This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering

A High Performance Concurrency Protocol for Smart Contracts of Permissioned Blockchain

Cheqing Jin[1,2], Shuaifeng Pang[1], Xiaodong Qi[1], Zhao Zhang[1,2,3✉], Aoying Zhou[1]
[1]*School of Data Science and Engineering, East China Normal University*
[2]*Shanghai Engineering Research Center of Big Data Management, East China Normal University*
[3]*Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology*
cqjin@dase.ecnu.edu.cn, {sfpang, xdqi}@stu.ecnu.edu.cn, {zhzhang, ayzhou}@dase.ecnu.edu.cn

Abstract—Although the emergence of the programmable smart contract makes blockchain systems easily embrace a wide range of industrial services, how to execute smart contracts efficiently becomes a big challenge nowadays. Due to the existence of Byzantine nodes, existing mature concurrency control protocols in database cannot be employed directly, since the mechanism of executing smart contracts varies a lot. Furthermore, even though smart contract execution follows a two-phase style, i.e., the primary node executes a batch of smart contracts in the first phase and the validators replay them in the second phase, existing parallel solutions merely focus on the optimization for the first phase, rather than the second phase. In this paper, we propose a novel two-phase concurrency control protocol to optimize both phases for the first time. First, the primary executes transactions in parallel and generates a transaction dependency graph with high parallelism for validators. Then, a graph partition algorithm is devised to divide the original graph into several sub-graphs to preserve parallelism and reduce communication cost remarkably. Finally, we propose a deterministic replay protocol to re-execute the primary's parallel schedule concurrently. Moreover, this two-phase protocol is further optimized by integrating with PBFT. Theoretical analysis and extensive experimental results illustrate that the proposed scheme outperforms state-of-art solutions significantly.

Index Terms—Blockchain, Smart Contract, Concurrency

1 Introduction

As a kind of distributed ledger shared by many non-trusted parties, blockchain technology has gained lots of attention and interest from public [1], [2] and academic communities [3], [4]. Modern blockchain systems introduce programmable smart contracts that use functions and state variables to describe complex business logic. Client requests are wrapped in transactions to invoke functions of smart contracts. In both permissionless and permissioned blockchain [5], smart contract execution follows a similar manner, during which a primary node (also called *miner*) firstly assembles and executes a batch of smart contract transactions in sequence, and then the rest (called *validators*) re-execute them serially in the same order to keep state consistent. Such design limits overall throughput and no concurrency is allowed during execution.

An interesting and natural observation is that the increment of consensus throughput will finally make the serial smart contract execution a real bottleneck, especially in permissioned blockchain systems. Permissionless blockchain tends to select PoW (Proof-of-Work) as its consensus algorithm for consideration of kinds of attacks which is low in throughput so that serial execution is not likely a bottleneck. As reported in some latest literature, state-of-the-art consensus algorithms can process up to thousands of transactions per second [6]–[8]. For example, Quorum, a permissioned implementation of Ethereum that runs Byzantine fault tolerance (BFT) consensus algorithm [9], is reported to reach 2000+ tps [10]. A series of experiments upon real-world smart contract transactions from Ethereum network (Table 1 gives the results) show that the throughput of serial execution engine is around 100 tps, far below that of state-of-the-art consensus algorithms. Hence, it is crucial to devise a concurrent mechanism to improve smart contract execution performance of permissioned blockchain systems.

TABLE 1: Serial execution time over real-world transactions

# of transactions	50	100	150	200	250	300
Execution time (s)	0.44	0.93	1.30	2.16	2.69	3.27
TPS	113	107	115	93	93	92

Since transactions in a common blockchain system must be executed by the primary and replayed by all validators, we follow a *two-phase* framework adopted in recent works [11]–[13], as shown in Figure 1. During the first phase (*proposal* phase), the primary collects and executes a batch of transactions concurrently and generates a serializable schedule. All transactions and supplementary scheduling information (e.g., the conflicts) recorded by the primary are packed into a block. Subsequently, the newly-generated block is appended to the blockchain network. The primary and all validators will arrive at a consensus over the latest block by running the consensus algorithm. During the second execution phase (*validation* phase), each validator replays all transactions deterministically based on the scheduling information provided by the primary. Again, such two-phase framework cannot be applied to permissionless blockchain systems since the primary in these networks has no interest in wasting resource on optimizing the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering

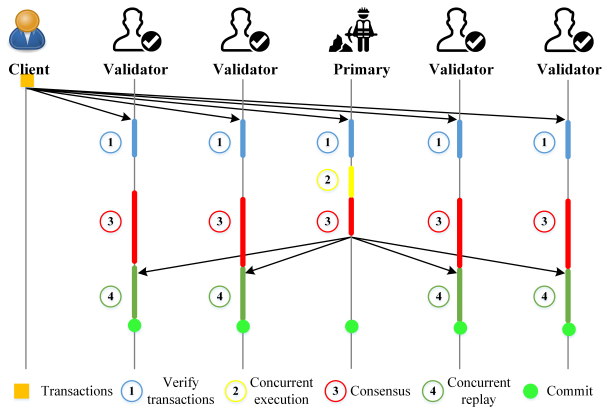


Fig. 1: The general two-phase execution framework

replay efficiency.

Existing works transit mature concurrency control protocol in database to blockchain and generate scheduling information with different granularity. Dickerson et al. used abstract locks and inverse logs in the primary phase to discover a serializable schedule [12]. Scheduling information is described as a simple directed acyclic graph (called happen-before graph) to help validators recognize transactions without conflicts and execute them concurrently with for-join [14] method. Anjana et al. replaced permissive locks with OCC (Optimistic Concurrency Control) and adopted a concise graph structure to capture the conflicts [11]. Zhang et al. can employ any protocols to produce a conflict-serializable schedule in the primary phase [13]. A finer-grained structure that records write sets of transactions predetermines all conflict data items. Transactions can be re-executed concurrently with a small number of conflicts.

Such related works consider two phases separately, i.e., they simply adopt mature concurrency control protocols to discover a conflict-serializable schedule and ignore some key properties of scheduling information, e.g., parallelism of directed acyclic graph. For instance, in [12], the primary may discover a serializable order in which every transaction has conflicts with its former transaction, i.e., generating a chain graph. In such a case, validators replay transactions without concurrency. Hence, besides serializability, the concurrency protocol devised for the primary needs to consider the parallelism property of scheduling information.

Another issue with these works lie in the granularity of scheduling information. In general, existing concurrent works are assorted into two classes according to the way of describing the conflicts among transactions. The mechanisms in one category record conflict information with fine-grained representations so that all transactions can be replayed independently. The MVTO protocol proposed by Zhang et al. that requires all write sets transferred to the validators suffers from huge communication overhead. The other category that adopts a concise graph structure to capture the conflicts, though, sharply reduces the communication cost, is expensive in replaying each transaction due to the overhead on monitoring the status of conflict transactions. Consequently, it is critical to achieve the balance between replay efficiency and network overhead.

In view of aforementioned challenges, we conceive a novel two-phase concurrency control protocol for smart contract execution, aiming at boosting performance for the primary and validators simultaneously. Unlike other related works which commit transactions in a random order, our approach takes advantage of the batching feature of blockchain and selects an optimal commit order that maximizes the parallelism of the scheduling information represented as a directed acyclic graph (transaction dependency graph). Since this fine-grained graph preserves conflict information and multiple versions of state variables which will bring huge communication cost, we divide the original graph into several small sub-graphs to lower the overall communication cost significantly. A deterministic replay protocol based on partitioned sub-graph is proposed for validators to guarantee the same serial order with the primary.

The major contributions are listed below.

- We design a novel two-phase concurrency protocol aiming at improving execution efficiency of the primary and the validators at the same time. The primary utilizes a batching OCC protocol and selects abort transactions carefully to achieve two goals simultaneously.
- A graph partition algorithm is devised to divide the original transaction dependency graph into small sub-graphs. This kind of medium-granular scheduling information preserves parallelism well and further reduce the communication cost. In the second phase, our presented deterministic protocol is capable of replaying all smart contracts efficiently on multi-core validators.
- We implement a multi-threaded prototype for the two-phase protocol and conduct thorough evaluations of both execution phases. The features of PBFT, including three rounds of communications and one-third fault tolerance rate, are considered to make further optimization. We implement the optimized protocol and the original one in open-source BFT-SMaRt and conduct experiments in a distributed setting.

Organization. The rest of paper is organized as follows. Section 2 reviews the latest works about concurrent smart contract execution and other concerned techniques. Section 3 formally defines the problem. In Section 4, we explain our concurrent two-phase execution scheme in detail. Section 5 analyzes the overall cost of the proposed protocol. The experimental evaluations are reported in Section 6. Section 7 concludes this paper in brief.

2 Related Work

We review latest research works close to our work in this section.

Concurrency control protocols. Concurrency control in DBMS [15] has been actively studied for more than 40 years. Generally, these works are classified into two kinds, pessimistic and optimistic. The most frequently used pessimistic scheme to ensure serializability is two-phase locking (2PL) [16]. In contrast, Kung et al. propose a validation-based, non-locking optimistic concurrency control scheme,

or in short, OCC [17], where each transaction goes through three phases: (i) During the read phase, transactions read data items directly from the storage and write to private locations. (ii) Transactions subsequently enter the validation phase to check if they have conflicts with previously committed transactions, e.g., *stale read*. (iii) In the final phase, write phase, transactions will install updates once they pass the validation. Wang et al. introduce an adaptive concurrency protocol which combines with both OCC and lock [18]. Ding and Kot utilize transaction batching and reordering to optimize OCC [19]. Ansari et al. improve transaction reordering dynamically by presenting a *steal-on-abort* technique [20]. Although such concurrent control protocols are commonly used to execute smart contract transactions efficiently, these works have no throughput optimization or determinism guarantee for the validators to replay transactions.

Deterministic replication. Determinism is also a concerned topic in concurrent execution. Thomson et al. use ordered locking to run a batch of transactions in a deterministic manner [21]. Recent deterministic system implementations [22] adopt a dependency graph generated from the transactional input log to avoid centralized processing. In both works, the accessed data sets of transactions must be known in advance. OLLP technique proposed by [21] determines transaction's full read/write sets by modifying transaction code and invoking a reconnaissance query that performs necessary reads. Such method incurs additional cost for transaction processing, and invalid "reconnoitered" read/write sets may cause transactions to abort, which further increases the latency. Moreover, these deterministic protocols are not designed for Byzantine environment so that they cannot be applied to blockchain systems.

Concurrent smart contract execution. There exist research works on adding concurrency to smart contract execution. Dickerson et al. firstly present a two-phase solution, permitting the primary and validators to execute smart contracts concurrently by treating every invocation as a speculative action [12]. Anjana et al. replace the lock-based STM adopted in [12] with an optimistic one and apply a decentralized mechanism in the *validation* phase [11]. Zhang et al. allow employing any concurrency control mechanism that produces a conflict-serializable schedule in the *proposal* phase [13]. Validators use MVTO protocol with the help of write sets provided by the primary to re-execute transactions. Sergey and Hobor explore the similarity between multi-transactional behaviors of smart contracts in Ethereum and shared-memory concurrency problem [23]. These approaches consider the problem separately while ours takes the overall interest of the primary and validators into account at the same time.

Sharma et al. [24] transit well-understood database concepts, namely transaction reordering and early transaction abort to Hyperledger Fabric 1.2. Although it outperforms Fabric up to a factor of 12x for tps, the proposed approach is tightly coupled with the consensus process of Fabric 1.2, which uses centralized orderer based on Apache Kafka and is not general for mainstream blockchain systems.

```
contract SimpleBank{
    address owner;
    mapping(address => uint) public accounts;
    function transfer(address addr1,address addr2,
                      uint amount){
        if(accounts[addr1] >= amount){
            accounts[addr1] = accounts[addr1] - amount;
            accounts[addr2] = accounts[addr2] + amount;
            return true;
        }
        return false;
    }
    // more functions defined
    // deposit, withdrawal, amalgamate...
}
```

Fig. 2: A simple smart contract reflecting banking application

3 Problem Statement

We formalize key issues in this section. Important notations are listed in Table 2. For simplicity, smart contract transaction is abbreviated as transaction hereafter.
缩写

TABLE 2: Important notations used in this paper

Symbol	Description	Symbol	Description
G	graph	P	partition of G
$\|V\|$	# of vertices	$\|E\|$	# of edges
$\omega(v)$	weight of v	$\rho(G)$	density of G
R_j^i	consistent read set	$G[V_i]$	subgraph consists of V_i
$RS(T_i)$	read set of T_i	$WS(T_i)$	write set of T_i
τ	workload threashold	$\mathcal{L}$	serialization order
$\omega(e)$	weight of e		

3.1 Representing the conflicts among transactions

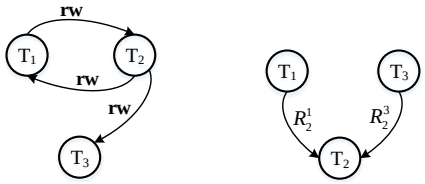
Since data race induced by concurrent transaction execution will lead to data inconsistency, conflicts must be resolved during runtime. Conflict graph (CG) [1] [25] [26] has long been adopted in concurrency control to capture conflicts among transactions, in which vertices are transactions, and edges stand for *read-write* conflict dependencies. Note that *write-write* conflict dependencies need not to be tracked in CG because each transaction maintains its own write set in OCC protocol.

Definition 1 (Conflict Graph). *A conflict graph $CG(V, E)$ is a directed graph, where $V = \{T_1, T_2, \ldots, T_n\}$, $E = \{(T_i, T_j) | i \neq j, RS(T_i) \cap WS(T_j) \neq \emptyset\}$, and $RS(T)$ and $WS(T)$ denote read set and write set respectively.*

Figure 2 shows a snippet of smart contract expressed in Solidity language [27] that implements a banking application. Initially, two state variables are declared: *owner* represents the customer who creates this contract, *accounts*

1. Conflict graph is also called serialization graph in some literature

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering



(a) CG (b) TDG

Fig. 3: An example of CG and TDG

maps addresses to their balances accordingly (at Lines 2-3). Function *transfer* that wires money from one account to another (at Lines 4-12).

Suppose there are three concurrent transactions calling the functions defined in the banking smart contract. T_1 tries to make a deposit into the account of Alice. T_2 is to move all the funds from Bob to Alice. T_3 wants to make a withdrawal on Bob. The read/write operations of such transactions are abstracted below.

$$T_1 : \mathsf{r}(Alice)\mathsf{w}(Alice)$$
$$T_2 : \mathsf{r}(Bob)\mathsf{r}(Alice)\mathsf{w}(Alice)$$
$$T_3 : \mathsf{r}(Bob)\mathsf{w}(Bob)$$

By Definition 1, we check the read sets and write sets between any two transactions and construct a conflict graph shown in Figure 3(a). As usual in traditional concurrency control theory, the absence of a cycle in CG proves that the graph is serializable. If CG is acyclic, a serialization order $\mathcal{L}$ can be acquired by repeatedly committing a transaction without any incoming edge using a topological order. Otherwise, we need to abort several transactions to make the graph acyclic.

However, since CG cannot capture data dependencies among those aborted transactions and committed transactions, we define a transaction dependency graph (TDG) to represent the final schedule of a batch B.

Definition 2 (Transaction Dependency Graph)**.** *A transaction dependency graph $TDG(V, E)$ is a directed acyclic graph, where $V = \{T_1, T_2, \ldots, T_n\}$, $E = \{(T_i, T_j, R_j^i) | 1 \leq i \neq j \leq n\}$. R_j^i, namely consistent read set, keeps all values that T_j reads from T_i.*

An example TDG is presented in Figure 3(b). If aborting T_2, we can commit the remaining two transactions in $\mathcal{L} : T_1 \rightarrow T_3$. Subsequently, if T_2 is re-executed, two dependency edges will be included in the final TDG with two consistent read sets $R_2^1 = \{Alice\}$ and $R_2^3 = \{Bob\}$ respectively. The weight of vertex T_i, denoted as $\omega(T_i)$, indicates the execution time of T_i. Hereafter, we use the terms vertex and transaction interchangeably. The edges in TDG correspond to both the precedence constrains and communication messages. The weight of an edge $e = (T_i, T_j)$, denoted as $\omega(e)$, suggests the communication overhead.

3.2 Problem definition

Since OCC determines the committing order during the validation phase, transactions that violate serializability need to be aborted. Removing vertices from a directed graph to make it cycle-free is actually classic feedback vertex set problem (FVS) [28]. Note that finding the smallest feedback vertex set is one of the basic requirements for FVS problem.

(a) Min-Density TDG (b) Partition TDG

Fig. 4: Two example solutions for Def. 3 and Def. 4 respectively

In this study, a more complex objective function is introduced based on the FVS problem, to make the transformed TDG with lower density for the sake of better replaying performance for validators. Equation 1 defines the density property of graph, $\rho(G)$, by the ratio of the number of edges to the maximal possible number of edges.

A graph with small degree maximizes the parallelism due to fewer inter-conflicts among transactions [29]. Hence, $\rho(G)$ describes the parallelism property well.

$$\rho(G) = \frac{|G.E|}{|G.V|(|G.V| - 1)} \tag{1}$$

Generating TDG with high degree of parallelism is converted to the FVS problem, as defined in Definition 3.

Definition 3 (Min-Density FVS Problem)**.** *Given a conflict graph CG after executing a batch of transactions, let $\mathbb{V}$ denote all feedback vertex sets whose removal makes CG serializable, i.e., deleting any vertex set $V' \in \mathbb{V}$ and corresponding edge set $E' = \{(u, v) | (u, v) \in CG.E \wedge (u \in V' \vee v \in V')\}$ makes CG acyclic. Min-Density FVS problem intends to find a feedback vertex set that minimizes the density of transformed TDG.*

Example 1. *In Figure 3(a), both $\{T_1\}$ and $\{T_2\}$ form solutions of FVS problem since the removal of any one makes the graph acyclic. However, aborting T_1 results in a TDG with lower density shown in Figure 4(a) ($\mathcal{D} = 1/6$). Hence, $\{T_1\}$ is the solution for our Min-Density FVS problem.*

According to the Def. 3, the solution to Min-Density FVS problem is suit for original FVS problem, thus the Min-Density FVS problem is a NP-hard as well. Otherwise, we can find a solution to FVS problem through a easier algorithm, which is impossible.

Generating a medium-grained schedule log (represented by TDG) is essential to improve replaying efficiency in the validation phase due to two reasons: (i) It is nearly infeasible to reach optimal result in time due to limited physical cores. (ii) Offering all consistent read sets occasions enormous network overhead. Therefore, it's vital to partition TDG to diminish communication cost while preserving the parallelism as much as possible.

Definition 4 (Graph partition problem)**.** *Let P denote all balanced partitions of TDG, i.e., $\forall\ p \in P$, $p = \{V_1, V_2, \ldots\}$, satisfies $\forall G[V_i], \omega(G[V_i]) \leq \tau$. This problem intends to compute a partition $\hat{p} \in P$ that meet $\forall p \in P, \omega(E_{\hat{p}}) \leq \omega(E_p)$, where $\omega(E_p)$ indicates the weight of all cut edges under partition p.*

The τ-constrained partitioning of $TDG(V, E)$ divides V into multiple disjoint subsets $\{V_1, V_2, ...\}$ with the workload of each subgraph no greater than threshold τ. The weight of a subgraph that consists of vertex set V_i, $\omega(G[V_i]) = \sum_{v \in G[V_i].V} \omega(v)$, represents the sum of vertex weights. An

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering

edge (T_i, T_j, R_j^i) is called *cut edge* if $T_i \in G[V_p], T_j \in G[V_q]$ and $p \neq q$. Let $\omega(E_p)$ denote the weight of all *cut edges* under a certain partition p, $\omega(E_p) = \sum_{e \in E_p} \omega(e)$. Definition 4 gives a constrain on each subgraph and accompanies the problem with an objective function based on $\omega(E_p)$.

Example 2. *Figure 4(b) illustrates a partition TDG which cuts the Min-Density TDG into two subgraphs and eliminates the network overhead for transferring R_1^2.*

With a sparser TDG, the weight $\omega(E_p)$ of all cut edges for TDG partition should be smaller, thus the communication cost is lower. Therefore, if the communication is bounded, we can say sparser TDG can achieve higher parallelism.

4 Execution Framework

The two-phase execution framework is an appropriate approach to keep state consistent among all nodes in a Byzantine environment. Existing research works consider this two-phase solution separately, where the primary adopts one of the mature and proven concurrency control protocols while neglecting the replay efficiency in validators.

Regarding this drawback, a novel protocol is proposed, taking the optimization of both phases into account at the same time. To be more specific, our approach is guided by three goals, including contriving a concurrency control protocol suitable for blockchain, determining the appropriate granularity for the schedule log, and devising a deterministic and efficient replay protocol based on the schedule log for validators.

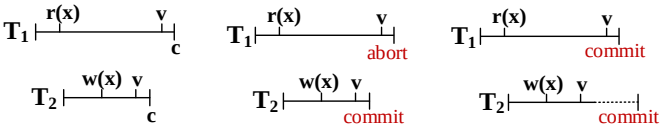
We propose a variant of OCC protocol, aiming at boosting performance for the primary and improving replay efficiency for validators, to fulfill the first goal. Seeking a solution to meet the first goal is exactly the same as the Min-Density FVS problem (Def. 3). More details about the variant protocol are described in Section 4.1. A practical algorithm is devised to lessen the network overhead while preserving the maximum degree of concurrency during replaying. The key of the second goal is a graph partition problem defined by Def. 4. Section 4.1 details the partition algorithm. As for the third goal, we propose a deterministic OCC protocol benefiting from **partitioned TDG**. We describe the concurrent validator scheme in Section 4.3.

4.1 Concurrency control protocol in the proposal phase

Batching and reordering transactions

Each block is essentially a batch of transactions. We take advantage of the batching feature and apply transaction reordering to the native OCC. As the final serialization order of transactions is only decided during the validation phase of OCC, we employ reordering to discover an order with the least abort. The protocol waits until all transactions in a batch complete their read phase, and selects an optimal validation order to reduce the number of conflicts and aborts. Moreover, reordering can also improve the throughput of the primary.

Figure 5 gives a simple transaction reordering example. Suppose two transactions T_1 and T_2 in a batch are executed concurrently. T_1 reads x before T_2 writes x. If T_2 enters the validation phase of OCC ahead of T_1, it can successfully commit but T_1 fails validation because of stale read, as shown in Figure 5(b). But with reordering, T_2 can be serialized after T_1 (Figure 5(c)). Thus, two transactions can both commit successfully.



(a) Two Txs (b) Original OCC (c) Reordering

Fig. 5: Reordering transactions to reduce abort

The goal of Min-Density FVS problem is to find a minimal subset B' of all vertices whose removal minimizes $\mathcal{D}$ of the output transaction dependency graph. Unfortunately, since the FVS problem is known as NP-hard [30], we propose a greedy algorithm for finding B' next.

Algorithm 1: Concurrent Proposal Phase

Input: A batch of transactions B
Output: A transaction dependency graph TDG

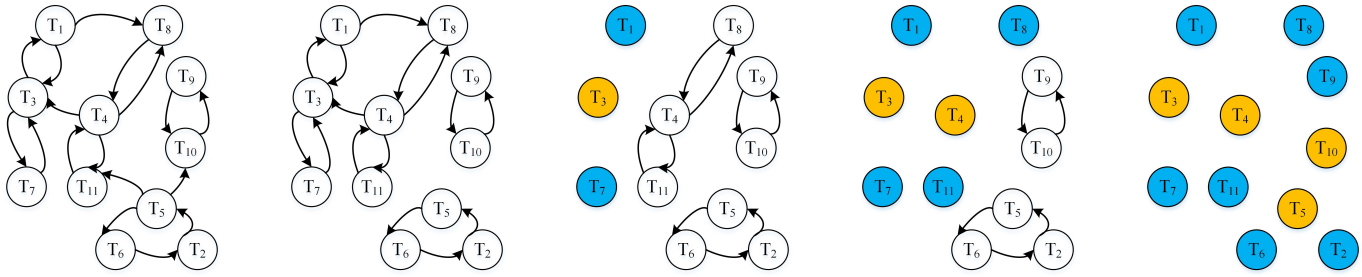
```
1  Initialize an output TDG;
2  B' ← B;
3  while |B'| > 0 do
4      CG ← ExecuteParallel(B');
5      B' ← FindAbortTransactionSet(CG);
6      CG' ← RemoveVertexFromCG(B');
7      L ← TopologicalSort(CG');
8      for each t ∈ L do
9          Commit(t);
10         TDG ← UpdateGraph(t);
11 return TDG;
```

Algorithm 1 briefs how to enable concurrency in the *proposal* phase and generate a schedule log represented by TDG. Line 2 initializes B' with B, which indicates the current restart transaction set. Line 4 firstly runs B' in parallel. When all transactions reach validation phase of OCC, we construct a local CG by creating one vertex per transaction and one edge per *read-write* conflict. Next, function $FindAbortTransactionSet$ computes an optimal vertex set B' upon CG. After removing B' and related edges from CG, the algorithm repeatedly commits transactions without any incoming edge using topological sort. Sequentially, the primary can obtain a serilaizable schedule $\mathcal{L}$ for rest transaction in CG' by a topological sort (line 7). Each successful commit triggers function $UpdateGraph$, which creates a new vertex and makes use of t's read set to generate edges. The procedure loops until all transactions in one batch are committed.

Finding abort transactions

The heart of our concurrent *proposal* phase lies on the greedy function $FindAbortTransactionSet$ (Algorithm 2). The heuristic rule behind this function is that each strongly connected component (SCC) of CG must contain at least one cycle. $FindAbortTransactionSet$, as a ranking function, selects vertices that are most likely to be involved in an abort set and meet our objective, generating a TDG with the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering



(a) Original Conflict Graph (b) Find all SCCs (c) Abort T_3 (d) Abort T_4 (e) Abort T_5 and T_{10}

Fig. 6: An example of $FindAbortTransactionSet$ algorithm. Yellow vertices form the abort set, and blue ones are pruned

highest parallelism. Line 2 trims vertices whose in-degree or out-degree is zero. This prune function is called each time we delete a vertex from CG (Line 11). Some DFS-based algorithms like Tarjan's [31] and Kosaraju's [32] can compute all SCCs in linear time.

For each SCC, function $GreedySelectVertex$ returns a subset of aborted transactions, and the output set V is the union of all subsets. Lines 7-12 in Algorithm 2 explain how we recursively choose vertices to form the abort set. SCCs of size one are pruned cause they have no chance to be included in any cycle (Line 8). Then we sort all vertices within one SCC in descending order by the degree-based strategy defined by Strategy 1 and greedily select the top-ranked vertex (Line 10). Line 11 removes the chosen vertex $\overline{V}$ and returns the remaining pruned graph. Then we recursively call function $GreedySelectVertex$ until sizes of all SCCs are no greater than 1. Such degree-based heuristic approach has been proven to perform well for FVS problem [33].

Strategy 1 (max-sum & min-out strategy)**.** *Suppose we have an aborted vertex T_a in CG with an outgoing edge set $E_o = \{(T_a, T_i)|i \neq a\}$ and an incoming edge set $E_i = \{(T_j, T_a)|j \neq a\}$. If we restart T_a, then we have a corresponding incoming edge set $E_i' = \{(T_i, T_a)\}$ of T_a in the transformed TDG, and none of E_i is included. As the objective is to minimize both the FVS size and the density of TDG, we choose the vertex with the greatest sum of degrees, breaking ties using minimal out-degree so that a size-minimized FVS as well as a sparser TDG can be obtained.*

Algorithm 2: FindAbortTransactionSet

Input: Conflict Graph CG
Output: V, a vertex set to be aborted

```
1  V ← ∅;
2  CG' ← Prune(CG);
3  SCC ← Tarjan(CG');
4  for each S ∈ SCC do
5      V ← V ∪ GreedySelectVertex(S);
6  return V;

7  function GreedySelectVertex(S)
8      if |S.V| ≤ 1 then
9          return ∅;
10     V' ← ChooseVertexByStrategy(S);
11     S' ← Prune(S \ V');
12     return V' ∪ GreedySelectVertex(S');
```

Example 3. *Figure 6 gives an instance of the greedy algorithm described above. We use Tarjan's SCC algorithm [31] to partition the original CG into several SCCs, and remove SCCs of size one which will be marked blue. In the first SCC that consists of six vertices T_1, T_3, T_4, T_7, T_8 and T_{11}, both vertices T_3 and T_4 have the greatest sum of degrees, but T_3's out-degree is smaller, we then abort T_3, mark it yellow and subsequently remove all relevant edges. Vertices with zero in-degree or out-degree are trimmed during the algorithm. Here, vertices T_1 and T_7 are all removed (Figure 6(c)). The algorithm takes the remaining part of the first SCC as input and recursively finds the next aborted transaction, T_4 (Figure 6(d)). We then move to next SCC containing vertices T_2, T_5, and T_6. Since three vertices have the same in-degree and out-degree, we can add any one of them to the abort set. Here we select vertex T_5 and prune vertices T_2 and T_6. The same goes for the last SCC containing vertices T_9 and T_{10}. In Figure 6(e), we have our final abort set $B' \leftarrow \{T_3, T_4, T_5, T_{10}\}$.*

Reordering transactions to acquire an optimal commit order surely increase the throughput in the *proposal* phase, so does latency. However, validators occupy the vast majority of nodes in blockchain systems. Sacrifices made by the primary bring much more benefit to validators and further improves the overall performance.

4.2 Generate Moderate Granularity Schedule Log

By now, we have a density-minimized TDG. This kind of fine-grained schedule log that provides every edge a consistent read set, however, will cause too much communication overhead. We intend to cut TDG into even sub-graphs to enhance parallelism and decrease the communication cost as well.

Since finding an optimal partition of TDG that minimizes the size of all R_j^i is an NP-hard [30] problem, we propose another greedy algorithm described in Algorithm 3. Our algorithm is on the basis of a simple rule, that is edges with larger weight are preferred to be included in a part so that the probability of edges with smaller weight being cut edge is increased.

Algorithm 3 generates a balanced partition, each sub-graph is no more than the threshold τ, and the size of all R_j^i is minimized. Lines 1 calculates the cost of all vertices in TDG. Then we reorder all edges of E by their weight in descending order (Line 3). Subsequently, we traverse each edge e to compute a partition with minimal edge cut (Lines 4-10). Basically, if e connects to at least one unvisited vertex u, we add it to the current subgraph $currV$ (Line 10). When

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering

Algorithm 3: τ-Constrained Partitioned TDG

Input: A transaction dependency graph $TDG(V, E)$, workload threshold τ
Output: A partition of TDG, $P = \{V_1, \ldots, V_k\}$

1 $cost \leftarrow \sum_{v \in V} \omega(v)$;
2 $currV \leftarrow \emptyset$;
3 $E' \leftarrow SortByWeight(E)$; /* Sort in descending order */
4 **for** *each* $e \in E'$ **do**
5 $uv \leftarrow e.getUnvisitedVertex()$;
6 **if** $\omega(currV) + \omega(uv) > \tau \cdot cost$ **then**
7 $P.add(currV)$; /* Get one part */
8 $currV \leftarrow \emptyset$;
9 $visited[uv] \leftarrow true$;
10 $currV \leftarrow currV \cup \{uv\}$;
11 **for** *each* $v \in V$ **do**
12 **if** $visited[v] = false$ **then**
13 **if** $\omega(currV) + \omega(v) > \tau \cdot cost$ **then**
14 $P.add(currV)$;
15 $currV \leftarrow \emptyset$;
16 $visited[v] \leftarrow true$;
17 $currV \leftarrow currV \cup \{v\}$;
18 **return** P;

$\omega(currV)$ exceeds the upper bound weight $\tau \cdot cost$, we get a new part (Lines 6-8). After visiting every edge of E, there may exist some unvisited vertices without incoming and outgoing edges. So, lines 11-17 iterate over all vertices in V and assign them to an appropriate part. Algorithm 3 can compute a balanced partition in one-pass accessing with time complexity of $O(|V| + |E|)$. And sorting edges takes $O(|E| \log |E|)$ time using quick sort algorithm [34].

Example 4. *Figure 8 illustrates a simple bi-partition process after applying our partition algorithm to an example transaction dependency graph. Suppose a vertex set* $V \leftarrow \{T_1, T_2, \ldots, T_{11}\}$ *has a corresponding weight set* $W \leftarrow \{11, 13, 8, 6, 7, 12, 1, 9, 3, 2, 1\}$. τ *is set to 0.5. We sort all edges by their weight as line 3 suggests. Then we start by the edge with the greatest weight which is edge* $e : T_8 \rightarrow T_9$. *Both start vertex and end vertex of* e *are unvisited, so we add them to* $currV$. *Now the cost of* $currV$ *is updated to 12. Next, we process edges* $T_4 \rightarrow T_8$, $T_1 \rightarrow T_4$ *and* $T_1 \rightarrow T_3$ *in order. After that, our first subgraph contains vertices* T_1, T_3, T_4, T_8 *and* T_9. *When dealing with edge* $T_2 \rightarrow T_6$, *the cost of current part* V_1 *is 37, which exceeds* $\tau \cdot 73$, *so vertices* T_2 *and* T_6 *are included in a new part. After visiting all edges, there are still two unvisited vertices* T_7 *and* T_{10}. *We add them to* $currV$ *one by one. Finally, we have a 2-way partition presented in Figure* $8(e)$ *whose edge-cut is 4.*

4.3 Replay Transactions in the Validation Phase

A deterministic OCC protocol

The commit order of the reordering OCC is not deterministic since the interleavings are arbitrary. Since TDG offers all conflict relationships among transactions, and the commit order is predefined as $\mathcal{L}$, batching and reordering transactions in the second phase is unnecessary. So we propose a deterministic optimistic concurrency control protocol called DeOCC with the help of partitioned TDG. Providing

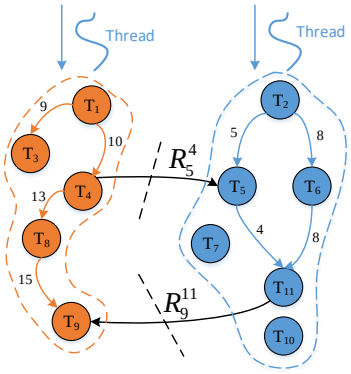


Fig. 7: Decentralized transaction replaying on every node

requisite *consistent read sets* for every subgraph, DeOCC is a non-blocking and no rollback protocol naturally. Every part can be executed concurrently and independently. We employ a decentralized execution approach proposed by Anjana [11], which is shown in Figure 7. Multiple threads are working on subgraphs concurrently in the absence of master thread.

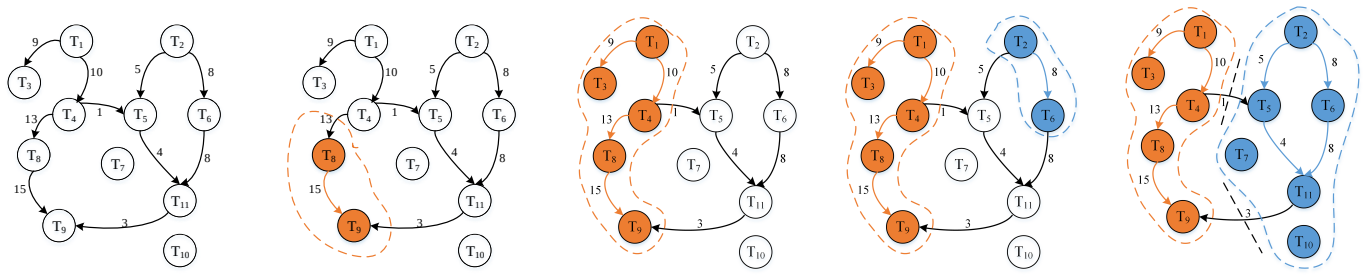
We add several modifications to allow transactions to run concurrently with a predetermined serialization order. Original OCC transactions read values either from shared storage or their own write sets. Since a *consistent read set* R_j^i preserves all data items that T_j needs to read from T_i's updates, every read operation is able to obtain a consistent value from the graph structure. DeOCC transactions have no more occasion to check the consistency of their read sets. Moreover, they need to verify the correctness of R_j^i cause the primary may be malicious and send false value. Of course, the inherent verification mechanism of current blockchain platforms still works under our deterministic protocol. After executing all transactions and finishing the state transition, one can recalculate the state Merkle root and compare it with the root sent by the primary. Nevertheless, this default setting wastes computational resource because the correctness is verified only after completing the execution. Instead, we propose a new verification scheme embedded in our DeOCC protocol, which will be explained later. Recall that the validators need to produce the same serialized commit order with the primary. Given this constraint, DeOCC transactions must commit according to $\mathcal{L}$.

Algorithm 4 describes a normal DeOCC transaction. When a DeOCC transaction t begins, it gets an additional parameter, serial number seq_t, representing the order of this transaction in $\mathcal{L}$.

In read phase, a read operation on *item* returns the value either from transaction t's own write set (Line 5) or its *consistent read set* (Line 7). A value from local storage is returned if transaction t doesn't have any *consistent read set*. Write operations remain the same with the OCC transaction which buffer new values in other locations and apply a *deferred write* strategy.

Transactions that pass our new verification scheme will enter the write phase. We force transactions to commit according to the predefined order $\mathcal{L}$ so that every transaction has to wait until all its predecessor transactions commit (Line 14). After installing transaction's updates, the global sequence value seq_c is assigned to seq_t as line 18 in Algo-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering



(a) Original TDG (b) Add T_8 and T_9 to V_1 (c) Get first part (d) Add T_2 and T_6 to V_2 (e) Edge cut is 4

Fig. 8: An example of 2-way partitioned TDG

rithm 4 suggests.

Algorithm 4: A DeOCC transaction t

```
1  when BeginTransaction do
2      seq_t ← seq; /* Assign a sequence number */
3  when ReadData do
4      if WS(t).contains(item) then
5          ReadData(item, WS(t));
6      else if TDG.contains(item) then
7          ReadData(item, TDG);
8      else
9          ReadData(item);        /* Read from local
               storage */
10     R(t).put(item);            /* Update read set */
11 when WriteData do
12     WS(t).put(item);           /* Deferred writes */
13 when CommitTransaction do
14     ConditionWait(seq_c = predecessor(seq_t));
15     if VerifyReadset(RS(t)) then
16         ComputeConsistentSet(WS(t), TDG);
17         WriteBack(WS(t));
18         seq_c ← seq_t;
19     else
20         AbortTransaction(t);
```

Restricting commit order leads to a serial commit, which will cause a loss of parallelism. To migrate the loss as much as possible, we see room for the transaction that is next to commit. Notice that, at any time, there exists one single transaction which is about to commit according to the predefined commit order. Another fact is that most of the OCC overhead lie on the *deferred write* mechanism and validation of the read set. Base on these two facts, we further improve the normal DeOCC protocol with a fast mode proposed. When current committed sequence number seq_c equals to a transaction t's sequence number seq_t (Line 1), we say t then switches to the fast mode. Algorithm 5 demonstrates some details about the fast mode execution.

As shown in Algorithm 5, the validation phase moves forward when a transaction is going to enter the fast mode (Lines 2-5). Likewise, transactions need to validate the normal DeOCC execution done up to that point.

With our fast mode, write operations in read phase use *direct write* strategy instead of deferred updates. Since transactions perform in place updates, read operations no longer read values from R_j^i or its write set. They simply read the current data item's value. Also, the cost of tracking

Algorithm 5: Fast mode of a DeOCC transaction t

```
1  when seq_c = predecessor(seq_t) do
2      if VerifyReadset(RS(t)) then
3          WriteBack(WS(t));
4      else
5          AbortTransaction(t);
6  when ReadData do
7      ReadData(item);            /* Read from local
           storage */
8  when WriteData do
9      WriteData(item);           /* Direct writes */
10 when CommitTransaction do
11     seq_c ← seq_t;
```

read sets will be eliminated. For we combine read phase and write phase, there is no more *write back* step when transactions commit.

The correctness of normal DeOCC is maintained because the condition wait (Line 14 of Algorithm 4) guarantees atomicity and makes only one transaction commit at a time. As for the fast mode, to conform with the predefined order, only one single transaction is given the privilege to write directly to the shared storage. Hence, moving forward the validation phase and replacing the deferred writes with direct writes do not affect the correctness.

Verification of malicious primary

As aforementioned, a DeOCC transaction will never read an inconsistent value because R_j^i prepares all proper values for every transaction. However, parties involved in blockchain platforms like Ethereum have no trust in each other. In other words, *consistent read sets* transferred by the primary require to be verified for their correctness.

The default setting of blockchain uses the Merkle root to check the validity of the state transition. All of the existing research works adopt the default verification mechanism to identify malicious behaviors. By this way, however, lots of computational resources may be wasted if the primary is a malicious one. We intend to embed the verification process to our deterministic protocol. Basically, each validator obtains the *consistent read sets* R_j^i for each transaction after execution. If it's provided with the $\overline{R_j^i}$ in the transaction dependency graph, then the transaction contrasts the computed R_j^i and $\overline{R_j^i}$, and a *verified* tag will be attached to all items in $\overline{R_j^i}$ if $R_j^i = \overline{R_j^i}$. In this way, transactions

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering

check whether their read values are all tagged with *verified*. Any unverified item will cause aborting the block. Different from the previous works, our verification scheme can detect malicious primary beforehand.

4.4 Integrated with PBFT

The approach described in this section assumes there is only one round of communication between the primary and validators. However, permissioned blockchain often adopts a BFT-like consensus algorithm, e.g., PBFT, which includes three communication phases. When integrated with such consensus protocol, our proposed concurrency protocol can be further optimized.

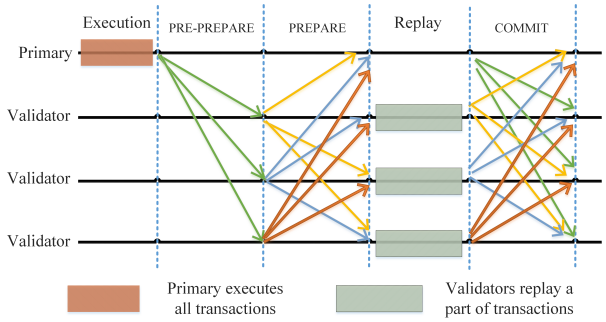


Fig. 9: Opitmized protocol when combined with PBFT

In a distributed system that runs PBFT algorithm, at least $3f + 1$ nodes are required to reach agreement, where f is the maximum number of faults the system can tolerate. But, as for execution, a simple majority of correct nodes suffice to mask Byzantine faults, i.e., $2f + 1$ execution nodes can ensure the correctness of results. We optimize the two-phase execution protocol in terms of validator replay by making use of such property. We assign every subgraph to $2f+1$ execution nodes, adequate to ensure the correctness of results. Figure 9 shows how the optimized protocol works. The primary starts consensus after executing a batch of transactions and generating TDG. Once a node is prepared, i.e., receiving $2f$ matching prepare messages from other nodes, all subgraphs assigned to it are computed with consistent hashing. More specifically, the consistent hashing maps each subgraph to a location of a hash ring, and finds the nearest $2f + 1$ nodes clockwise for execution. After replaying, validators send commit message along with the executing results to other nodes. In commit phase of PBFT, besides $2f + 1$ matched commit messages, a validator needs to receive $f + 1$ same results for every subgraph. Thus validators can save $\frac{f}{3f+1}$ computation cost in average. We present the optimization effect in evaluations.

Modifying the primary to execute transactions concurrently does not affect the correctness of PBFT, because it happens before the consensus processes. The embedded validation phase does not affect the correctness as well. If the primary sends a false TDG, validators won't broadcast commit messages. After timeouts, PBFT triggers view change protocol. And the deterministic of execution is maintained by DeOCC introduced in Section 4.3.

5 Performance Analysis

We analyze the performance of our two-phase framework in this section. As mentioned above, the primary finds a serialization order using reordering OCC in the *proposal* phase, and then validators replay and verify the schedule log with DeOCC protocol in the second phase. The overall cost of our method consists of two parts: communication cost and computational cost. Communication cost is brought by the information sent from the primary to validators, and the computational cost is the execution time of a batch of transactions. To some extent, there is a trade-off between communication cost and computational cost, i.e., receiving more information contained in the schedule log will enable higher execution speed in validators.

Communication cost. Suppose the original transaction dependency graph $TDG(V, E)$ is broke down to k subgraphs which are $G[V_1], G[V_2], \ldots, G[V_k]$. Each subgraph $G[V_i]$ has an extra input data set D_i. The overall communication cost is computed as $|T| = \sum_{i=1}^{k}(|G[V_i| + |D_i|)$.

For simplicity, the time to transfer data via network is computed as the data volume divided by network bandwidth. Let β denote the network bandwidth, the overall cost is computed as $|T|/\beta$.

Computational cost. In our framework, all smart contracts must be evaluated one by one in each validator. Moreover, all smart contracts belonging to the same subgraph will be verified in the same core. Hence, if the number of subgraphs is smaller than the number of cores, i.e, $k < m$, some cores will be idle. Therefore, in real-life cases, we will set $k \gg m$ so that all sub-tasks can be assigned to each core evenly to reduce the execution cost.

The upper and lower bounds of the computation cost to execute G on an m-core processor are computed below, where Ψ is the overall computation cost, and τ is the maximum workload of each subgraph.

$$\begin{aligned} LB &= \Psi/m \\ UB &= \Psi(\frac{1-\tau}{m} + \tau) \end{aligned}$$

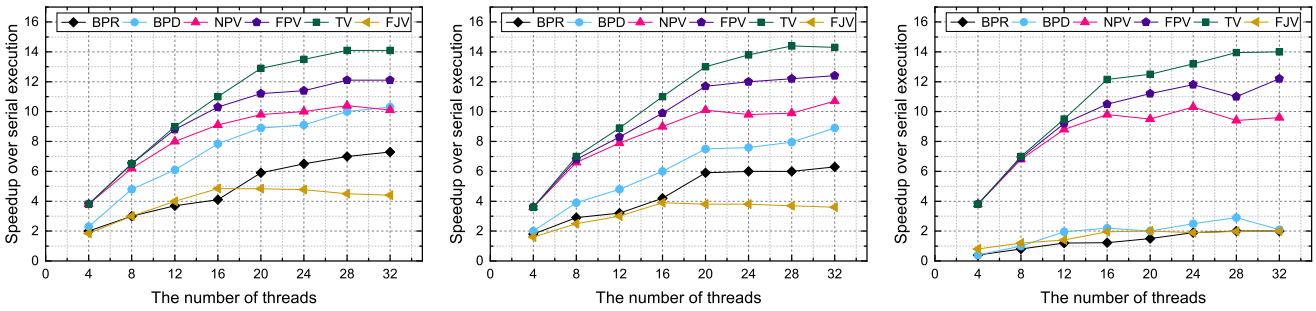
Each validator will try to evaluate all smart contracts in a batch. Under the best situation, the evaluation task will be dispatched to m cores evenly. Hence, the lower bound will be Ψ/m. However, due to load unbalance, all m cores may not complete at the same time. In ultimate situation, when one core starts to execute the largest sub-task, the rest $m-1$ cores happen to finish all subtasks assigned to them at the same time. In this way, we get the upper bound (UB), as listed above.

It is interesting to discuss the case where each subgraph only contains one transaction, i.e., $k = n$. In this case, the m-core validation can verify all smart contracts almost evenly. Meanwhile, the read set for each smart contract must be ready, which means more data to be transferred.

6 Experiments

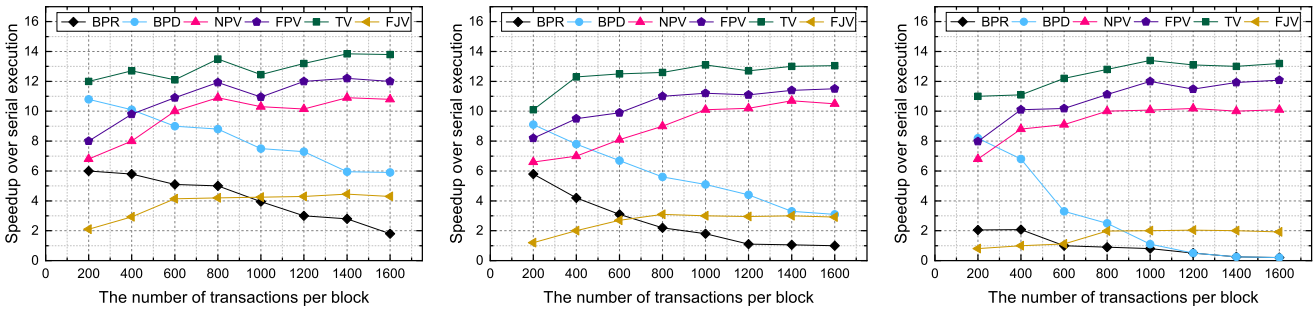
We report extensive experimental results in this section.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering



(a) Low contention (b) Medium contention (c) High contention

Fig. 10: Speedup against the number of threads



(a) $skew = 0.1$ (b) $skew = 0.5$ (c) $skew = 0.7$

Fig. 11: Speedup against the number of transactions

6.1 Benchmark

As popular for OLTP applications, SmallBank [26] is also widely employed to evaluate blockchain systems [4], [24], [35], [36]. Based on a schema of three tables, SmallBank defines four basic procedures to model a simple banking scenario, namely *Amalgamate*, *WriteCheck*, *DepositChecking* and *TransactSaving*, each owing several read/write operations on up to two accounts. For example, the *Amalgamate* transaction will read and write two accounts while the *DepositChecking* just accesses single one account. Since bank transfer is common, we also implement *SendPayment* transaction to extend the original benchmark. As shown in Figure 2, SendPayment transaction is implemented by invoking *transfer* function of the SimpleBank smart contract. All types of transactions are generated in random. The primary assembles transactions that invoke different smart contracts. So, our extended benchmark, called SmallBank+, can simulate real workload properly.

6.2 Experiment Setup

As the program written in Solidity cannot be executed in a multi-threaded manner, we implement SmallBank+ in Java to utilize CPU resource more efficiently. Each block contains a set of transactions which are implemented by using *callable* objects in Java. Our workload generator generates blocks with a combination of transaction count, the number of accounts and an initial balance for each group of experiments. The transaction type is generated uniformly using *Random* class in Java. Whenever a record is read or updated by a transaction, we add the key and value of the record to the corresponding read/write set. Data access pattern follows a Zipfian distribution to simulate data skew situation. Specifically, greater skew value means more transaction conflicts.

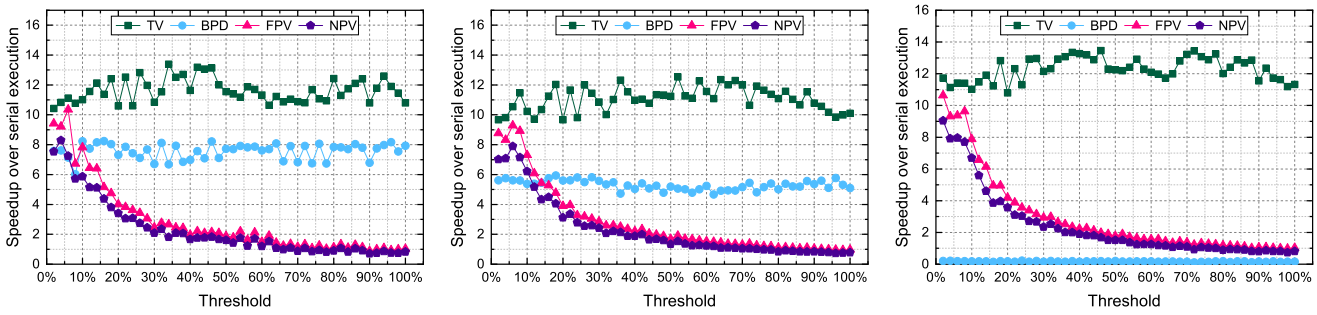
TABLE 3: Methods to be evaluated

Execution phase	Method	Description
proposal phase	BPD	Degree-based strategies (Sec. 4.1)
	BPR	Random-selection based primary
validation phase	TV	Total validator
	FPV	Fast DeOCC (Sec. 4.3)
	NPV	Normal DeOCC (Sec. 4.3)
	FJV	The fork-join validator [12]

A thread pool, created by *ThreadPoolExecutor*, executes all transactions concurrently. We implement Tarjan's algorithm [31] to separate all SCCs in an efficient way. Serial executor runs with only one thread as a baseline to highlight the effect of our proposed two-phase scheme. We populate the database with 100k customers, including 100k checking and 100k saving accounts. We use three different skew parameters to indicate the conflict intensity.

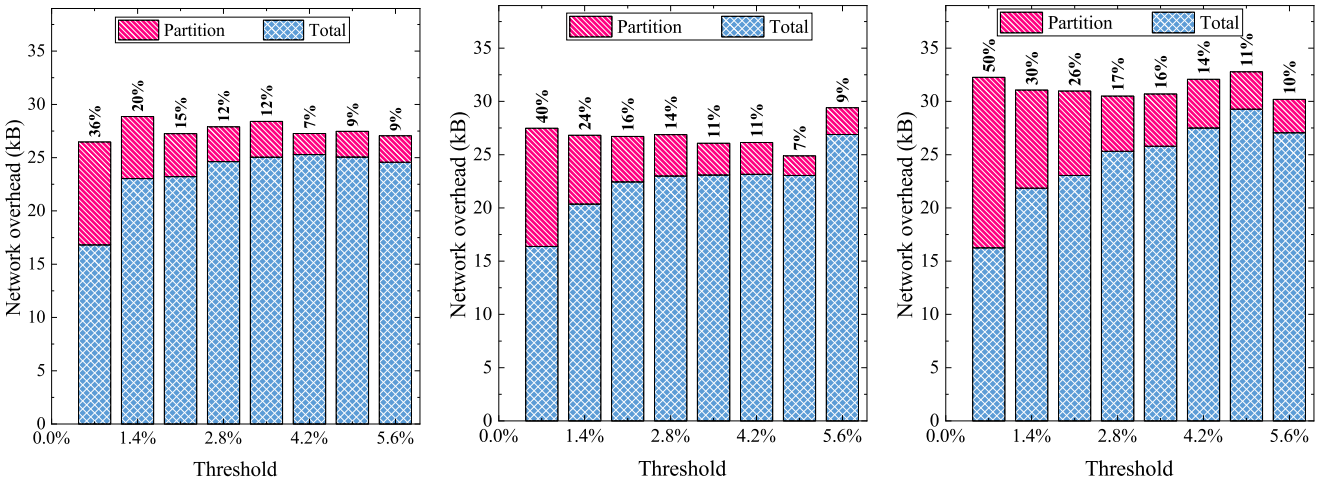
Table 3 lists all the methods evaluated in this paper, including those work in the *proposal* phase and *validation* phase respectively. First, in the proposal phase, we show the performance of batching primary that integrates with degree-based strategies (BPD). Moreover, we evaluate the primary by randomly selecting transactions to abort (BPR) for the purpose of comparison. Second, we evaluate four methods in the *validation* phase, including NPV, FPV, TV and FJV. The first two balance the communication cost and the degree of concurrency by partitioned TDG, while the third one, TV, merely focuses on the execution efficiency. The fork-join validator (FJV) proposed by Dickerson [12] is implemented and evaluated as a baseline. Third, we evaluate our method in a practical blockchain system that adopts PBFT as consensus protocol. We conduct the PBFT consensus protocol on four physical nodes, each of which is equipped with 160GB memory and 1Gb Ethernet. At the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2021.3059959, IEEE Transactions on Knowledge and Data Engineering



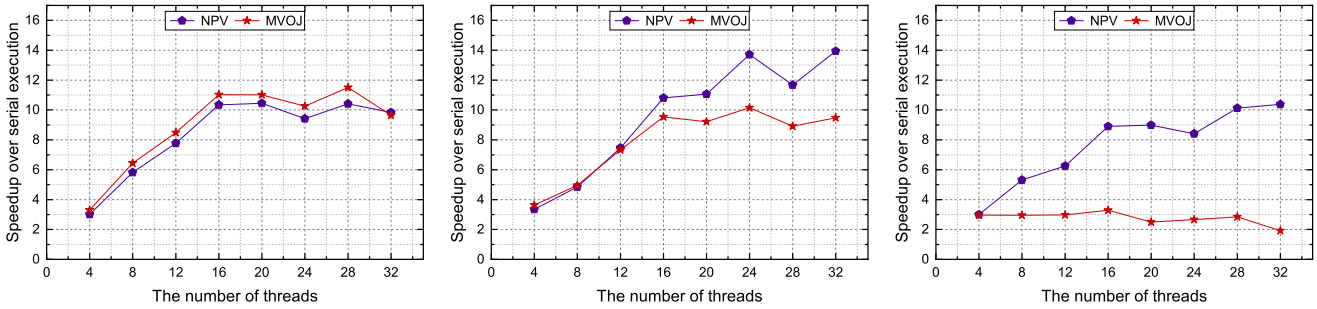
(a) Low contention. (b) Medium contention. (c) High contention.

Fig. 12: Speedup against the workload threshold (τ)



(a) $skew = 0.1$ (b) $skew = 0.5$ (c) $skew = 0.7$

Fig. 13: Communication cost with different τ



(a) Low contention (b) Medium contention (c) High contention

Fig. 14: Comparison with STM against number of threads

network layer, each node maintains a TCP connection with its peers, so that all nodes can communicate with each other via P2P protocol.

Since smart contract execution in real world blockchain systems is neither multi-threaded (EVM of Ethereum) nor two-phase style (Chaincode in Fabric), we conduct all experiments on one machine to reveal the performance improvement for both phases, which is fairly common in a series of related works [11]–[13]. The machine, equipped with 160GB memory and a 2-socket Intel Xeon Silver 4110 CPU @2.10GHz with 8 cores per socket and two hyper-threads per core, runs CentOS 7 system with JDK version 1.8. All of experimental figures show the average of 10 runs.

6.3 Experimental Reports

Varying the number of threads.

Figure 10 shows the execution throughput (Y-axis) against the number of threads (X-axis), by varying the number of threads from 1 to 32 with a fixed transaction count (400 transactions per block) under 3 different skew parameters. BPD achieves approximate 10× speedup over serial execution when contention is low ($skew = 0.1$) and 9× speedup even when handling workload with medium contention ($skew = 0.5$). Under all situations, BPD significantly surpasses BPR. In addition, the speedup of BPD and BPR drops below 3× when $skew = 0.7$, because high contention, though infrequent in blockchain applications, leads

to too many aborts. NPV gains 10× speedup in average. After switching to fast mode, FPV outperforms NPV by 20 percent. TV obtains theoretically optimal performance, with about 14× speedup as the peak throughput. The throughput of FJV keeps falling down when data skews. In all cases, both BPD and FPV outdo FJV markedly. Moreover, the behaviors of NPV and FPV are similar under different conflict intensity, indicating they are resistant for the data skews.

Varying the number of transactions per block.

Figure 11 reports the throughput of different approaches upon fixed 16 threads when the number of transactions per block varies from 200 to 2,000. BPD improves the execution throughput up to 10× speedup when the number of transactions per block is less than 400. Although the throughput of both BPD and BPR goes down with the block size rises, BPR degrades significantly, because random-selection strategy takes more time to compute an FVS solution when the conflict graph is getting more complicated. For NPV and FPV, the speedup will increase in the early stage when each block contains fewer transactions (e.g., 200-800 transactions per block in Figure 11(a)), due to the challenge to dispatch tasks evenly to all cores. The FJV method has similar property with partition validators, but with smaller speedup.

Impact of the workload threshold τ on performance.

We analyze how much impact on performance and communication cost when varying threshold τ from 0.0035 to 100%, which indicates that a dependency graph is cut into 1 to 288 subgraphs. Figure 12 shows the throughput when τ varies with 16 threads running. The speedup for FPV and NPV drops quickly when τ goes up. As reported in Section 5, greater τ causes more workload. When $\tau < 0.02$, FPV and TV have close performance, showing that FPV preserves the parallelism of validators as much as possible.

As all methods perform better with smaller τ, we focus on the communication cost when τ ranges from 0.0035 to 0.056. Figure 13 reports the the communication cost by varying τ from 0.0035 to 0.056. The communication cost here implies the total size of TDG to be transferred. When data skews, the communication overhead rises due to more inter-conflicts with a batch. In all three cases, our partition algorithm reduces the transferred size of consistent read set dramatically when $\tau > 0.015$. Even in the worst case ($skew = 0.7$), the communication cost saving is about 85%.

Comparison with MVOJ approach.

It is an alternate of Dickerson's in which transactions are executed concurrently in optimistic manner using STM. Since Anjana hasn't opened their source code and STM technique is hard to implement in our evaluation setups. Therefore, we implement the algorithm of [11] in Java, namely *Mulit-version Object-based Java*[2] (MVOJ), for fair comparison. Figure 14 reports the results of NPV and STM against the number of threads, by varying the number of threads from 1 to 32 under 3 different skew parameters. As can be seen, the performance of NPV and MVOJ is very close under low contention as shown in Figure 14(a), but NPV outperforms MVOJ significantly under high contention. Specifically, the NPV achieves up to 10× speed up under high contention. In contrast, the performance of MVOJ may degrade as the number of thread increases. This is because NPV can still partition TDG into mutliple sbugraphs evenly to achieve better parallelism while MVOJ would degrade to sequential execution. Therefore, we think our methods can outperform Anjana's work if both algorithms are implemented by same software tool.

2. The original algorithm based on STM is named as Multi-version Object-based STMs and we implement it in Java.

Integrated with BFT-SMaRt.

BFT-SMaRt is a robust Java-based Byzantine fault tolerant state machine replication library which implements a protocol similar to PBFT [37]. BFT-SMaRt is a high performance solution for BFT systems and can obtain 80k throughput under empty request/reply size benchmark [38]. The concurrent schemes proposed in this paper are integrated with BFT-SMaRt and evaluated in distributed settings. More specifically, the primary in BFT-SMaRt is instructed to execute a batch of transactions concurrently (Algorithm 1) before starting the three-phase consensus. The partitioned transaction dependency graph is encoded within the consensus message as well. After reaching consensus on the newly-proposed block, all validators replay the same batch of transactions concurrently and deterministically by using DeOCC. We also implement the optimized protocol in BFT-SMaRt and evaluate it in a fully distributed setup.

Figure 15 reports the the throughput of concurrent execution (fixed 16 threads) and serial execution when the number of clients varies from 20 to 200. The throughput of both methods combined with BFT-SMaRt increases when more clients are launched. Our original concurrent approach obtains about 5,300 tps while the serial one reaches only 3,600 tps. Since the high-performance consensus of BFT-SMaRt makes serial execution a new bottleneck, applying concurrent execution will definitely improve the overall performance. The optimized protocol outperforms the original one by 300 tps, which confirms the efficiency of the optimization.

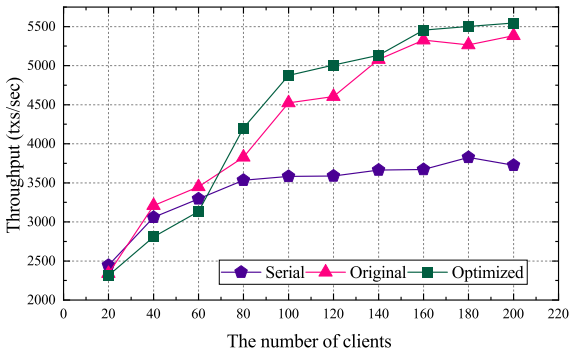


Fig. 15: System throughput vs. the number of clients

In the second set, we evaluate the overall throughput of BFT-SMaRt combined with the original concurrent protocol or the optimized one against the number of threads. Figure 16 shows the peak throughput of both protocols with 100 clients issuing requests. The performance displays a linear increasing with the increase of the number of threads whether or not it's optimized. Our optimized protocol consistently surpasses the original one. The throughput of the original protocol is lower than serial (3,600 tx/s) when using fewer threads, e.g., when the number of thread is 2. This is

because the impact caused by transferring extra information is more obvious than multi-threaded execution.

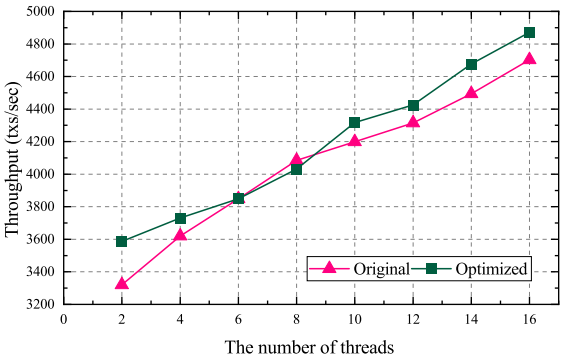


Fig. 16: System throughput vs. the number of threads

The last group of experiments shows the impact of τ on system performance. Figure 17 presents results for both original and optimized setups of BFT-SMaRt with 16 threads and 200 clients. In Figure 17, the peak throughput of the original protocol and the optimized protocol follow the same trend, which rise at first and then fall with the increase of τ. Because, when τ is small, the number of subgraphs is far greater than the number of cores, which implies that there is almost no influence on computation cost. However, as Figure 13 suggests, the communication cost decreases remarkably with the increase of τ. When τ is greater than 3.5%, the overall performance drops sharply. The reason is that fewer subgraphs make it harder to assign tasks to cores evenly, and the saving of communication cost is insignificant. Hence, appropriate granularity of transaction dependency graph is critical for the overall performance.

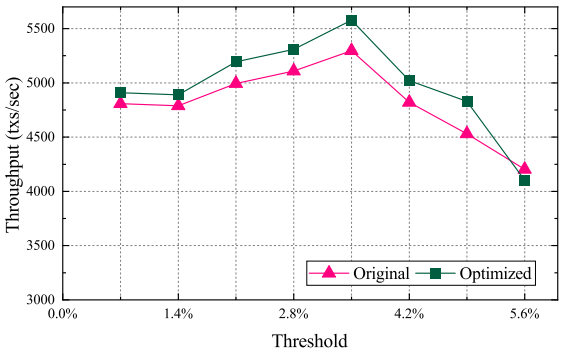


Fig. 17: System throughput vs. workload threshold τ

As a conclusion, our approach achieves maximal 10× speedup for the primary and 12× for concurrent validators. And we reduce approximate 90% communication overhead by only sacrificing about 14% performance. When integrated with an open source BFT system, the throughput can increase by 45% compared with serial execution. And the optimized approach yields an additional 300 increase of throughput on the basis of the original one.

7 Conclusion

In this study, we present an efficient two-phase execution framework to add concurrency to smart contracts of permissioned blockchain aiming for higher parallelism at both primary and validators. More specifically, (i) we propose an efficient variant of OCC protocol combined with natural batching feature and transaction reordering in the *proposal* phase, where a greedy algorithm is devised to solve the Min-Density FVS problem. (ii) We determine an appropriate granularity for the schedule log, along with a practical partition method to reduce the communication overhead and retain a large degree of concurrency during the *validation* phase. (iii) We bring up a concurrent scheme, named DeOCC, to deterministically and efficiently replay the same schedule discovered by the primary. The evaluation shows that our two-phase execution framework achieves approximate 10× speedup for the primary and 12× for validators, and outperforms state-of-art solutions significantly. Moreover, the communication overhead drops sharply after applying our graph partition algorithm.

There are two possible directions of future work. One is to explore adaptive concurrency control for smart contract transactions, which can dynamically fit all kinds of workloads. Another is to search for solutions of concurrent execution in TEE (Trusted Execution Environment) represented by SGX (Intel Software Guard Extensions) [39].

Acknowledgements

This work was supported in part by National Science Foundation of China (U1911203, 61972152 and U1811264), Guangxi Key Laboratory of Trusted Software (kx202005), and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students (YBNLTS2019-021). Zhao Zhang is the corresponding author.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] "Ethereum," https://github.com/ethereum, 2014.
[3] Y. Zhu, Z. Zhang, C. Jin, A. Zhou, and Y. Yan, "Sebdb: Semantics empowered blockchain database," *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pp. 1820–1831, 2019.
[4] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1085–1100.
[5] L. Feng, H. Zhang, W.-T. Tsai, and S. Sun, "System architecture for high-performance permissioned blockchains," *Frontiers of Computer Science*, vol. 13, no. 6, pp. 1151–1165, 2019.
[6] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 31–42.
[7] "Ethermint," https://github.com/cosmos/ethermint, 2016.
[8] "Quorum," https://github.com/jpmorganchase/quorum, 2016.
[9] "Istanbul bft," https://github.com/ethereum/EIPs/issues/650, 2018.
[10] A. Baliga, I. Subhod, P. Kamat, and S. Chatterjee, "Performance evaluation of the quorum blockchain platform," *arXiv preprint arXiv:1809.03421*, 2018.
[11] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 83–92.
[12] 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*. ACM, 2017, pp. 303–312.
[13] A. Zhang and K. Zhang, "Enabling concurrency on smart contracts using multiversion ordering," in *APWeb and WAIM Joint International Conference on Web and Big Data*. Springer, 2018, pp. 425–439.

[14] D. Lea, "A java fork/join framework," in *Java Grande*, 2000, pp. 36–43.

[15] Y. Ji, Y. Chai, X. Zhou, L. Ren, and Y. Qin, "Smart intra-query fault tolerance for massive parallel processing databases," *Data Science and Engineering*, vol. 5, no. 1, pp. 65–79, 2020.

[16] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[17] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[18] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 49–60, 2016.

[19] B. Ding, L. Kot, and J. Gehrke, "Improving optimistic concurrency control through transaction batching and operation reordering," *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 169–182, 2018.

[20] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham, and I. Watson, "Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering," in *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 2009, pp. 4–18.

[21] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 70–80, 2010.

[22] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 613–624, 2017.

[23] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 478–493.

[24] 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 2019 International Conference on Management of Data*, 2019, pp. 105–122.

[25] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," 1999.

[26] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 729–738.

[27] "Solidity," https://solidity.readthedocs.io/en/latest/.

[28] P. Festa, P. M. Pardalos, and M. G. Resende, "Feedback set problems," in *Handbook of combinatorial optimization*. Springer, 1999, pp. 209–258.

[29] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[30] V. Kann, "On the approximability of np-complete optimization problems," Ph.D. dissertation, 1992.

[31] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[32] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[33] H. Stamm, "On feedback problems in planar digraphs," in *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 1990, pp. 79–89.

[34] R. Sedgewick, "Implementing quicksort programs," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, 1978.

[35] "Hyperledger sawtooth," https://github.com/hyperledger/sawtooth-core, 2016.

[36] 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 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 543–557.

[37] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[38] A. N. Bessani, J. Sousa, and E. A. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2014, pp. 355–362.

[39] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

Cheqing Jin Cheqing Jin received the bachelor and master degrees from Zhejiang University, and the PhD degree in computer science from Fudan University, in 1999, 2002, and 2005, respectively. He is a professor with East China Normal University. He is the winner of the Fok Ying Tung Education Foundation Fourteenth Young Teacher Award. He is a senior member of China Computer Federation, and serves as an editor of Computer Research and Development. He has co-authored more than 100 papers, some of which received excellent paper awards, such as the best paper award of the Chinese Journal of Computers, best paper award of pervasive computing and embedding from the Shanghai Computer Society. His research interests include blockchain, streaming data management, location-based services, and uncertain data management.

Shuaifeng Pang received the BE degree in software engineering from East China Normal University, Shanghai, China, in 2017. Currently, he is a Master student supervised by Prof. Cheqing Jin. His research mainly focuses on concurrency of smart contracts in blockchain.

Xiaodong Qi received the BE degree in software engineering from Nanjing Normal University, Nanjing, China, in 2016. Currently, he is a PhD student supervised by Prof. Cheqing Jin. His research interests are in performance and scalability of blockchain systems, including BFT consensus protocols and blockchain storage. His works appeared in several major international conferences and journal on data management, including ICDE, TKDE, DASFAA, etc.

Zhao Zhang got her Bachelor degree in Computer Science from Northwest Normal University in 2000, and her Master and Ph.D degrees from East China Normal University in 2003 and 2012 respectively. She is a professor with East China Normal University (ECNU). Her research interests include distributed databases, blockchain and location based service. Her works appeared in several major international conferences and journal on data management, including VLDB, ICDE, TKDE, DASFAA, etc.

Aoying Zhou received the master and bachelor degrees in computer science from Sichuan University, and the PhD degree from Fudan University, in 1988, 1985, and 1993, respectively. He is a professor, and vice president of East China Normal University. He is the winner of the National Science Fund for Distinguished Young Scholars supported by the National Natural Science Foundation of China (NSFC) and the professorship appointment under the Changjiang Scholars Program of Ministry of Education (MoE). He is a CCF (China Computer Federation) fellow, and associate editor-in-chief of the Chinese Journal of Computer. He served general chair of the ER2004, vice PC chair of ICDE2009 and ICDE2012, and PC co-chair of VLDB2014. His research interests include Web data management, data management for data-intensive computing, in-memory cluster computing, and distributed transaction.