2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-7177-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDCS54860.2022.00034

NEZHA: Exploiting Concurrency for Transaction Processing in DAG-based Blockchains

Jiang Xiao*, Shijie Zhang*, Zhiwei Zhang†, Bo Li§, Xiaohai Dai*, and Hai Jin*
*National Engineering Research Center for Big Data Technology and System
*Services Computing Technology and System Lab, Cluster and Grid Computing Lab
*School of Computer Science and Technology, Huazhong University of Science and Technology, China
†Beijing Institute of Technology, China
§The Hong Kong University of Science and Technology, Hong Kong
Email: {jiangxiao, shijiezhang, xhdai, hjin}@hust.edu.cn, zwzhang@bit.edu.cn, bli@ust.hk

***Abstract*—A *Directed Acyclic Graph* (DAG)-based blockchain with its inherent parallel structure can potentially significantly improve the throughput performance over conventional blockchains. Such a performance improvement can be further enhanced through concurrent transaction processing. This, however, brings new challenges in concurrency control design in that there is an increased number of concurrent reads and writes to the same address in a DAG-based blockchain, which leads to a considerable rise of potential conflicts. Therefore, one critical problem is how to effectively and efficiently detect and order conflicting transactions. In this work, for the first time, we aim to improve system throughput and processing latency by exploring the address dependencies among different transactions. We propose NEZHA, an efficient concurrency control scheme for DAG-based blockchains. Specifically, NEZHA intelligently constructs an *address-based conflict graph* (ACG) while incorporating address dependencies as edges to capture all conflicting transactions. To generate a total order between transactions, we propose a *hierarchical sorting* (HS) algorithm to derive sorting ranks of addresses based on the ACG and sort transactions on each address. Extensive experiments demonstrate that, even under high data contention, NEZHA can increase the throughput over the conventional conflict graph scheme by up to 8×, while decreasing the transaction processing latency up to 10×.**

***Index Terms*—DAG-based blockchain, Transaction processing, Concurrency control, Conflict graph**

I. INTRODUCTION

Directed Acyclic Graph (DAG)-based blockchain is an emerging type of blockchain that utilizes its inherent parallel structure to overcome the throughput performance limitations. Pioneer DAG-based blockchains, such as Conflux [1] and OHIE [2], employ the DAG structure to generate multiple blocks in parallel. These DAG-based blockchains could outperform the conventional blockchain systems, have become pervasive to meet the high-performance demand of a wide range of applications.

Such a performance improvement can be enhanced through concurrent transaction processing [3], [4], which can exploit today's multi-core architectures to alleviate transaction processing overhead in a DAG-based blockchain system. However, processing transactions from multiple blocks concurrently can raise the number of transactions processed based on the same blockchain state. This may increase concurrent reads and writes to the same address, thereby causing a considerable rise in terms of potential conflicts. Concurrency control [5], [6] is a necessary design in DAG-based blockchains to detect all conflicts and reconcile them to guarantee correctness.

TABLE I
THEORETICAL RESULTS OF THE NUMBER OF CONFLICTS IN A DAG-BASED BLOCKCHAIN

Block concurrency	2	4	6	8
# Total conflicts	$780p$	$3,160p$	$7,140p$	$12,720p$
# Average conflicts per address	$26p$	$56p$	$106p$	$150p$

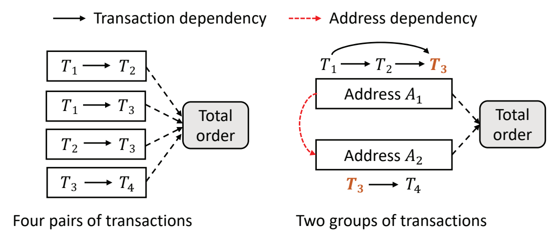
* We set the block size as 20 transactions and assume each transaction follows a fixed Zipfian distribution to access 10k accounts.

To understand the potential conflicts in a DAG-based blockchain, we first analyze the number of conflicting transactions with the increase of block concurrency. We use p to denote the probability of a conflict between any two transactions and calculate the total conflicts based on the number of pairs of transactions. Table I reveals that the potential conflicts exhibit a power-law growth as the number of concurrent blocks increases, as does the average number of conflicts on each address. The considerable rise in potential conflicts may pose two key challenges to enable the efficient concurrency control in DAG-based blockchains.

Challenge 1: The mismatch between the detection mechanism and the increased number of conflicting transactions. Detecting conflicts requires capturing conflicting relationships between each pair of transactions. It is impractical in a DAG-based blockchain since the power-law growth of potential conflicts can cause the overhead of detecting them to increase in a similar (power-law) manner.

Challenge 2: Efficient ordering the concurrent transactions to reconcile conflicts. Reconciling conflicts is to order all conflicting transactions for serializability [6]. However, the considerable potential conflicts in DAG-based blockchains may enlarge the overhead of determining a total order between conflicting transactions. *Optimistic concurrency control* (OCC) adopted in Fabric [7] can mitigate such overhead by aborting conflicting transactions. However, this is done at the cost of a high transaction abort rate of more than 40% [8], which can be enlarged in DAG-based blockchains due to the considerable conflicts. Hence, achieving efficient ordering with few transaction aborts is a non-trivial task.

Transaction	T_1	T_2	T_3		T_4
Read/write operation	*r*	*w*	*r*	*w*	*w*
Accessed address	A_1	A_1	A_2	A_1	A_2
Total order	$T_1 \succ T_2 \succ T_3 \succ T_4$				



(a) Using transaction dependency (b) Using address dependency

Fig. 1. Obtaining a total order between four concurrent transactions by using (a) transaction dependency and (b) address dependency

Unfortunately, efficient concurrency control remains an unsolved issue in the current DAG-based blockchains since they still employ a serial manner to process transactions to ensure consistent state transition. To address the above challenges, a preliminary approach is to exploit the potential dependencies between conflicting transactions, which can specify the correct order that satisfies serializability between them [9]. Recent studies [4]–[6], [10] employ transaction dependencies as edges to build a directed graph called *conflict graph* (CG) to guide the ordering of conflicting transactions with few aborts. Still, a transaction dependency only indicates the order between two transactions. It can incur a substantial computation overhead to obtain a total order, which is inefficient in DAG-based blockchains with a large number of conflicting transactions.

Hence, it motivates us to find a more efficient way to capture dependencies and generate a total order. The key insight in this paper is that *we explore the address dependency that indicates the order of transactions on different addresses.* Due to the increased conflicts per address as shown in Table I, more dependent transactions can be obtained on each address. Such address dependency speeds up the processing of all writes and reads by incorporating a relatively small yet fast solution compared with a transaction dependency. Figure 1 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 addresses A_1 and A_2 by relying on address dependency.

With this key insight, we propose a concurrency control scheme, named NEZHA, to efficiently detect and reconcile a large number of conflicts in DAG-based blockchains. Targeting the first challenge, NEZHA employs addresses to detect all dependent transactions. Specifically, NEZHA maps each transaction to the corresponding addresses rather than capturing dependencies between each pair of transactions. Different addresses containing dependent transactions constitute a novel scheduling graph called *address-based conflict graph* (ACG), where edges are used to capture address dependencies.

To efficiently obtain a total order between transactions, we devise a *hierarchical sorting* (HS) algorithm based on address dependencies. To ensure the validity of sorting results, HS first utilizes address dependencies to determine the proper sorting ranks of addresses. Then, HS sorts transactions on each address according to transaction dependencies between them. HS addresses the second challenge in the following aspects: i) utilizing address dependencies can alleviate the sorting overhead; ii) HS can identify all unserializable transactions during sorting without requiring additional detection; iii) HS can further reduce transaction aborts by reordering transactions with multiple write operations. Moreover, HS can yield a serialization order with a certain degree of concurrency, ensuring non-conflicting transactions can commit concurrently.

In summary, the contributions of our work are as follows.

- To the best of our knowledge, we are the first to focus on the conflict issue in the concurrent transaction processing of DAG-based blockchains. We propose an efficient concurrency control scheme NEZHA to detect and reconcile considerable conflicts for DAG-based blockchains.
- We devise a novel scheduling graph named *address-based conflict graph* (ACG) that can detect all conflicting transactions with modest overhead.
- We propose a *hierarchical sorting* (HS) algorithm based on address dependencies that can efficiently form a total serialization order among conflicting transactions with few transaction aborts.
- Experimental results demonstrate that, under high data contention, NEZHA can increase the effective throughput over the CG scheme by up to 8×, while decreasing the transaction processing latency up to 10×.

The remainder of the paper is organized as follows. Section II introduces the background and related work. The system overview is presented in Section III. The design details of NEZHA are presented in Section IV. Section V and Section VI describe implementations and evaluations of NEZHA. Lastly, Section VII concludes the paper.

II. Background and Related Work

This section starts with a brief introduction to mainstream DAG-based blockchains and the transaction processing framework suitable for them. Then, we review some related works on concurrency control in conventional blockchains.

A. DAG-based Blockchains

To improve blockchain scalability, some approaches utilize a *Directed Acyclic Graph* (DAG) structure to organize transactions or blocks. From the perspective of topological structure, the current DAG-based blockchains can be divided into three types: natural DAG, main chain-based DAG, and parallel chain-based DAG. In the natural DAG such as IOTA [15] and SPECTRE [16], newly released transactions or blocks can be arbitrarily appended to existing vertices, causing the DAG topology to grow in unpredictable directions. This irregular topology increases the difficulty of obtaining the total order of transactions, limiting their support for smart contracts [17].

In the latter two types of DAG-based blockchains, different consensus nodes can propose blocks in parallel. These

TABLE II
COMPARISON OF NEZHA WITH EXISTING CONCURRENCY CONTROL SCHEMES

Category	Approach	Concurrency		No Software/Hardware Assumption	Efficient Under Considerable Conflicts
		Concurrent Execution	Concurrent Commit		
PCC	PEEP [11]	✔	✔	✔	✖
OCC	Fabric [7]	✔	✖	✔	✖
	SlimChain [12]	✔	✖	✔	✖
OCC with CG	Fabric++ [5]	✔	✖	✔	✖
	FabricSharp [6]	✔	✖	✔	✖
	ParBlockchain [10]	✔	✖	✔	✖
	Jin et al. [4]	✔	✔	✔	✖
	Dickerson et al. [13]	✔	✔	✖	✖
	FastBlock [14]	✔	✔	✖	✖
	NEZHA	✔	✔	✔	✔

NOTE: ✔ : support ✖ : poor support

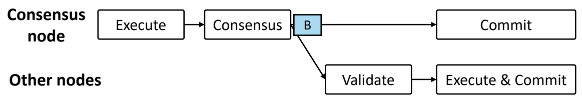
concurrent blocks are organized into a fixed DAG structure, leading to a directed topology growth. Conflux [1] and Prism [18] employ a main chain to guide the growth direction of DAG topology. In other works, multiple single chain instances are organized into a parallel chain structure, e.g., OHIE [2], Occam [19], Hashgraph [20], and DAG-Rider [21]. These two types of DAG-based blockchains have become the current mainstream due to their high security, good scalability, and ability to support smart contracts.

B. Transaction Processing Framework in Blockchains

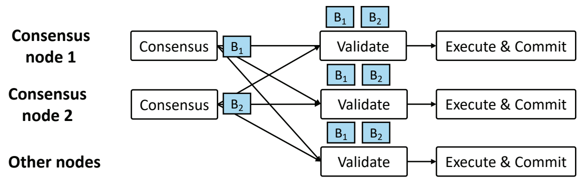
In DAG-based blockchains adopting the main chain or parallel chain structure, concurrent blocks are processed sequentially according to the predefined total order, and transactions in each block are also processed in a serial manner. Although the DAG structure can improve transaction throughput, the increased number of transactions also amplifies the defect of serial processing, i.e., it may weaken the performance improvement brought by the parallel generation of blocks. Exploring concurrent transaction processing for DAG-based blockchains is still a gap in the literature.

In contrast, in conventional blockchains with a single-chain structure, there exist many studies focusing on concurrent transaction processing to break the performance limitation incurred by serial transaction processing [4], [10]–[14]. Although those works vary in design details, they mostly follow a general transaction processing framework. As illustrated in Figure 2(a), the consensus node (i.e., block proposer) first simulates transaction executions to generate the transaction scheduling information (e.g., conflict graph) and includes it into a block. Then, all nodes participating in the consensus utilize a consensus protocol such as PoW (*Proof of Work*) or PBFT (*Practical Byzantine Fault Tolerance*) [22] to agree upon the new block. Once receiving the new block, other nodes validate it and commit a batch of transactions deterministically based on the proposed scheduling information. The processing framework in Fabric [7] and its related optimizations [5], [6] are slightly different from that described above, i.e., the process of simulating transaction executions is handled by a group of dedicated nodes called endorsers.

However, such a transaction processing framework widely adopted by conventional blockchains is not suitable for DAG-based blockchains. In DAG-based blockchains, if different consensus nodes execute their packed transactions before generating blocks, they will yield divergent execution results. Verifying these execution results requires other nodes to replay transactions in each concurrent block, which incurs huge overhead in verifying blocks. To mitigate this, a subtle adjustment to the overall framework is required, e.g., deferred execution in Conflux [1]. As shown in Figure 2(b), the execution phase before consensus is moved to the end. This adjustment can also enable consensus nodes and other nodes to process concurrent blocks in parallel after the consensus phase.



(a) Transaction processing in conventional blockchains



(b) Transaction processing in DAG-based blockchains

Fig. 2. Comparison of transaction processing framework

C. Concurrency Control in Blockchains

Concurrency control is the main approach to resolve conflict issues in concurrent transaction processing, which has been widely adopted in the field of database and blockchain. Recent concurrency control works in conventional blockchains can be divided into the following categories.

1) ***Pessimistic concurrency control* (PCC)**. PEEP [11] employs the ordered locking mechanism to eliminate potential conflicts from concurrent execution. However, the additional overhead incurred by locks may affect system performance, making it inapplicable to DAG-based blockchains.

2) ***Optimistic concurrency control* (OCC)**. Since no read/write locks are required, OCC is commonly adopted in blockchain transaction processing, e.g., Fabric [7] and SlimChain [12]. It forms a conflict-free execution environment by simply aborting transactions that conflict with confirmed

transactions. However, this scheme suffers from a high transaction abort rate under a high contention workload, which affects the system's effective throughput.

3) **OCC with *conflict graph* (CG)**. To reduce the transaction abort rate, some works [4]–[6], [10], [13], [14] leverage additional scheduling information, such as a *conflict graph* (CG), to enhance OCC. However, the construction overhead of CG is large under considerable conflicting transactions due to the need to capture dependencies between each pair of transactions. Moreover, detecting and resolving unserializable transactions in CG is still costly. Fabric++ [5] and FabricSharp [6] define this step as the problem of detecting and removing all the cycles in the graph. To reduce the overhead incurred by cycle detection and removal, Dickerson et al. [13] and FastBlock [14] respectively utilize *software transaction memory* (STM) and *hardware transactional memory* (HTM) to detect conflicts and generate a happen-before graph for transaction execution. Nevertheless, STM and HTM introduce the assumption of software and hardware configurations, which are not supported by all blockchain nodes.

Table II presents a comparison of the above concurrency control schemes with our work. To sum up, these concurrency control schemes cannot be well applied to DAG-based blockchains. Compared to them, NEZHA is able to efficiently resolve considerable conflicts in DAG-based blockchains.

III. System Overview

In this section, we provide an overview of the underlying system model and the concurrent transaction processing workflow for DAG-based blockchains. Then we introduce a strawman concurrency control scheme and its limitations.

A. System Model

Our focus in this paper is the DAG-based blockchains adopting the main chain or parallel chain structure, which can generate multiple valid blocks in parallel. We assume the underlying DAG-based blockchain employs the account-based data model instead of the UTXO-based data model, since the conflict issue caused by concurrent reads and writes to the same account address only occurs in the account-based model.

We can model the underlying DAG-based blockchain as $\mathcal{B} = \{B_e \mid e \geq 0\}$, where B_e is a set of valid concurrent blocks generated in epoch e. We assume the number of blocks (i.e., block concurrency) in epoch e is ω_e. B_e containing ω_e blocks can be denoted as $B_e = \{b_k^e \mid 0 \leq k < \omega_e\}$.

B. Concurrent Transaction Processing Workflow

We employ the transaction processing framework presented in Figure 2(b). To be specific, different consensus nodes directly broadcast newly generated blocks to the whole network without executing transactions. After receiving multiple concurrent blocks B_e in epoch e, each node concurrently processes all transactions from B_e based on the state snapshot of the last epoch $e-1$. Hence, the system state is updated in units of all concurrent blocks in each epoch. If no duplicate transactions appear in B_e, the number of transactions to be processed in epoch e is $N_e = \sum_{k=0}^{\omega_e} s_k$, where s_k denotes the number of transactions in block b_k^e.

Based on this framework, we propose a concrete concurrent transaction processing workflow for DAG-based blockchains, which contains the following four phases.

- **Validation phase.** The first step for each node is to verify all concurrent blocks. Since consensus nodes do not execute transactions to generate the latest state root in this framework, we make a slight design change that each block contains the state root generated in the previous epoch. Thus, each node verifies whether the state root in each concurrent block corresponds to the sate of the previous epoch. If a block fails the verification, each node will consider this block invalid and discard it.
- **Concurrent execution phase.** Each node picks transactions that first appear in all verified blocks and simulates their executions concurrently and speculatively based on the latest state snapshot. The simulation results contain the sets of addresses and values read and written by each transaction for subsequent concurrency control.
- **Concurrency control phase.** According to the simulation results, each node explores conflicting relationships between concurrent transactions and then generates the transaction scheduling information for reconciling conflicts. Based on this information, each node derives a total commit order among concurrent transactions.
- **Commitment phase.** Lastly, each node commits transactions according to the commit order. Specifically, each node applies the write values of each transaction to an in-memory state, and the updated elements in the state are then flushed to the underlying database.

C. Problem Statement

The problem of our concern is the considerable conflicts that occur during concurrent transaction processing in DAG-based blockchains. We denote by p the probability of a conflict arising between any two transactions. The number of potential conflicts $\mathcal{C}$ occurring in N_e transactions is:

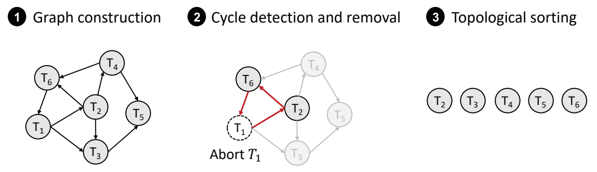
$$\mathcal{C} = \frac{N_e(N_e - 1)}{2} \cdot p \tag{1}$$

We observe that the number of conflicts $\mathcal{C}$ presents a power-law growth with N_e increases, which significantly enlarges the overhead of concurrency control to resolve conflicts.

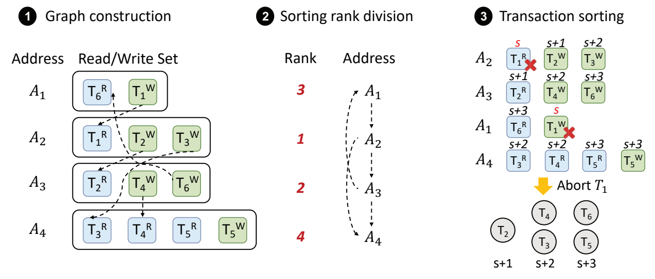
Main Goal. Our main goal is to devise an efficient concurrency control scheme to resolve $\mathcal{C}$ conflicts to derive a total serialization order among concurrent transactions with modest overhead, while yielding few transaction aborts.

D. Strawman Concurrency Control

This section describes a strawman concurrency control, which serves as the starting point for designing NEZHA. This approach relies on transaction dependencies that can indicate the serialization order between conflicting transactions. To capture transaction dependencies, we need to compare the read and written addresses between each pair of transactions. We



(a) Overview of strawman



(b) Overview of NEZHA

Fig. 3. A comparison of strawman and NEZHA

use $RS(T)$ and $WS(T)$ to denote the set of addresses read and written by a given transaction T, respectively. The specific definition of transaction dependency is as follows.

Definition 1. ***Transaction Dependency.*** *Given two transactions T_u and T_v ($u < v$), a transaction dependency $T_u \rightarrow T_v$ exists if the address read or written by T_u is also written by T_v. To be more specific, there are two types of transaction dependencies:*

- *$T_u \xrightarrow{rw} T_v$ (read-write dependency) exists if $RS(T_u) \cap WS(T_v) \neq \emptyset$.*
- *$T_u \xrightarrow{ww} T_v$ (write-write dependency) exists if $WS(T_u) \cap WS(T_v) \neq \emptyset$.*

Based on the captured transaction dependencies, we can build a directed graph called *conflict graph* (CG) that takes transactions as vertices and transaction dependencies as edges. CG can guide the ordering of transactions to reduce over-aborting transactions that are still serializable. The specification of CG is described by Definition 2.

Definition 2. ***Conflict Graph.*** *A conflict graph CG is a directed graph, which contains a set of vertices $V = \{T_1, T_2, ..., T_{N_e}\}$ and a set of edges $E = \{(T_u, T_v) | 1 \leq u \neq v \leq N_e, T_u \rightarrow T_v\}$, where $|V| = N_e$ and $|E| \geq \mathcal{C}$.*

The specific steps of concurrency control with CG is presented in Figure 3(a), which includes three steps: ❶ ***graph construction***, ❷ ***cycle detection and removal***, and ❸ ***topological sorting***. However, employing CG for concurrency control in DAG-based blockchains still suffers from efficiency issues in the three steps, making it hard to meet our main goal.

- ***Graph construction.*** In the graph construction step of CG, transaction dependencies need to be captured between each pair of transactions, which requires a cumbersome step of comparing the read and written addresses between them. The time complexity of the comparison step is $O(\frac{|V|^2-|V|}{2})$, which will cause enormous computational overhead in the case of considerable transactions.
- ***Cycle detection and removal.*** In CG, cycles will inevitably arise if there are unserializable transactions. Hence, cycle detection and removal is a necessary step before topological sorting to ensure that the graph is acyclic. Fabric++ [5] applies Johnson's algorithm [23] to discover all cycles within CG, which requires $O((|V| + |E|) \cdot (c + 1))$ (c is the number of cycles). However, the considerable conflicts may lead to a rise in the number of cycles, which will aggravate the overhead of this step in DAG-based blockchains.
- ***Topological sorting.*** Topological sorting on CG yields a serial order for transaction commitment. In the case of considerable transactions to be processed, the serial commit order will significantly increase the latency of transaction commitment. In essence, non-conflicting transactions can be committed concurrently to update the state. Thus, the transaction commitment concurrency should also be exploited to improve transaction processing efficiency.

IV. NEZHA CONCURRENCY CONTROL

A. Overview

To meet our main goal, we propose an efficient concurrency control scheme called NEZHA for DAG-based blockchains. The high-level overview of NEZHA is presented in Figure 3(b), which includes three steps: ❶ ***graph construction***, ❷ ***sorting rank division***, and ❸ ***transaction sorting***. First, NEZHA employs addresses to discover all dependent transactions and organizes them into a novel scheduling graph called *address-based conflict graph* (ACG). Next, NEZHA utilizes a *hierarchical sorting* (HS) algorithm to rank addresses based on address dependencies captured from ACG and then sorts transactions on each address successively to yield a total order. In what follows, we illustrate the design of NEZHA step by step.

B. Address-based Conflict Graph

We start by alleviating the overhead of detecting all dependent transactions. We discover that each address may exhibit more transaction dependencies due to the increased conflicts per address in DAG-based blockchains. We thus utilize addresses to capture transaction dependencies instead of capturing between each pair of transactions.

To be specific, for an address A_j, we employ a read/write set RW_j to store transactions that read and write to this address. For a given transaction T_v, we use more fine-grained read/write units T_v^R and T_v^W to better represent its read and write operations. Then, we map the read and write units of each transaction to the read/write sets of their corresponding addresses. Each write unit thus has a dependency relationship with other read and write units on the same address. To respect the read-write dependency order between transactions, we put all read units in front of write units in advance on each address.

TABLE III
ADDRESSES ACCESSED BY SIX TRANSACTIONS

Transaction	T_1	T_2	T_3	T_4	T_5	T_6
Read Address	A_2	A_3	A_4	A_4	A_4	A_1
Written Address	A_1	A_2	A_2	A_3	A_4	A_3

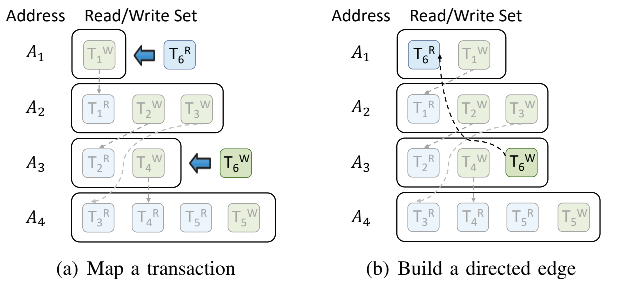
Since each address maintains all dependent transactions that read and write to it, we can obtain a partial order between transactions on each address. However, some transactions may read and write to multiple addresses, leading to a sequential relationship between sets of transactions on different addresses. Taking Figure 1 as an example, T_1 and T_2 precede T_3 on address A_1, while T_3 precedes T_4 on address A_2. Hence, T_1 and T_2 must precede T_4 in the total order. To obtain a total order among all transactions, we need to explore the sequential relationship between sets of transactions on different addresses. We define such a relationship as address dependency. The specification of address dependency is described by Definition 3.

Definition 3. ***Address Dependency.*** *Given two addresses A_i and A_j ($i \neq j$), A_i is dependent on A_j ($A_i \dashrightarrow A_j$) if there exists a transaction T_v whose $T_v^W \in RW_i \wedge T_v^R \in RW_j$.*

We capture the address dependency between write-read units of a transaction rather than between its write-write units or read-read units. The reason is that the relationship between write-read units of a transaction can better indicate the order among transactions on different addresses. Specifically, for a transaction T_v that reads address A_j and writes to address A_i, its write unit T_v^W tends to be at the back in RW_i while its read unit T_v^R tends to be at the front in RW_j. Thus, transactions before T_v^W on A_i generally precede transactions on A_j. To better capture the address dependency, we build a directed edge between write-read units of each transaction that reads and writes to different addresses. We employ these edges to organize the read/write sets of all addresses into a novel conflict graph called *address-based conflict graph* (ACG). The specific definition of ACG is as follows.

Definition 4. ***Address-based Conflict Graph.*** *An address-based conflict graph $ACG = (V, E)$ is a directed graph, where $V = \{RW_j | j = 1, 2, ..., n\}$, $E = \{(RW_i, RW_j) | 1 \leq i \neq j \leq n, \exists v \in [1, N_e], T_v^W \in RW_i \wedge T_v^R \in RW_j\}$. n denotes the number of accessed addresses.*

Let us go through a concrete example of ACG construction. Suppose there are six concurrent transactions T_1 to T_6, and the addresses accessed by them are shown in Table III. We sequentially map their read and write units to the corresponding addresses and place them in the correct positions of the read/write sets. Except for T_5, a directed edge is built between write-read units of each transaction. Figure 4 depicts an example of adding T_6. As a result, an ACG that contains read/write sets of four addresses A_1 to A_4 is constructed. To sum up, we eliminate the step of discovering dependencies between each pair of transactions and complete the construction of ACG in $O(u \cdot N_e)$ (u denotes the average number of read/write units of each transaction), thereby ensuring that detecting all dependent transactions can be done in linear time.



(a) Map a transaction (b) Build a directed edge

Fig. 4. Construction of ACG, taking adding T_6 as an example

C. Hierarchical Sorting

After building an ACG, the next step is to determine the specific order of each transaction. To generate a total commit order with a certain degree of concurrency through address dependencies, we need to devise a novel sorting algorithm based on ACG. Inspired by Lamport's logical clock [24], we can assign a unified sequence number for each read/write unit to represent their sequence in the total order. Due to the atomicity of transactions, all read and write units of a transaction should be assigned the same sequence number.

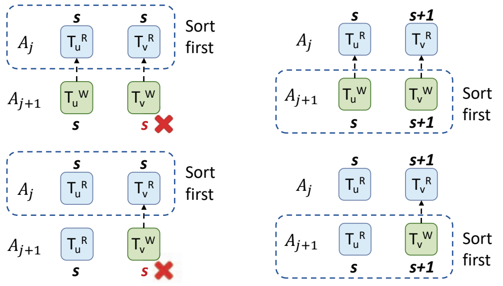
According to the order indicated by the read-write dependency ($\xrightarrow{rw}$) and the write-write dependency ($\xrightarrow{ww}$), we specify the following rules for assigning sequence numbers to read/write units on the same address:

- For any read and write units T_u^R, T_v^W ($u \neq v$), the sequence number of T_u^R is smaller than that of T_v^W.
- For any two write units T_u^W, T_v^W ($u < v$), the order between them is determined according to their subscripts (e.g., ids) to ensure deterministic sorting, i.e., T_u^W has a smaller sequence number than T_v^W.
- For any two read units T_u^R, T_v^R ($u \neq v$), they can be assigned the same sequence number since there is no conflict between read operations.

However, for addresses with dependencies, determining which of them the transactions are sorted first is an issue. We discover that some sorting priorities between addresses may incur anomalies that affect the validity of sorting results.

Sorting anomaly. Figure 5(a) depicts two possible anomalies when we first assign the sequence number s to the read units T_u^R and T_v^R on A_j. These two anomalies occur in the case that A_{j+1} is dependent on A_j. Since T_u and T_v have a read-write or write-write dependency on A_{j+1}, they cannot be assigned the same sequence number s. Instead, T_v should be assigned a larger sequence number (e.g., $s + 1$) in both cases.

As shown in Figure 5(b), if we first assign sequence numbers to the read/write units of T_u and T_v on A_{j+1}, the sorting anomaly will be resolved, i.e., T_u and T_v maintain the valid partial order. By analyzing the above sorting anomalies, we conclude that the sorting priorities of addresses should follow their dependency orders. The rationale is that determining the sequence of a transaction by its write units is more accurate



(a) Sorting anomalies (b) Change sorting priority

Fig. 5. Two sorting anomalies and solutions to them

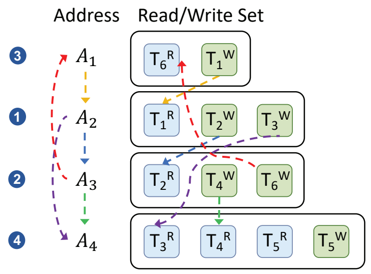


Fig. 6. Sorting ranks of addresses according to their dependencies (*the blue label denotes the sorting rank of each address*)

than by its read units. Therefore, we rank all addresses by address dependencies to indicate their sorting priorities.

Sorting rank division. Figure 6 depicts all address dependencies obtained according to the built edges in the ACG presented in Figure 4. These dependencies can form a graph with each address as a vertex. Thus, a topological sorting algorithm can be employed to generate a total order among addresses to indicate their sorting ranks. However, sometimes there may exist cycles in the graph, e.g., $A_1 \dashrightarrow A_2 \dashrightarrow A_3 \dashrightarrow A_1$. This phenomenon is caused by unserializable transactions, such as T_1 and T_6 in this example. Specifically, T_6 precedes T_1 on address A_1, whereas T_2 after T_1 precedes T_6 on address A_3.

Since we only need to determine the sorting ranks rather than resolving unserializable transactions to remove cycles, we take a trick that prioritizes the address with the most dependencies if cycles exist. The rationale is that, for the address with more dependencies, its transaction sorting result will affect the sorting of more addresses. The optimized topological sorting algorithm for ranking addresses is presented in Algorithm 1.

In the absence of cycles, addresses with zero in-degree are successively assigned sorting ranks, and their associated edges are deleted from the graph (lines 9-11 and line 23). Once there are no addresses with zero in-degree, i.e., cycles exist in the graph, we will look for the address with the minimum in-degree (lines 14-15). Suppose there are multiple addresses with the minimum in-degree, we will compare the number of their out-degrees and prioritize the one with the maximum out-degree (i.e., most dependencies) and the minimum address subscript (lines 20-21). Figure 6 also presents the sorting ranks of A_1 to A_4 after rank division. A_2 is assigned the highest sorting rank since it has two dependent addresses A_3 and A_4. A_3, which also has two dependent addresses, is assigned the second rank due to its larger subscript. A_1 and A_4 are assigned the third and fourth sorting ranks, respectively.

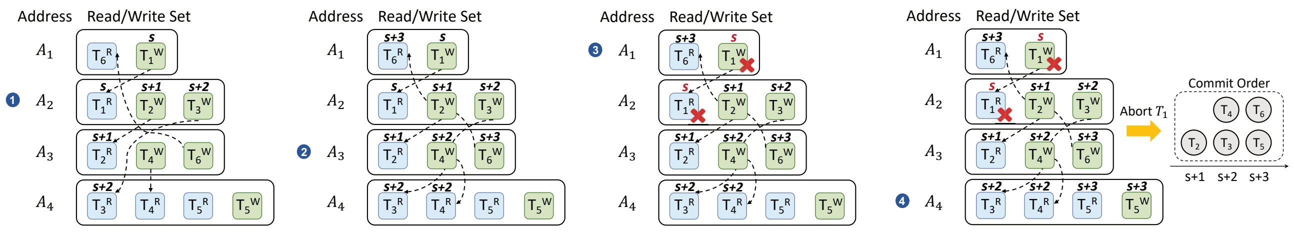
Algorithm 1 Sorting Rank Division

```
Input: $G$ is the graph constructed with each address $A_j$ as a vertex
Output: $seq$ is the sequence that represents sorting ranks
 1: $minAddrs \rightarrow$ the set containing the vertices with the minimum
          in-degree
 2: $selected \rightarrow$ the selected vertex in this round
 3: procedure AddressRank($G$)
 4:     if $G$.vertices $== \emptyset$ then return
 5:     end if
 6:     $min :=$ find the minimum in-degree in $G$
 7:     for $A_j$ in $G$ do
 8:         if $A_j$.inDegree $== min$ then
 9:             if $min == 0$ then
10:                 $selected = A_j$
11:                 Append($seq, selected$)
12:                 break
13:             else
14:                 Clear($minAddrs$)
15:                 Append($minAddrs, A_j$)
16:             end if
17:         end if
18:     end for
19:     if $min > 0$ then
20:         $selected =$ find the first address with the maximum
            out-degree in $minAddrs$
21:         Append($seq, selected$)
22:     end if
23:     $G' :=$ remove the vertex and edges of $selected$ from $G$
24:     AddressRank($G'$)
25: end procedure
```

Sorting on each address. After sorting rank division, we start to assign sequence numbers to the read/write units on each address in the order of their respective sorting rank. The transaction sorting algorithm on each address is presented in Algorithm 2. We first check if there exist read units that have been assigned sequence numbers. If not, all read units are assigned the same initial sequence number (lines 5-8). If so, we use the minimum sequence number among them to assign the remaining read units (lines 10-14).

We start to sort write units after all read units are sorted. First, we examine all write units that have assigned sequence numbers. If we find a write unit whose read unit is also stored on this address, according to the ordering rule between read-write units, we reassign this read and write unit a sequence number greater than the maximum one among read units (lines 17-19). To detect unserializable transactions, we only need to check whether the sequence number of a write unit is smaller than the maximum one among read units instead of tedious cycle detection. If so, we abort the transaction corresponding to this write unit (lines 20-24). After detection, the remaining write units are given increasing sequence numbers different from the existing ones (lines 25-35).

Let us continue with the previous example of the six transactions T_1 to T_6. Figure 7 depicts the detailed sorting process. According to the predefined sorting ranks, A_2 and A_3 are the first two addresses where read/write units will be sorted. After sorting read/write units on A_2 and A_3, the five transactions T_1



(a) Sort on A_2 (b) Sort on A_3 (c) Sort on A_1 (d) Sort on A_4

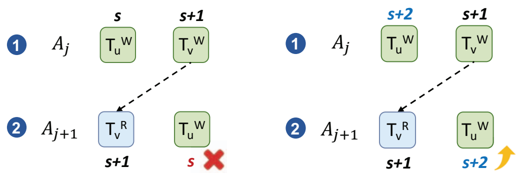
Fig. 7. Transaction sorting on each address

Algorithm 2 Transaction Sorting

```
Input: $initialSeq$ is the default initial sequence number
 1: $maxRead \rightarrow$ the maximum sequence number of read unit
 2: $writeSeq \rightarrow$ the sequence number of write unit
 3: $sortedRSet :=$ find read units with sequence numbers in $RW_j$
 4: if $sortedRSet$.length $== 0$ then
 5:     for $T_u^R$ in $RW_j$ do
 6:         $T_u$.sequence $= initialSeq$
 7:         $maxRead = initialSeq$
 8:     end for
 9: else
10:     $minSeq, maxSeq :=$ find the minimum and maximum
          sequence number in $sortedRSet$
11:     $maxRead = maxSeq$
12:     for remaining $T_u^R$ in $RW_j$ do
13:         $T_u$.sequence $= minSeq$
14:     end for
15: end if
16: $sortedWSet :=$ find write units with sequence numbers in $RW_j$
17: if $sortedWSet$ contains $T_v^W$ whose $T_v^R$ exists in $RW_j$ then
18:     $T_v$.sequence $= maxRead{+}{+}$
19: end if
20: for $T_u^W$ in $sortedWSet$ do
21:     if $T_u$.sequence $< maxRead$ then
22:         Abort $T_u$                        ▷ Unserializable transaction
23:     end if
24: end for
25: if $maxRead == 0$ then            ▷ No read units on this address
26:     $writeSeq = initialSeq$
27: else
28:     $writeSeq = maxRead + 1$
29: end if
30: for remaining $T_u^W$ in $RW_j$ do
31:     while $writeSeq$ is assigned do
32:         $writeSeq{+}{+}$
33:     end while
34:     $T_u$.sequence $= writeSeq$
35: end for
```

to T_4 and T_6 have been assigned sequence numbers of s to $s+3$. T_3 and T_4 maintain the same commit sequence (i.e., $s+2$) since their write operations do not conflict. During sorting on A_1, the aforementioned unserializable transactions T_6 and T_1 can be detected since the sequence number of T_1^W is smaller than T_6^R. We choose to abort T_1 with an abnormal sequence number to assure that our algorithm is deterministic. After read/write units on A_4 are sorted, we can obtain a total commit order with a certain degree of concurrency. As shown in Figure 7(d), transactions assigned the same sequence number can be committed concurrently.

D. Enhanced Design: Reordering



(a) Unserializable T_u^W (b) Reorder T_u^W

Fig. 8. Reordering design to resolve unserializable transactions

According to the reorderability theorem proposed in [6], some unserializable transactions caused by write-write dependencies can become serializable by switching their orders. This opens up an opportunity for us to further reduce transaction aborts during sorting. Figure 8(a) presents an unserializable anomaly in the case of sorting two consecutive write units with a write-write dependency. According to the address dependency, the write units of T_v and T_u on A_j are first assigned increasing sequence numbers. However, T_v and T_u also exhibit a read-write dependency on A_{j+1}. The sorting result on A_j causes T_u to be aborted since its write unit T_u^W has a smaller sequence number than T_v^R on A_{j+1}.

Unlike the stable order between read and write units, the order between write units can be switched. Hence, we can adjust the sequence numbers of unserializable transactions that write to multiple addresses. Considering that there may be other write units behind the unit to be adjusted, to avoid adjusting the sequence numbers of other write units, we can assign the write unit to be adjusted with a sequence number greater than the maximum one on the address. Taking Figure 8(b) as an example, when discovering the sequence number of T_u^W on A_{j+1} is abnormal, we check if T_u has another write unit on A_j. If so, we find the maximum assigned sequence number on A_j and A_{j+1} (i.e., $s+1$). We thus assign a larger sequence number $s+2$ to T_u to make it serializable.

V. Implementation

We implement the proposed NEZHA in Go[1]. For the underlying DAG-based blockchain system, we choose OHIE [2], a prevailing DAG-based blockchain system that employs multiple Nakamoto consensus instances to enable parallel block generation. We implement the prototype of OHIE in Go and build an execution layer on top of it to support system state updates. In the execution layer, we adopt the *Ethereum Virtual Machine* (EVM) to provide an execution environment for

[1]https://golang.org

TABLE IV
TRANSACTION PROCESSING LATENCY UNDER THE WORKLOAD WITH A UNIFORM DISTRIBUTION

Block concurrency	2	4	6	8	10	12
Serial latency (ms)	$4,700$	$10,900$	$17,200$	$23,800$	$30,000$	$36,600$
NEZHA latency (ms)	123.4 (e)	246.4 (e)	369.3 (e)	511.7 (e)	641.5 (e)	743.4 (e)
	22.1 (c)	32.8 (c)	44.9 (c)	56.4 (c)	71.6 (c)	87.1 (c)

*The latency of simulating transaction executions is denoted with "e", and the latency of concurrency control and transaction commitment is denoted with "c".

smart contracts written in Solidity[2]. We also implement an EVM-based read/write logger to record the addresses and values that each transaction reads and writes during simulation execution. To utilize CPU resources, we set up multiple worker threads to simulate transaction executions and commit transactions. In the data storage layer, we employ a fast key-value storage engine LevelDB[3] for storing block data and state data. Moreover, the *Merkle Patricia Trie* (MPT) is utilized to efficiently organize the state object of each account.

VI. PERFORMANCE EVALUATION

In this section, we demonstrate that NEZHA is effective and efficient for resolving considerable conflicts in DAG-based blockchain systems. By evaluating NEZHA, we aim to answer the following questions:

(1) What is the performance of NEZHA in concurrency control latency and overall transaction processing latency?
(2) How does NEZHA perform in transaction abort rate?
(3) How much throughput improvement does NEZHA bring to the DAG-based blockchain?

A. Experimental Setup

Our experiments are conducted on the Alibaba Cloud platform. We create a cluster of 14 nodes that are located within the same region and connected via 100 Mbps Ethernet. 12 nodes serve as miners to propose blocks in parallel, 1 node serves as a full node to synchronize the entire system state, and 1 node serves as a client to propose transactions. Each node consists of Intel Xeon (Ice Lake) Platinum 8369B CPU running at 3.5 GHz and runs Ubuntu 20.04 LTS as the operating system. The miner nodes and the full node are equipped with 16 vCPUs and 64 GB RAM, while the client node is equipped with 4 vCPUs and 16 GB RAM.

To simplify the experiments, we measure the performance of transaction processing and system throughput on the full node side. Regarding the underlying DAG-based blockchain OHIE, we set the maximum block concurrency as 12 (i.e., at most 12 parallel chains) and the block size as 200 transactions. Each experiment is run at least four times, and the average value is taken as the final result.

Compared Schemes. We compare NEZHA with two schemes:

- **Baseline scheme (Serial):** We choose the serial transaction processing scheme adopted by the current DAG-based blockchains as the baseline. For this baseline, we employ the EVM execution engine to execute and commit transactions in a serial manner.
- **Strawman scheme (CG):** The strawman scheme adopts CG for concurrency control, which is described in Section III. For a fair comparison, we implement the CG scheme by utilizing the same graph construction and processing algorithm as Fabric++ [5] and FabricSharp [6], in which Tarjan's algorithm [25] and Johnson's algorithm [23] are used for cycle detection and removal.

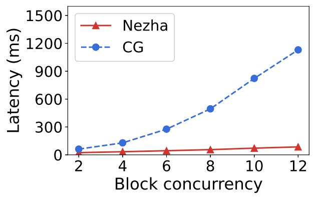
Workloads. We use the workloads based on the Smallbank benchmark, which is also adopted to evaluate Fabric++ [5] and FabricSharp [6]. A SmallBank contract written in Solidity consists of six types of transactions: `updateSavings`, `updateBalance`, `sendPayment`, `writeCheck`, `almagate`, and `getBalance`, where the first five transactions conduct write operations on user accounts and the last one only conducts read operation. During each call to the smart contract, we select one of these six transactions in a uniform distribution. We set the total number of accounts that transactions may access as 10k. The access skewness $skew$ follows a Zipfian distribution. The larger $skew$ is, the more read/write hot accounts among 10k accounts, indicating an increase in potential conflicts. A $skew$ value of 0 indicates that the account access follows a uniform distribution.

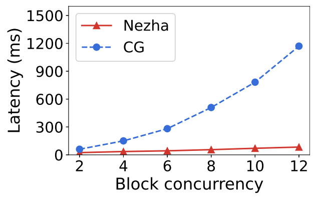
B. Transaction Processing Latency

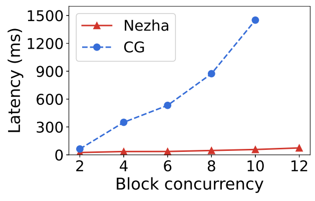
In this series of experiments, we first compare NEZHA with **Serial** to evaluate overall transaction processing latency. For **Serial**, we measure the overall latency of serial transaction execution and commitment. For NEZHA, we measure the overall latency of concurrent simulation, concurrency control, and transaction commitment. Table IV presents the results under $skew = 0$. As expected, the inefficiency issue of **Serial** adopting the EVM execution engine is magnified under high transaction throughput. In contrast, even under large block concurrency, the overall latency of NEZHA remains within 1 s, where the latency occupied by concurrency control is only a tiny part. In general, NEZHA can improve the transaction processing latency up to 40$\times$ speedup over the baseline.

In the comparison of NEZHA and **CG**, we only evaluate the latency of concurrency control and transaction commitment, since NEZHA and **CG** adopt the same simulation mechanism. We measure their latencies under four $skew$ values in steps of 0.2. As depicted in Figure 9, in the four cases, the latency growth rate of **CG** is much faster than NEZHA as block concurrency increases. Especially in the cases of $skew =$ 0.6 and 0.8, the latency of **CG** exhibits a more dramatic

[2]https://soliditylang.org
[3]https://github.com/google/leveldb


(a) $skew$ = 0.2


(b) $skew$ = 0.4


(c) $skew$ = 0.6

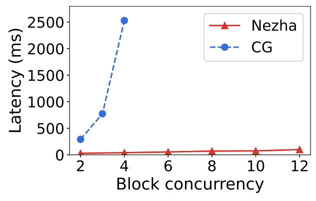

(d) $skew$ = 0.8

Fig. 9. Overall latency of concurrency control and transaction commitment under varying block concurrency

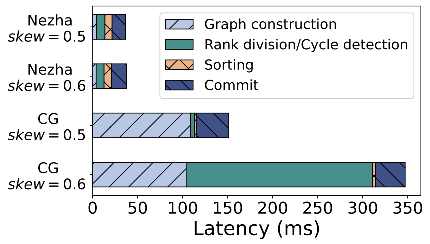


Fig. 10. Latency of each sub-phase in concurrency control

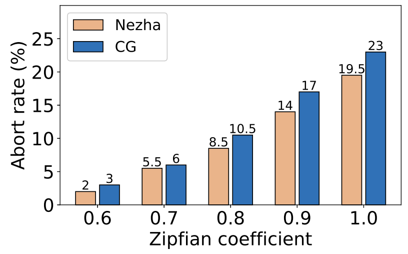


Fig. 11. Transaction abort rate under varying Zipfian coefficient $skew$

growth. When the block concurrency reaches 12 in the case of $skew = 0.6$, the latency of **CG** exceeds 10 s. What's worse, in the case of $skew = 0.8$, the **CG** process fails due to being out of memory when the block concurrency exceeds 4. On the contrary, even under high data contention, the latency of NEZHA is less than 100 ms. An interesting observation is that the latency of NEZHA fluctuates slightly with the increase of block concurrency and the value of $skew$. This is because the rise in conflicts leads to a decrease in the number of accessed addresses. The latency of sorting rank division of addresses thus slightly decreases. In general, NEZHA can speed up the concurrent transaction processing to 10× at most over **CG**.

To better understand the advantages of NEZHA in concurrency control, we measure the latency of each sub-phase in concurrency control. In this experiment, we fix the block concurrency to 4 and choose two $skew$ values of 0.5 and 0.6. As presented in Figure 10, we can observe that the graph construction latency of **CG** occupies the most when $skew = 0.5$. Although the adopted graph construction algorithm reduces the squared time complexity of $O(\frac{|V|^2-|V|}{2})$, the overhead of building an edge between each pair of dependent transactions is inevitable. As $skew$ rises to 0.6, the latency of cycle detection and removal occupies the most since the recursive Johnson's algorithm has a complexity of $O((|V|+|E|)\cdot(C+1))$, which takes up a considerable amount of memory when the number of cycles is large. This can also explain why the latency of **CG** in Figure 9(c) and 9(d) increases sharply. By contrast, due to our novel ACG and HS algorithm without cycle detection, the graph construction overhead of NEZHA is negligible, and the sorting latency remains stable as $skew$ increases. Besides, NEZHA is also superior to **CG** in the latency of transaction commitment, which benefits from a commit order with high concurrency.

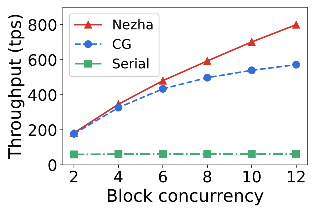
C. Transaction Abort Rate

In this experiment, we evaluate the transaction abort rate of NEZHA and **CG** under high data contention. Since **CG** is prone to failure due to being out of memory under large block concurrency, we choose to conduct this comparison experiment under a block concurrency of 1. Figure 11 depicts the comparison results. In the cases of $skew = 0.6$ and 0.7, both NEZHA and **CG** maintain a low transaction abort rate, and there is no significant gap between them. As the value of $skew$ continues to rise, the transaction abort rates of NEZHA and **CG** increase greatly. This is due to the increase in unserializable transactions. However, the gap between NEZHA and **CG** is also enlarged, e.g., the transaction abort rate of NEZHA is 3.5% lower than **CG** when $skew = 1.0$. Such superiority benefits from our enhanced design, which can resolve unserializable transactions caused by write-write dependencies.

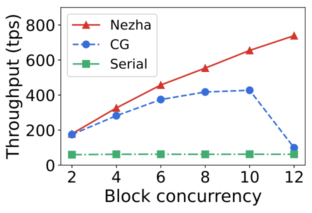
D. Effective Throughput

In the last two experiments, we evaluate the effective system throughput of OHIE with NEZHA and comparison schemes. Note that the effective throughput here represents the number of valid transactions that pass transaction processing and persist their states, not merely the consensus throughput. We set the expected block generation latency as 1 second. Figure 12 presents the experimental results. From Figure 12(a) and 12(b), we can observe that the inefficiency issue of **Serial** seriously affects the growth of system throughput. Even under large block concurrency, the system throughput still remains around 60 tps (*Transactions Per Second*). On the contrary, enabling concurrent transaction processing can significantly improve the system throughput. For **CG**, in the case of $skew =$ 0.2, the effective throughput increases as block concurrency rises, whereas the growth rate decreases due to the rise in concurrency control latency and transaction abort rate. In the case of $skew = 0.6$, the effective throughput drastically drops at the block concurrency of 12. This phenomenon is caused by a sudden increase in the concurrency control latency of **CG**, which corresponds to Figure 9(c).

In contrast, although the effective throughput of NEZHA does not outperform **CG** much when the block concurrency



(a) $skew$ = 0.2



(b) $skew$ = 0.6

Fig. 12. Effective system throughput under varying block concurrency

is small, its throughput increases almost linearly as block concurrency rises. Hence, NEZHA will have a greater advantage over **CG** in throughput when the block concurrency is large. Moreover, compared with the case of $skew = 0.2$, the effective throughput of NEZHA does not drop much when $skew = 0.6$. The root cause is that, although the number of aborted transactions increases as the value of $skew$ rises, the concurrency control latency does not vary a lot.

To sum up, the above series of experiments demonstrate that NEZHA can enable efficient concurrency control and yield few transaction aborts in the case of considerable transactions and conflicts, which is more suitable for DAG-based blockchains than the compared schemes.

VII. CONCLUSION

This paper presents an efficient concurrency control scheme NEZHA towards DAG-based blockchains, which can detect and reconcile considerable conflicts with modest overhead. NEZHA achieves this by exploiting address dependency to construct a novel *address-based conflict graph* (ACG) for discovering dependent transactions in an efficient way. Based on ACG, NEZHA introduces a *hierarchical sorting* (HS) algorithm to efficiently generate a total commit order with high concurrency. The experimental results demonstrate that NEZHA outperforms conventional schemes and shows stable performance with increasing conflicts.

ACKNOWLEDGMENT

This work was supported by National Key Research and Development Program of China under Grant No. 2021YFB2700700, Key Research and Development Program of Hubei Province No. 2021BEA164, National Natural Science Foundation of China (Grant No. 62072197), RGC RIF grant R6021-20, and RGC GRF grants under the contracts 16209120 and 16200221. Jiang Xiao is the corresponding author of the paper.

REFERENCES

[1] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, and A. C.-C. Yao, "A decentralized blockchain with high throughput and fast confirmation," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 515–528.
[2] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "Ohie: Blockchain scaling made simple," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 90–105.
[3] D. Reijsbergen and T. T. A. Dinh, "On exploiting transaction concurrency to speed up blockchains," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 1044–1054.
[4] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," *IEEE Transactions on Knowledge and Data Engineering*, 2021.
[5] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the International Conference on Management of Data (ICDE)*, 2019, pp. 105–122.
[6] P. Ruan, D. Loghin, Q.-T. Ta, M. Zhang, G. Chen, and B. C. Ooi, "A transactional perspective on execute-order-validate blockchains," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2020, pp. 543–557.
[7] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2018, pp. 1–15.
[8] J. A. Chacko, R. Mayer, and H.-A. Jacobsen, "Why do my blockchain transactions fail? a study of hyperledger fabric," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2021, pp. 221–234.
[9] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981.
[10] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1337–1347.
[11] Z. Chen, X. Qi, X. Du, Z. Zhang, and C. Jin, "Peep: A parallel execution engine for permissioned blockchain systems," in *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, 2021, pp. 341–357.
[12] C. Xu, C. Zhang, J. Xu, and J. Pei, "Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing," in *Proceedings of the VLDB Endowment*, vol. 14, no. 11, 2021, pp. 2314–2326.
[13] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2017, pp. 303–312.
[14] Y. Li, H. Liu, Y. Chen, J. Gao, Z. Wu, Z. Guan, and Z. Chen, "Fastblock: Accelerating blockchains via hardware transactional memory," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 250–260.
[15] S. Popov, "The tangle," 2018. [Online]. Available: http://www.descryptions.com/Iota.pdf
[16] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: A fast and scalable cryptocurrency protocol," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 1159, 2016.
[17] Q. Wang, J. Yu, S. Chen, and Y. Xiang, "Sok: Diving into dag-based blockchain systems," *arXiv preprint arXiv:2012.06128*, 2020.
[18] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 585–602.
[19] J. Xu, Y. Cheng, C. Wang, and X. Jia, "Occam: A secure and adaptive scaling scheme for permissionless blockchain," in *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 618–628.
[20] L. Baird, "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance," *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep*, 2016.
[21] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is dag," in *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, 2021, pp. 165–175.
[22] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999, pp. 173–186.
[23] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.
[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*. ACM, 2019, pp. 179–196.
[25] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.