set to 1.0 or higher, the rise in conflicts per key yields more dependent transactions on each key. This incurs substantial computational overhead to establish total order, rendering it inefficient in scenarios with high contention and a large volume of conflicting transactions. The result demonstrates KCG improves the transactions abort rate compared to FIFO. We compare the average latency in Fig. 5(b). We observe that our KCG latency is always lower than CG. Moreover, when the skew surpasses 0.8, the CG method becomes inapplicable. This is due to the increase in conflicting transactions leading to a larger transaction processing time in CG. In contrast, it has a relatively minor influence on our KCG system, where the rise in conflicts results in a limitation of accessed keys.

As presented in Fig. 6, we evaluate the latency and memory cost of each sub-phase in CG and KCG. The first and second columns represent the utilization of time. Overall, the latency of KCG is 36.3% lower than that of CG, where the time excluding the transaction commit stage is nearly equivalent across all systems. However, latency for other time periods in KCG is markedly lower than the corresponding stage latency in CG. Particularly under high data contention scenarios with a skew rising to 1.0, the latency for cycle detection and removal in CG substantially increases. This change is due to the recursive Johnson's algorithm, which has a time complexity of $O((|V|+|E|)\cdot(C+1))$, and consumes a significant amount of memory when the number of cycles is large. Turning to the third and fourth columns, they indicate the utilization of memory resources. The sorting phase in both systems only constitutes a small fraction, with the KCG method outperforming CG in the other two stages. We can see that the graph construction and keys rank/cycle detection occupy a large portion of memory due

to the overhead involved in establishing an edge between each pair of dependent transactions and the object overhead created for each cycle during the process of cycle detection in strongly connected components.

5.2.3 Impact of Read Transaction Percentage: This section presents an evaluation of how the percentage of read transactions affects the performance of simulated blockchain systems. We vary the percentage of read transactions from 0.1 to 0.9 in steps of 0.2. Figure 7(a) illustrates the impact of the percentage of read transactions on the transaction abort rate. It can be observed that as the percentage of read transactions increases, the transaction aborting rate decreases. This can be attributed to fewer conflicting transactions occurring when a higher proportion of read transactions is present, resulting in fewer updates to transaction records within a short time slot. Similar to the previous case, the CG method is not applicable when there are too many cycles in the conflict graph, while the proposed KCG method remains effective under any read transaction rate. Hence, the transaction abort rate of KCG outperforms that of FIFO and CG.

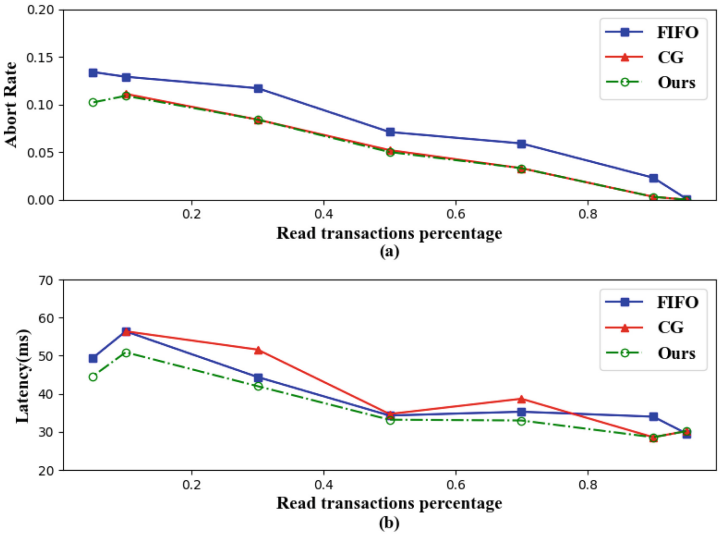

Fig. 7. Impact of the read transaction percentage on (a) aborting rate, (b) latency of FIFO, CG, and ours.

Figure 7(b) depicts the average latency of the three systems. As the proportion of read transactions increases and the proportion of write transactions decreases, the average latency shows a decreasing trend. Specifically, KCG demonstrates lower latency compared to FIFO and CG. However, when the read

transaction proportion is as low as 10%, there is a possibility of CG experiencing errors or taking significantly longer than expected, primarily due to the time-consuming cycle detection and removal. To summarize, the aforementioned experiments demonstrate that the key-based CG construction method enables efficient concurrency control and yields few transaction aborts in scenarios with considerable transactions and conflicts. This method proves to be more suitable for high contention scenarios than the compared schemes.

6 Conclusion and Future Work

This paper focuses on enhancing the efficiency of handling concurrency conflicts in Hyperledger Fabric. We propose a key-based transaction reordering algorithm called KCG, which effectively resolves conflicts with minimal overhead. We first adopted a key-based transaction conflict graph construction method to replace transaction dependency to support parallel transaction processing. Then, the keys rank and sorting method is used to generate a transaction commit order. Through evaluations conducted on a real workload using Smallbank, we demonstrate that the key-based ordering method outperforms both the original and CG ordering method adopted by Fabric and Fabric++ permissioned blockchain systems, particularly as the number of conflicting transactions increases. In future work, we will further analyze resource consumption in concurrent transaction processing and strive to improve the abort rate and latency of Hyperledger Fabric.

Acknowledgement. The authors gratefully acknowledge the financial support provided by National Key R&D Program of China (No. 2022ZD0115302), in part by the National Natural Science Foundation of China (No. 62202479, No. 61772030), the Major Program of Xiangjiang Laboratory (No. 22XJ01004) and the Major Project of Technology Innovation of Hunan Province (No. 2021SK1060-1).

References

1. Ethereum (2023). https://www.ethereum.org/zh/. Accessed 25 May 2023
2. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, pp. 1–15 (2018)
3. Chacko, J.A., Mayer, R., Jacobsen, H.A.: Why do my blockchain transactions fail? A study of hyperledger fabric. In: Proceedings of the 2021 International Conference on Management of Data, pp. 221–234 (2021)
4. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 303–312 (2017)
5. Ding, B., Kot, L., Gehrke, J.: Improving optimistic concurrency control through transaction batching and operation reordering. Proc. VLDB Endow. **12**(2), 169–182 (2018)

6. Gorenflo, C., Golab, L., Keshav, S.: XOX fabric: a hybrid approach to blockchain transaction execution. In: 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 1–9. IEEE (2020)
7. Gorenflo, C., Lee, S., Golab, L., Keshav, S.: Fastfabric: scaling hyperledger fabric to 20 000 transactions per second. Int. J. Network Manage **30**(5), e2099 (2020)
8. István, Z., Sorniotti, A., Vukolić, M.: StreamChain: do blockchains need blocks? In: Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers, pp. 1–6 (2018)
9. Kwon, M., Yu, H.: Performance improvement of ordering and endorsement phase in hyperledger fabric. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 428–432. IEEE (2019)
10. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In: Concurrency: the Works of Leslie Lamport, pp. 179–196 (2019)
11. Li, Y., et al.: FastBlock: accelerating blockchains via hardware transactional memory. In: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), pp. 250–260. IEEE (2021)
12. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. Decentralized business review, p. 21260 (2008)
13. Nasirifard, P., Mayer, R., Jacobsen, H.A.: FabricCRDT: a conflict-free replicated datatypes approach to permissioned blockchains. In: Proceedings of the 20th International Middleware Conference, pp. 110–122 (2019)
14. Reijsbergen, D., Dinh, T.T.A.: On exploiting transaction concurrency to speed up blockchains. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pp. 1044–1054. IEEE (2020)
15. Ruan, P., Loghin, D., Ta, Q.T., Zhang, M., Chen, G., Ooi, B.C.: A transactional perspective on execute-order-validate blockchains. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 543–557 (2020)
16. Sharma, A., Schuhknecht, F.M., Agrawal, D., Dittrich, J.: Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In: Proceedings of the 2019 International Conference on Management of Data, pp. 105–122 (2019)
17. Sun, Q., Yuan, Y.: GBCL: reduce concurrency conflicts in hyperledger fabric. In: 2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS), pp. 15–19. IEEE (2022)
18. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
19. Trabelsi, H., Zhang, K.: Early detection for multiversion concurrency control conflicts in hyperledger fabric. arXiv e-prints arXiv:2301.06181 (2023). https://doi.org/10.48550/arXiv.2301.06181
20. Xiao, J., Zhang, S., Zhang, Z., Li, B., Dai, X., Jin, H.: NEZHA: exploiting concurrency for transaction processing in DAG-based blockchains. In: 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), pp. 269–279. IEEE (2022)
21. Xu, L., Chen, W., Li, Z., Xu, J., Liu, A., Zhao, L.: Solutions for concurrency conflict problem on hyperledger fabric. World Wide Web **24**, 463–482 (2021)