ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems

Mohammad Javad Amiri Divyakant Agrawal Amr El Abbadi
University of California Santa Barbara
Santa Barbara, California
{amiri, agrawal, amr}@cs.ucsb.edu

***Abstract*—Many existing blockchains do not adequately address all the characteristics of distributed system applications and suffer from serious architectural limitations resulting in performance and confidentiality issues. While recent permissioned blockchain systems, have tried to overcome these limitations, their focus has mainly been on workloads with no-contention, i.e., no conflicting transactions. In this paper, we introduce *OXII*, a new paradigm for permissioned blockchains to support distributed applications that execute concurrently. OXII is designed for workloads with (different degrees of) contention. We then present *ParBlockchain*, a permissioned blockchain designed specifically in the OXII paradigm. The evaluation of ParBlockchain using a series of benchmarks reveals that its performance in workloads with any degree of contention is better than the state of the art permissioned blockchain systems.**

***Index Terms*—Blockchain, Permissioned, Consensus, Dependency graph, contention**

I. Introduction

A blockchain is a distributed data structure for recording transactions maintained by many nodes without a central authority [12]. In a blockchain, nodes agree on their shared states across a large network of *untrusted* participants. Blockchain was originally devised for Bitcoin cryptocurrency [28], however, recent systems focus on its unique features such as transparency, provenance, fault-tolerant, and authenticity to support a wide range of distributed applications. Bitcoin and other cryptocurrencies are *permissionless* blockchains. In a permissionless blockchain, the network is public, and anyone can participate without a specific identity. Many other distributed applications such as supply chain management [23] and healthcare [6], on the other hand, are deployed on *permissioned* blockchains consisting of a set of known, identified nodes that still do not fully trust each other.

Distributed applications have different characteristics that need to be addressed by permissioned blockchain systems. Such applications require high performance in terms of throughput and latency, e.g., a financial application needs to process tens of thousands of requests every second with very low latency. Distributed applications might also have workloads with high-contention, i.e., conflicting transactions. Under these workloads, several transactions simultaneously perform conflicting operations on a few popular records. These conflicting transactions might belong to a single application or even a set of applications using a shared datastore. While the sequential execution of transactions prevents any possible inconsistency, it adversely impacts performance and scalability. In addition, confidentiality of data is required in many applications. In blockchain, the logic of each application can be written as a *smart contract*, as exemplified by Ethereum [3]. A smart contract is a computer program that self-executes once it is established and deployed. Since smart contracts include the logic of applications, it might be desired to restrict access to such contracts. Cryptographic techniques are used to achieve confidentiality, however the considerable overhead of such techniques makes them impractical [5].

Existing permissioned blockchains, e.g., Tendermint [25] and Multichain [21], mostly employ an *order-execute* paradigm where nodes agree on a total order of the blocks of transactions using a consensus protocol and then the transactions are executed in the same order on all nodes sequentially. Such a paradigm suffers from performance issues because of the sequential execution of transactions on all nodes, and also confidentiality issues since every node access every smart contract. Hyperledger Fabric [5], on the other hand, presents a new paradigm for permissioned blockchains by switching the order of the execution and ordering phases. In Hyperledger Fabric, transactions of different applications are first executed in parallel and then an ordering service consisting of a set of nodes uses a consensus protocol to establish agreement on a total order of all transactions. Fabric addresses the confidentiality issues by restricting accesses to smart contracts, allows the non-deterministic execution of transactions by switching the order of the ordering and execution phases, and improves performance by executing transactions in parallel. However, in the presence of any contention in the workload, it has to disregard the effects of conflicting transactions which negatively impacts the performance of the blockchain.

In this paper, we present *OXII*: an order-execute paradigm for permissioned blockchains. OXII is mainly introduced to support distributed applications processing workloads with *some degree of contention*. OXII consists of *orderer* and *agent* nodes. Orderers establish agreement on the order of the transactions of different applications, construct the blocks of transactions, and generate a *dependency graph* for the transactions within a block. A dependency graph, on the one hand, gives a partial order based on the conflicts between transactions, and, on the other hand, enables higher concurrency by allowing the parallel execution of non-conflicting transactions. A group of agents of each application called *executors* are then responsible for executing the transactions of that application.

We then present *ParBlockchain*, a permissioned blockchain system designed specifically in the OXII paradigm. ParBlockchain processes transactions in the ordering and execution phases. In the ordering phase, transactions are ordered in a dependency graph and put in blocks. In the execution phase, the executors of each application execute the transactions of the corresponding application following the dependency graph. As long as the partial order of transactions in the dependency graph is preserved, the transactions of different applications can be executed in parallel.

A key contribution of this paper is to show how workloads with conflicting transactions can be handled efficiently by a blockchain system without rolling back (aborting) the processed transactions or executing all transactions sequentially. This paper makes the following contributions:

- *OXII*, a new paradigm for permissioned blockchains to support distributed applications that execute concurrently. OXII uses a dependency graph based concurrency control technique to detect possible conflicts between transactions and to ensure the valid execution of transactions while still allowing non-conflicting transactions to be executed in parallel.
- *ParBlockchain*, a permissioned blockchain system designed specifically in the OXII paradigm. The experiments show that workloads with any degree of contention will benefit from ParBlockchain.

The rest of this paper is organized as follows. Section II briefly describes current blockchain paradigms and their limitations. The OXII paradigm is introduced in Section III. Section IV presents ParBlockchain, a permissioned blockchain system designed specifically in the OXII paradigm. Section V shows the performance evaluation. Section VI presents related work, and Section VII concludes the paper.

II. Background

A blockchain is a distributed data structure for recording transactions maintained by many nodes without a central authority [12]. A blockchain replicates data over nodes using State Machine Replication (SMR). State machine replication is a technique for implementing a fault-tolerant service by replicating servers [26]. In the state machine replication model replicas agree on an ordering of incoming requests and then execute the requests in the same order. State machine replication approaches have been used in different synchronous and asynchronous networks to tolerate crash, malicious, or both failures. In a crash failure model, replicas may fail by stopping, and may restart, however, they may not collude, lie, or otherwise, attempt to subvert the protocol. In contrast, in a Byzantine failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior.

Blockchains use asynchronous fault-tolerant protocols to establish consensus. Since the nodes in a blockchain could behave maliciously, blockchains mainly use Byzantine fault-tolerant protocols to reach consensus.

In general, "ordering" and "execution" are the two main tasks of any fault-tolerant system. Fault-tolerant protocols mainly follow an *order-execute* paradigm where the network first, orders transactions and then executes them in the same order on all nodes sequentially.

Existing blockchain systems can be divided into two main categories: permissionless blockchain systems, e.g., Ethereum (with PoS-based consensus) [3] and permissioned blockchain systems, e.g., Tendermint (with BFT-type consensus) [25].

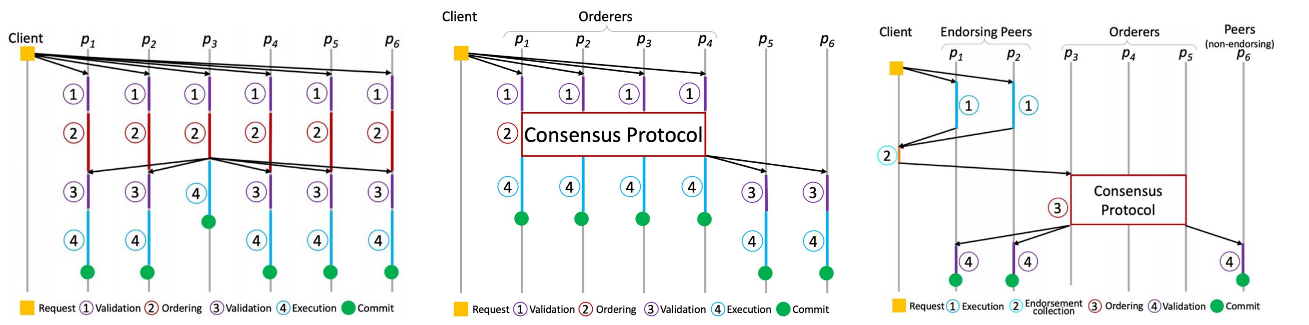
Permissionless blockchains are public, and anyone can participate without a specific identity. Permissionless blockchains mainly follow the order-execute paradigm where nodes validate the transactions, put the transactions into blocks, and try to solve some cryptographic puzzle. The lucky peer that solves the puzzle multicasts the block to all nodes. When a node receives a block of transactions, it validates the solution to the puzzle and all transactions in the block. Then, the node executes the transactions within a block sequentially. Such a paradigm requires all nodes to execute every transaction and all transactions to be deterministic.

Figure 1(a) shows the transaction flow for a permissionless blockchain. When a peer receives transactions from clients, in step 1, the peer validates the transactions, puts them into a block, and tries to solve the cryptographic puzzle. If the peer is lucky (p_3 in the figure) and solves the puzzle before other peers, it multicasts the block to all the peers. All the nodes then validate the block and its transactions (step 3), execute the transactions sequentially (step 4), and finally, update their respective copies of the ledger. Note that if multiple peers solve the puzzle at the same time, a fork happens in the blockchain. However, once a block is added to either of the fork branches, nodes in the network join the longest chain.

A permissioned blockchain, on the other hand, consists of a set of known, identified nodes but which do not fully trust each other. In permissioned blockchains, since the nodes are known and identified, traditional consensus protocols can be used to order the requests [11].

A permissioned blockchain can follow either order-execute or execute-order paradigm. In order-execute permissioned blockchains, as can be seen in Figure 1(b), a set of peers (might be all of them) validate the transactions, agree on a total order for the transactions, put them into blocks and multicast them to all the nodes. Each node then validates the block, executes the transactions using a "smart contract", and updates the ledger. A *smart contract* is a computer program that self-executes once it is established and deployed. Smart contracts are similar to *databases triggers* where the logic of the contract is triggered to be executed once some conditions or terms are met. They have the advantages of supporting real-time updates, accurate execution, and little human intervention.

In order-execute permissioned blockchains, similar to order-execute permissionless blockchains, every smart contract runs on every node. Smart contracts include the logic of applications and it might be desirable to restrict access to such contracts. While cryptographic techniques are used to achieve confidentiality, the considerable overhead of such techniques makes them impractical [5]. Furthermore the sequential execution of transactions on every node reduces the blockchain


Client
p_1 p_2 p_3 p_4 p_5 p_6
Request ① Validation ② Ordering ③ Validation ④ Execution ● Commit
Orderers
Consensus Protocol
Endorsing Peers
Peers (non-endorsing)
Request ① Execution ② Endorsement collection ③ Ordering ④ Validation ● Commit

(a) Order-Execute Paradigm (Permissionless) (b) Order-Execute Paradigm (Permissioned) (c) Execute-Order Paradigm (Permissioned)

Fig. 1. Existing Paradigms for Blockchains

performance in terms of throughput and latency.

In contrast to the order-execute paradigm, Hyperledger Fabric [5] presents a new paradigm for permissioned blockchains by switching the order of execution and ordering. The execute-order paradigm was first presented in Eve [22] in the context of Byzantine fault-tolerant SMR. In Eve peers execute transactions concurrently and then verify that they all reach the same output state, using a consensus protocol. In fact, Eve uses an Optimistic Concurrency Control (OCC) [24] by assuming low data contention where conflicts are rare.

Hyperledger Fabric uses a similar strategy; a client sends a request to a subset of peers, called endorsers (the nodes that have access to the smart contract). Each endorser executes the request and sends the result back to the client. When the client receives enough endorsements (specified by some endorsement policy), it assembles a transaction including all the endorsements and sends it to some specified (ordering) peers to establish a *total order* on all transactions. This set of nodes establishes consensus on transactions, creates blocks, and broadcasts them to every node. Finally, each peer validates a transaction within a received block by checking the endorsement policy and read-write conflicts and then updates the ledger. Since a validation phase occurs at the end, the paradigm is called *execute-order-validate*. Figure 1(c) presents the flow of transactions in Fabric. Note that in Fabric the consensus protocol is pluggable and the system can use a crash fault-tolerant protocol, e.g., Paxos [27], a Byzantine fault-tolerant protocol, e.g., PBFT [13], or any other protocol.

While Fabric solves the confidentiality issue by executing each transaction on a specified subset of peers (endorsers) and increases the performance of blockchains by executing the transactions in parallel (instead of sequentially as the order-execute paradigm does), it performs poorly on workloads with high-contention, i.e., many *conflicting transactions* in a block, due to its high abort rate.

Two transactions *conflict* if they access the same data and one of them is a write operation. In such a situation, the order of executing the transactions is important, indeed, the later transaction in a block has to wait for the earlier transaction to be executed first. As a result, if two conflicting transactions execute in parallel, the result is invalid. Although Fabric guarantees correctness by checking the conflicts in the validation phase (the last phase) and disregarding the effects of invalid transactions, the performance of the blockchain is highly reduced by such conflicts.

III. The OXII Paradigm

In this section, we introduce OXII, a new order-execute paradigm for permissioned blockchains. OXII is mainly designed to support distributed applications with high-contention workloads.

OXII consists of a set of nodes in an asynchronous distributed network where each node has one of the following roles:

- *Clients* send operations to be executed by the blockchain.
- *Orderers* agree on a total order of all transactions.
- *Executors* validate and execute transactions.

The set of nodes in OXII is denoted by N where O of them are orderers, and E of them are executors.

OXII supports distributed applications running concurrently on the blockchain. For each application a program code including the logic of that application (*smart contract*) is installed on a (non-empty) subset of executor peers called the *agents* of the application. We use $\mathcal{A} = \{A_1, ..., A_n\}$ to denote the set of applications (ids) and $\Sigma(A_i)$ to specify the non-empty set of agents of each application A_i where $\Sigma : \mathcal{A} \mapsto 2^E - \emptyset$. Every peer in the blockchain knows the agents of each application and the set of orderers.

Each pair of peers is connected with point-to-point bi-directional communication channels. Network links are pair-wise authenticated, which guarantees that a Byzantine node cannot forge a message from a correct node, i.e., if node i receives a message m in the incoming link from node j, then node j must have sent message m to i beforehand.

A. Orderers

Checking accesses, ordering the requests, constructing blocks, generating dependency graphs, and multicasting the blocks are the main services of orderers in the OXII paradigm.

Since multiple applications run on the blockchain and each application might have its own set of clients, orderers act as trusted entities to restrict the processing of requests that are

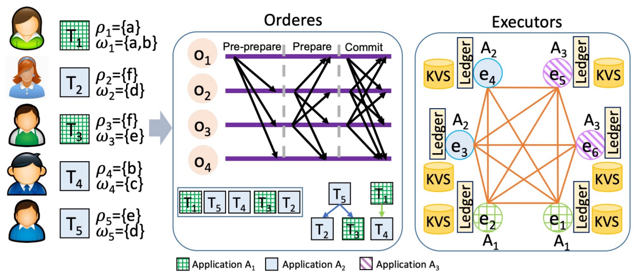


Fig. 2. The Components of OXII Paradigm

sent by unauthorized clients. If a client is not authorized to perform an operation on the requested application, orderers simply discard that request. Orderers also check the signature of the requests to ensure their validity.

Orderers use an asynchronous fault-tolerant protocol to establish consensus. Fault-tolerant protocols use the state machine replication algorithm [26] where replicas agree on an ordering of incoming requests. The algorithm has to satisfy two main properties, (1) *safety*: all correct nodes receive the same requests in the same order, and (2) *liveness*: all correct client requests are eventually ordered. Fischer et al. [19] show that in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility result, most fault-tolerant protocols satisfy safety without any synchrony assumption and consider a synchrony assumption to satisfy liveness.

OXII, similar to Fabric [5], uses a pluggable consensus protocol for ordering, thus resulting in a modular paradigm. Depending on the characteristics of the network and peers OXII might employ a Byzantine, a crash, or a hybrid fault-tolerant protocol. The number of orderers is also determined by the utilized protocol and the maximum number of simultaneous failures in the network. For example, crash fault-tolerant protocols, e.g., Paxos [27], guarantee safety (consistency) in an asynchronous network using $2f+1$ nodes to overcome the simultaneous crash failure of any f nodes while in Byzantine fault-tolerant protocols, e.g., PBFT [13], $3f+1$ nodes are needed to provide safety in the presence of f malicious nodes. Furthermore, orderers do not have access to any smart contract or the application state, nor do they participate in the execution of transactions. This makes orderers independent of the other peers and adaptable to a changing environment.

Orderers batch multiple transactions into blocks. Batching transactions into blocks improves the performance of the blockchain by making data transfers more efficient especially in a geo-distributed setting. It also amortizes the cost of cryptography. The batching process is *deterministic* and therefore produces the same blocks at all orderers.

Figure 2 shows the components of the OXII paradigm. As can be seen, clients send requests (transactions) to be executed by different applications. Here, transactions T_1 and T_3 are for some application A_1 and T_2, T_4, and T_5 are for another application A_2. The orderers, i.e., o_1, o_2, o_3, and o_4, then order the transactions and put them into a block. In the figure, orderers use PBFT [13] to order the requests. The resulting block contains five transactions which are ordered as $[T_1, T_5, T_4, T_3, T_2]$.

Next, orderers generate a "dependency graph" for the transactions within a block. In order to generate dependency graphs a priori knowledge of transactions' read- and write-set is needed. Each transaction consists of a sequence of reads and writes, each accessing a single record. Here we assume that the read-set and write-set are pre-declared or can be obtained from the transactions via a static analysis, e.g., all records involved in a transaction are accessed by their primary keys. Note that even if that assumption does not hold, the system can employ other techniques like speculative execution [18] to obtain the read-set and write-set of each transaction.

Given a transaction T, $\omega(T)$ and $\rho(T)$ are used to represent the set of records written and read, respectively. Each transaction T is also associated with a timestamp $ts(T)$ where for each two transactions T_i and T_j within a block such that T_i appears before T_j, $ts(T_i) < ts(T_j)$.

We define "ordering dependencies" to show possible conflicts between two transactions from the same or different applications. Two transactions conflict if they access the same data and one of them is a write operation.

Definition: Given two transactions T_i and T_j. An *ordering dependency* $T_i \succ T_j$ exists if and only if $ts(j) > ts(i)$ and one of the following hold:

- $\rho(T_i) \cap \omega(T_j) \neq \varnothing$
- $\omega(T_i) \cap \rho(T_j) \neq \varnothing$
- $\omega(T_i) \cap \omega(T_j) \neq \varnothing$

Definition: Given a block of transactions, the *dependency graph* of the block is a directed graph $G = (\mathcal{T}, \mathcal{E})$ where $\mathcal{T}$ is the set of transactions within the block and $\mathcal{E} = \{(T_i, T_j) \mid T_i \succ T_j\}$

We use the example in Figure 2 to illustrate the dependency graph construction process. As can be seen the block consists of five transactions which are ordered as $[T_1, T_5, T_4, T_3, T_2]$, i.e., $ts(T_1) < ts(T_5) < ts(T_4) < ts(T_3) < ts(T_2)$. Since T_4 reads data item b which is written by an earlier transaction T_1 there is an ordering dependency $T_1 \succ T_4$, thus (T_1, T_4) is an edge of the dependency graph. Similarly, T_2 writes data item d which is also written by T_5 ($T_5 \succ T_2$) and T_3 writes data item e which is read by T_5 ($T_5 \succ T_3$). As a result, edges (T_5, T_2) and (T_5, T_3) are also in the graph.

The constructed graph can be used by the executors to manage the execution of transactions. In particular, transactions that are not connected to each other in the dependency graph, e.g., T_1 and T_2, can be processed concurrently by independent execution threads.

The dependency graph generator is an independent module in the OXII paradigm. Therefore, it can also be adapted to a multi-version database system [8]. In a multi-version database, each write creates a new version of a data item, and reads are directed to the correct version based on the position of the corresponding transaction in the block (log). Since writes do not overwrite each other, the system has more flexibility to manage the order of reads and writes. As a result, for any two transactions T_i and T_j within a block where T_i appears before T_j, T_i and T_j can concurrently write the same data item or T_i reads and T_j writes the same data item. However, if T_i wants to write and T_j wants to read the same data item, they cannot be executed in parallel.

It should be noted that in some dependency graph construction approaches, e.g., DGCC [33], transactions are broken down into transaction components, which allows the system to parallelize the execution at the level of operations. The dependency graph generator module in OXII can also be designed in a similar manner.

A dependency graph exposes conflicts between transactions to give a partial order of transactions. Hence, as long as the transactions are executed in an order consistent with the dependency graph, the results are valid. Indeed, using dependency graphs results in higher concurrency by enabling the non-conflicting transactions within a block to be executed in parallel. Such parallelism improves the performance of OXII paradigm in comparison to the traditional order-execute paradigm where transactions are executed sequentially.

When the dependency graph is generated, orderers multicast a message including the block and its dependency graph to all executors. Depending on the employed consensus protocol, either the leader or all the orderers multicast the message.

B. Executors

Executing and validating transactions, updating the ledger and the blockchain state, and multicasting the blockchain state after executing transactions are the main services of executor peers. Executors in OXII correspond to the endorsers in Hyperledger [5]. Each executor peer maintains three main components: (1) The blockchain ledger, (2) The blockchain state, and (3) Some smart contracts.

The blockchain ledger is an append-only data structure recording all transactions in the form of a hash chain. When a block of transactions is executed and validated, each executor peer appends the block to its copy of the ledger.

Each executor node is an agent for one or more applications where for each application the smart contract of that application, i.e., a program code that implements the application logic, is installed on the node. When an executor receives a block from the orderers, it checks the application of the transactions within the block. If the executor is an agent for any of the transactions, it executes the transactions on the corresponding smart contract following the dependency graph. In fact, the executor confirms the order of dependent transactions and executes independent transactions in parallel. Finally, it multicasts the execution results (updated blockchain state) to all other peers.

For each transaction within a block where the executor is not an agent of the transaction, the executor waits for matching updates from a specified number of executors, who are the agents of the transaction, before committing the update. This is needed to prevent a malicious executor to commit an invalid result and also tolerate the non-deterministic execution of transactions. The required number of matching results from executors is decided by the system and known to all executors (similar to endorsement policies in Hyperledger). We use $\tau(A)$ to denote the required number of matching updates for the transactions of application A.

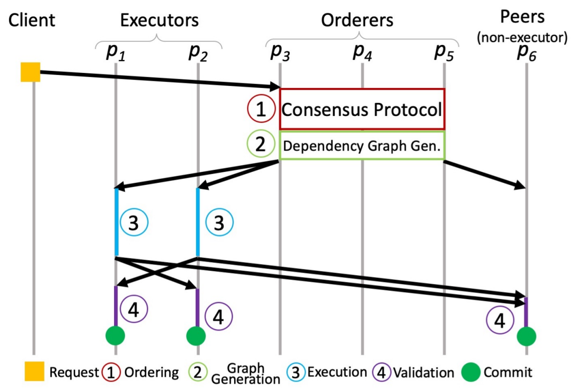


Fig. 3. The Flow of Transactions in ParBlockchain

In Figure 2, executor nodes e_1 and e_2 are the agents of application A_1 (with transactions T_1 and T_3) and executor nodes e_3 and e_4 are the agents of application A_2 (with transactions T_2, T_4 and T_5).

IV. ParBlockchain

In this section, we present ParBlockchain, a permissioned blockchain designed specifically in the OXII paradigm. We first give a summary of ParBlockchain and then explain the ordering and execution phases in detail.

A. ParBlockchain Overview

ParBlockchain is a permissioned blockchain designed in the OXII paradigm to execute distributed applications.

The normal case operation for ParBlockchain to execute transactions proceeds as follows. Clients send requests to the orderers and the orderers run a consensus algorithm among themselves to reach agreement on the order of transactions. Orderers then construct a block of transactions and generate a dependency graph for the transactions within the block.

Once the dependency graph is generated, the block along with the graph is multicast to all the executor nodes. The executors which are the agents of the applications of transactions within the block, execute the corresponding transactions and multicast the results, i.e., updated records in the datastore, to every executor node. Each executor node in the network waits for the required number of matching results from the executors before updating the ledger and blockchain state (datastore). The required number of matching results for each application, which is needed to deal with malicious executors and the non-deterministic execution of transactions, is determined by the system and might be different for different applications.

The flow of transactions in ParBlockchain can be seen in Figure 3 where p_3, p_4, and p_5 are the orderer nodes, and p_1, p_2, and p_6 are the executor nodes from which p_1 and p_2 are the agents for the requests. Upon receiving requests from clients, orderers order the requests, put them into a block, generate the dependency graph for the block, and multicast the block along with the graph to all the executor nodes. The agents of the corresponding application (p_1 and p_2) execute the transactions and multicast the updated state of the blockchain to the other executor nodes. Upon receiving the required number of matching messages for each transaction, each executor commits (or aborts) the transaction by updating the blockchain state. The block is also appended to the ledger.

B. Ordering Phase

The goal of the ordering phase is to establish a total order on all the submitted transactions. A client c requests an operation op for some application A by sending a message $\langle \text{REQUEST}, op, A, ts_c, c\rangle_{\sigma_c}$ to the orderer p it believes to be the *primary* (an orderer node that initiates the consensus algorithm). Here, ts_c is the client's timestamp and the entire message is signed with signature σ_c. We use timestamps of clients to totally order the requests of each client and to ensure exactly-once semantics for the execution of client requests.

Upon receiving a client request, the primary orderer p checks the signature to ensure it is valid, makes sure the client is allowed to send requests for application A (access control), and then initiates a consensus algorithm by multicasting the request to other orderers. Depending on the utilized consensus protocol, several rounds of communication occurs between orderers to establish a total order on transactions.

Once the orderers agree on the order of a transaction, they put the transaction in a block. Batching multiple transactions into blocks improves the throughput of the broadcast protocol. Blocks have a pre-defined maximal size, maximal number of transactions, and maximal time the block production takes since the first transaction of a new block was received. When any of these three conditions is satisfied, a block is full. Since transactions are received in order, the first two conditions are deterministic. In the third case, to ensure that the produced blocks by all orderers are the same, the primary sends a *cut-block* message in the consensus step of the last request.

When a block is produced, orderers generate a dependency graph for the block as explained in Section III-A. Generating dependency graphs requires a priori knowledge of transactions' read- and write-set. Here, we assume that the requested operations include the read- and write-set.

When the graph is constructed, each orderer node o multicasts a message $\langle \text{NEWBLOCK}, n, B, G(B), \mathbf{A}, o, h\rangle_{\sigma_o}$ to all executor nodes where n is the sequence number of the block, B is the block consisting of the request messages, $G(B)$ is the dependency graph of B, $\mathbf{A}$ is the set of applications that have transactions in the block, and $h = H(B')$ where $H(.)$ denotes the cryptographic hash function and B' is the block with sequence number $n-1$.

C. Execution Phase

Each request for an application is executed on the specified set of executors, i.e., agents of that application. Upon receiving a new block message $\langle \text{NEWBLOCK}, n, B, G(B), \mathbf{A}, o, h\rangle_{\sigma_o}$ from some orderer o, executor e checks the signature and the

Algorithm 1 Execution of Transactions on an executor e

```
Input: A block $B$ and its dependency graph $G(B)$
 1: Initiate Set $W_e$ to be empty
 2: for transaction $x$ in $B$ do
 3:     if $e$ is an agent of $x$'s application then
 4:         Add $x$ to $W_e$
 5:     end if
 6: end for
 7: while $W_e$ in not empty do
 8:     for transaction(node) $x$ in $W_e$ do
 9:         if all $Pre(x)$ are in $C_e \cup X_e$ then
10:             trigger Execute($x$)
11:         end if
12:     end for
13: end while
```

hash to be valid and logs the message. It also checks the set $\mathbf{A}$ to see if the block contains any transaction that needs to be executed by the node, i.e., an application $A_i \in \mathbf{A}$ such that $e \in \Sigma(A_i)$.

When an executor node receives a specified number of matching new block messages, e.g., $f + 1$ messages if the consensus protocol is PBFT, it marks the new block as a valid block and enters the execution phase. The execution phase consists of three procedures that are run concurrently: (1) Executing the transaction following the dependency graph, (2) Multicasting commit messages including the execution results to other executor nodes, and (3) Updating the blockchain state upon receiving commit messages from a sufficient number of executor nodes.

If an executor node is not an agent of any transaction within the block, the node becomes a *passive* node and only the third procedure is run to update the blockchain state. However, if a node is an agent for some transaction's application in the block, it runs all three procedures; executes the corresponding transactions following the dependency graph, multicasts the results, and also updates the blockchain state.

A transaction can be executed only if all of its "predecessors" in the dependency graph are committed. We define functions Pre and Suc to present the set of predecessors and successors of a node in a dependency graph respectively. More formally, Given a dependency graph $G = (\mathcal{T}, \mathcal{E})$, and a node (transaction) x in $\mathcal{T}$, $Pre(x) = \{y \mid (y, x) \in \mathcal{E}\}$ and $Suc(x) = \{y \mid (x, y) \in \mathcal{E}\}$.

The execution procedure on a node e is shows in Algorithm 1. An empty set W_e is initiated to keep all the transactions that will be executed by executor e, i.e., e is an agent for the application of those transactions. Set X_e stores the executed transactions by e and C_e keeps the committed transactions. For each transaction x in W_e, the procedure checks the predecessors of x, if x has no predecessor, or all of its predecessors are executed by e or committed, transaction x is ready to be executed, so an execution thread is triggered.

To multicast the execution results depending on the transactions' applications three different situations could happen. If all the transactions within a block belong to the same application, an agent e executes all of the transactions following the dependency graph and multicast a *commit* message $\langle \text{COMMIT}, S, e\rangle_{\sigma_e}$ to all other executor nodes. Here,

Algorithm 2 Multicasting the Results

```
Initialize set $X_e$ to be empty
$cut$ = false
Upon obtaining an execution result $(x, r)$
    Add pair $(x, r)$ to $X_e$
    Remove $x$ from $W_e$
for $y$ where $(x, y)$ is an edge in $G(B)$ do
    if $y$'s application is different from $x$'s application then
        $cut$ = true
        break
    end if
end for
if $cut$ = true then
    Multicast $\langle \text{COMMIT}, X_e, e \rangle_{\sigma_e}$ to all executors
    Clear $X_e$
end if
```

Algorithm 3 Updating the Blockchain State

```
for transaction $x$ in $B$ do
    Initialize set $R_e(x)$ to be empty
end for
Initialize set $C_e$ to be empty
Upon Receiving a valid $\langle \text{COMMIT}, S, n \rangle_{\sigma_n}$ message
for valid $(x, r) \in S$ do
    Add $(r, n)$ to $R_e(x)$
    if Matching records in $R_e(x) \geq \tau(A)$ then
        Update the blockchain state
        Add $x$ to $C_e$
    end if
end for
```

S presents the state of the blockchain and consists of a set of pairs (x, r) where x is a transaction (id) and r is the set of updated records resulting from the execution of x on the datastore. Note that if a transaction x is not valid, the executor puts $(x,$"abort"$)$ in S.

If the transactions within a block are for different applications but the transactions of each application access a disjoint set of records, the agents still can execute the corresponding transactions independently and multicast a single commit message with all the results to other executor nodes. In this case, the dependency graph is disconnected and can be decomposed to different components where the transactions of each component are for the same application and there is no edge that connects any two components.

However, if there are some dependencies between the transactions of two applications, the agents of those two applications cannot execute the transactions independently. In fact, the agents of one application have to wait for the agents of other applications to execute all their transactions and send the commit message which might result in a deadlock situation.

Figure 4 shows three dependency graphs for a block of seven transactions T_1 to T_7. In Figure 4(a), all the seven transactions belong to the same application, A_1. Therefore, the agents of application A_1 can execute the transactions following the dependency graph and multicast the results of all transactions together when they all are executed. In Figure 4(b) although the transactions belong to different applications (T_2, T_3, T_5, and T_7 are for application A_1 and T_1, T_4 and T_6 are for application A_2), there is no dependency between the transactions of application A_1 and the transactions of application A_2. As a result, the agents can still execute independently and multicast the results once the execution of their transactions is completed. However, in Figure 4(c) since there are some dependencies between the transactions of the two applications, the agents cannot execute their transactions independently. For example, to execute transaction T_2, the agents of application A_2 need the execution results of transaction T_5 from the agents of A_1. Similarly, transaction T_4 cannot be executed before committing the execution results of transaction T_6.

To prevent a deadlock situation, one possibility is that agents send a commit message as soon as the execution of each transaction is completed. While this approach solves the blocking problem, the number of exchanged commit messages will be large. Indeed, if a block includes n transactions and each application has on average m agents, there will be total $n * m$ exchanged commit messages for the block.

A more efficient way is to send commit messages when the execution results are needed by some other agents. Basically, an agent keeps executing the transactions and collecting the results until the results of an executed transaction is needed by some other transactions which belong to other applications. At that time, the agent generates a commit message including the results of all the executed transactions and multicasts it to all executor nodes. Upon receiving a commit message from an executor, the node validates the signature and logs the message. Once the node receives the specified number of matching results for a transaction, the results are reflected in the datastore and the transaction is marked as committed.

Algorithm 2 presents the multicasting procedure on a node e. An empty set X_e is initiated to store the results of the executed transactions. When the execution of a transaction x is completed, the execution result (x, r) is added to X_e and transaction x is removed from the waiting transactions W_e.

Then, the procedure checks all the successor nodes of x in the dependency graph. If any of the successor nodes of x belongs to an application different from the application of x, the execution result of transaction x might be needed by other agents, thus a multicasting has to occur. To do so, node e removes all the stored results from X_e and puts them in a commit message and multicast the commit message to all other executor nodes.

For example, in Figure 4(c), upon executing the transaction T_5, since T_5 has a successor node T_2 that belongs to another application, the executor node multicasts a commit message including the execution results of T_5 to all other executor nodes. Note that if T_1 is already executed, the executor node puts the execution results of T_1 in the commit message as well. Similarly, when the execution of T_6 is completed, the executor node multicasts a commit message including the execution results of T_6 and any other executed but not yet multicast transactions.

Finally, the updating procedure receives commit messages from other executor nodes and updates the blockchain state. The updating procedure on a node e is presented in Algorithm 3. the procedure first initializes an empty set $R_e(x)$ for each transaction x in the block. It also initializes an empty set

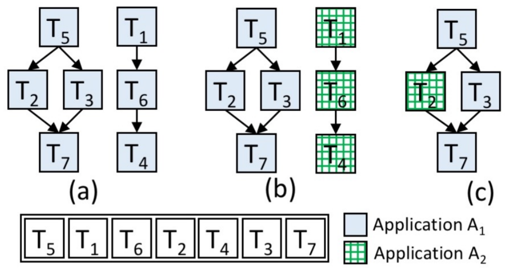


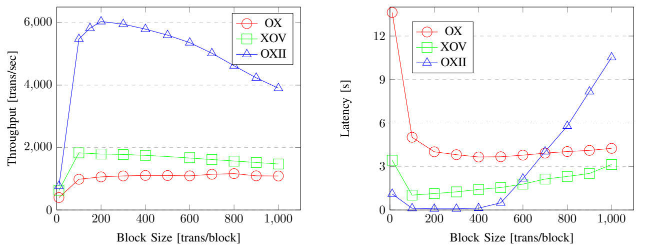
Fig. 4. Three Dependency Graphs

C_e to collect committed transactions. When node e receives a commit message $\langle \text{COMMIT}, S, n\rangle_{\sigma_n}$ from some executor n, it checks the signature to be valid and then checks the set S. Recall that S consists of pairs of transactions and their execution results. For each pair (x, r), it first checks whether node n is an agent for the application of transaction x and then a pair of (r, n), i.e., execution results and the executor to $R_e(x)$. Assuming A is the x's application, If the number of the matching tuples in $R_e(x)$ is equal to $\tau(A)$, i.e., the specified number of messages for the transaction's application, the execution results are valid and can be committed. As a result, the procedure updates the blockchain state (datastore) and adds the transaction x to the committed transactions C_e.

V. Experimental Evaluations

In this section, we conduct several experiments to evaluate different paradigms for permissioned blockchains. We discussed the two existing paradigms for permissioned blockchains in Section II: sequential order-execute (OX) where requests are ordered and then executed sequentially on every node, and execute-order-validate (XOV) introduced by Hyperledger Fabric [5] where requests are executed by the agents of each application, ordered by the ordering service, and validated by every peer. We implemented two permissioned blockchain systems specifically designed in the OX and XOV paradigms as well as ParBlockchain that is designed in the OXII paradigm. It should be noted that our implementation of XOV is different from the Hyperledger fabric system. Hyperledger is a distributed operating system and includes many components which are not the focus of our evaluations. In fact, the purpose of our experiments is to compare the architectural aspect of the blockchain systems, thus, all three systems are implemented using the same programming language (*Java*). To have a fair comparison, we also used similar libraries and optimization techniques for all three systems as far as possible.

We implemented a simple accounting application where each client has several accounts. Each account can be seen as a pair of $(amount, PK)$ where PK is the public key of the owner of the account. Clients can send requests to transfer assets from one or more of their accounts to other accounts. For example, a simple transaction T initiated by client c might "transfer x units from account 1001 to account 1002". The transaction is valid if c is the owner of account 1001 and the account balance is at least x. Here the read-set of transaction T is $\rho(T) = \{1001\}$ and its write-set is $\omega(T) = \{1001, 1002\}$. A transaction might read and write several records.



(a) Throughput (b) Latency

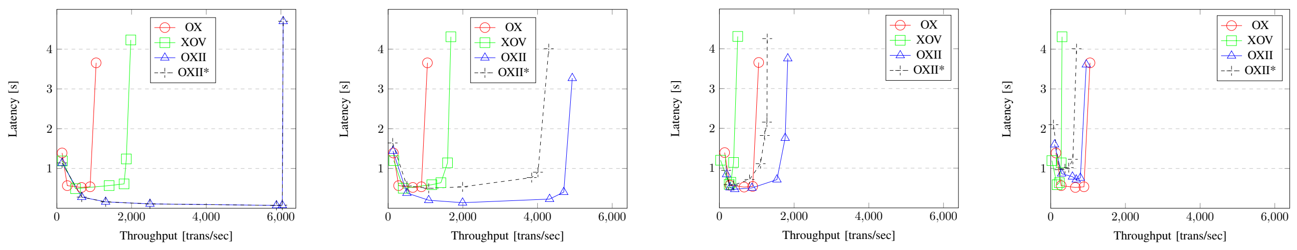
Fig. 5. Throughput/Latency Measurement by Increasing the Block Size

The experiments were conducted on the Amazon EC2 platform. Each VM is Compute Optimized c4.2xlarge instance with 8 vCPUs and 15GM RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. For orderers, similar to Hyperledger [5], we use a typical Kafka orderer setup with 3 ZooKeeper nodes, 4 Kafka brokers and 3 orderers, all on distinct VMs. Unless explicitly mentioned differently, there are three applications in total each with a separate executor (endorser) node.

When reporting throughput measurements, we use an increasing number of clients running on a single VM, until the end-to-end throughput is saturated, and state the throughput just below saturation. Throughput numbers are reported as the average measured during the steady state of an experiment.

A. Choosing the Block Size

An important parameter that impacts both throughput and latency is the block size. To evaluate the impact of the block size on performance, in this set of experiments, assuming that the transactions have the same size, we increase the number of transactions in each block from 10 to 1000 in a no-contention workload. For each block size, the peak throughput and the corresponding average end-to-end latency is measured. As can be seen in Figure 5, by increasing the number of transactions per block till $\sim$200, the throughput of OXII increases, however, any further increasing reduces the throughput due to the large number of required computations for the dependency graph generation. Similarly, by increasing the number of transactions per block till $\sim$200, the delay decreases. Afterward, adding more transactions to the dependency graph becomes more time consuming than multicasting the block. As a result, OXII is able to process more than 6000 transactions in $78ms$ with 200 transactions per block. In the OX paradigm, since nodes execute transactions sequentially, the block creation time is negligible in comparison to the execution time, thus other than in the first experiment, increasing the number of transactions per block does not significantly affect the throughput and latency. In the XOV paradigm, since executors (endorsers) of the three applications can execute the transactions in parallel, the performance is better than OX (twice as much as OX in its peak throughput). However, its performance is still much less than OXII, i.e., the peak throughput of XOV is 30% of the peak throughput of OXII as OXII can execute many (and not only three) non-conflicting transactions in parallel. As can be seen, the peak throughput of XOV is obtained in $\sim$100 transactions per block.



(a) No-contention (0%) (b) Low-contention (20%) (c) High-contention (80%) (d) Full-contention (100%)

Fig. 6. Throughput/Latency Measurement by Increasing the Degree of Contention in the Workload

B. Performance in Workloads with Contention

In the next set of experiments, we measure the performance of all three paradigms for workloads with different degrees of contention. we consider no-contention, low-contention (20% conflict), high-contention (80% conflict), and full-contention workloads where the results are shown in Figure 6(a)-(d) respectively. Note that the dependency graph of each block in the first workload has no edge whereas the dependency graph of each block in the last workload is a chain. In OX and OXII there are 200 transactions per block and for XOV, we keep changing the block size to find its peak throughput. Contentions could happen between the transactions of the same application or the transactions of different applications (if they access shared data). In OX, since nodes execute transactions sequentially, there is no difference between these two types of contention. In XOV also, since the execution is the first phase, there is no much difference between contention within an application or across applications and they both result in transaction abort. In OXII, however, as discussed in Section IV-C, for contention across applications, the agents of different applications communicate to each other during the execution of a block of transactions, thus the performance is affected. As a result, in this set of experiments, for each workload, we report the performance of OX, XOV, OXII with conflicting transactions within an application, and OXII with conflicting transactions across applications (the dashed line).

As mentioned earlier, in the OX paradigm, transactions are executed sequentially. As a result, the performance of OX remains unchanged in different workloads. XOV can execute 3 (number of applications) transactions in parallel and since the workload has no-contention, the execution results are valid. OXII, on the other hand, significantly benefits from no-contention workloads by executing the transactions in parallel. As shown in Figure 6(a), OXII executes more than 6000 transactions with latency less than 80 ms whereas the peak throughput of OX is 900 transactions with more than 500 ms latency. XOV can also execute around 1800 transactions in 600 ms (70% less throughput and 7.5 times latency in comparison to OXII). Since the workload has no conflicting transaction, there is no contention across applications.

By increasing the degree of contention (Figure 6(b) and Figure 6(c)), the throughput of XOV decreases dramatically, e.g., the peak throughput of XOV in a high-contention workload is around 25% of its peak throughput in a no-contention workload. This decrease is expected because XOV validates and aborts the conflicting transactions at the very end (last phase). The throughput of OXII is also affected by increasing the degree of contention, however, it still shows better performance than both OX and XOV, i.e., OXII is still able to process 1600 transactions in sub-second latency whereas OX and XOV process 900 and 350 transactions respectively. Processing the workloads with contention across the applications decreases the performance of OXII due to the increasing rounds of communication between executors of different applications.

In a full-contention workload, as can be seen in Figure 6(d), OXII similar to OX, executes the transactions sequentially, but, because of the dependency graph generation overhead, its performance is a bit worse than OX. The performance of the XOV paradigm, on the other hand, is highly reduced. Since all the transactions within a block conflict, it can only commit one transaction per block (we reduced the block size of XOV to record its peak throughput).

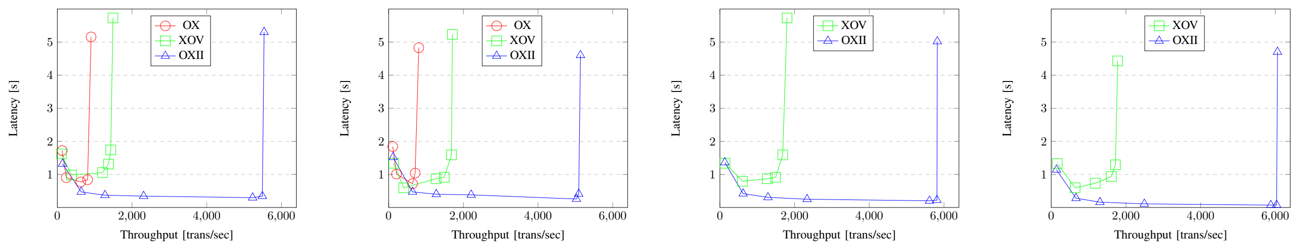
In a full-contention workload with contention across applications (dashed line in Figure 6(d)), OXII has high latency and low throughput. Such a workload can be seen as a chain of translations where consecutive transactions belong to different applications. As a result, to execute each transaction, a message exchange between a pair of executors is needed.

C. Scalability over Multiple Data Centers

In the last set of experiments, we measure the scalability of the blockchain systems over multiple data centers. To this end, each time we move one group of nodes, i.e., clients, orderers, executors, or non-executors, to AWS Asia Pacific (Tokyo) Region data center, leaving the other nodes in the AWS US West Region data center (the RTT between these two data centers is 113 ms). We consider a no-contention workload. The results can be seen in Figure 7.

Moving the clients has the most impact on the XOV paradigm because in XOV clients participate in the first two phases. Indeed, they send the requests to the executors (endorsers), receive endorsements, and then send the endorsements to the orderer nodes. Whereas in OX and OXII, clients send the requests and do not participate in other phases of the protocol. As a result, as can be seen in Figure 7(a), the delay of XOV becomes much larger.

Orderers are the core part of all three blockchains; they receive transactions from clients, agree on the order of the transactions, put the transactions into blocks, and send the blocks to every node. As a result, moving them to a far data



(a) Apart Client

(b) Apart Orderers

(C) Apart Executors

(d) Apart Non-executors

Fig. 7. Throughput/Latency Measurement by Moving a Group of Nodes to a Further Data Center

center, as shown in Figure 7(b), results in a considerable delay. Note that in OX, a subset of nodes are considered as orderers.

In the last two experiments (Figure 7(c)-(d)), we move executor (endorser) and non-exe nodes to the far data center. Since there is no such a separation between nodes in the OX paradigm, we do not perform these two experiments. Moving executor nodes adds latency to the two phases of communication in XOV (clients to executors and executors to clients) and one phase of communication in OXII (orderers to executors). Note that when the executors execute the messages and receive enough number of matching results from other executors, the transaction is counted as committed. In addition, no communication between executors is needed since we consider a no-contention workload. Finally, moving non-executor nodes has no impact on the performance of OXII, because those nodes are only informed about the blockchain state. But in XOV, non-executors validate the blocks.

VI. Related Work

The Order-execute paradigm is widely used in different permissioned blockchains. Existing permissioned blockchains that employ the order-execute paradigm, differ mainly in their ordering routines. The ordering protocol of Tendermint [25] differs from the original PBFT in two ways, first, only a subset of nodes participate in the consensus protocol and second, the leader is changed after the construction of every block (leader rotation). Quorum [14] as an Ethereum-based [3] permissioned blockchain introduces a consensus protocol based on Raft [29]: a well-known crash fault-tolerant protocol. Chain Core [1], Multichain [21], Hyperledger Iroha [4], and Corda [2] are some other prominent permissioned blockchains that follow the order-execute paradigm. As discussed in Section II, these permissioned blockchains mainly suffer from performance and confidentiality issues. Hyperledger Fabric [5] is a permissioned blockchain that employs the execute-order(-validate) paradigm introduced by Eve [22]. Fabric presents modular design, pluggable fault-tolerant protocol, policy-based endorsement, and non-deterministic execution for the first time in the context of permissioned blockchains. Several recent studies attempt to improve the performance of Fabric [20], [30]–[32].

We utilize some of the Fabric properties such as modular design and pluggable fault-tolerant protocol in OXII. However, OXII is an *order-execute* paradigm. In addition, while Fabric checks the read-write conflict in the last phase (validation) which might result in transaction abort, OXII ensures correct results by generating dependency graphs in the first phase (ordering). As a result, workloads with contention benefit most from OXII. Fabric also needs four phases of communications other than the ordering protocol (clients to endorsers, endorsers to clients, clients to orderers, and orderers to peers) while OXII requires three phases (clients to orderers, orderers to executors, executors to peers) which results in less latency. It should be noted that since execution is the first phase of Fabric, in comparison to OXII, its performance is less affected by the inconsistencies between the execution results (arise from malicious executors or non-deterministic execution).

In domain of permissionless blockchains also, concurrent speculative execution of smart contracts in Ethereum using software transactional memory primitives is proposed in [15].

Our work is also related to concurrency control in DBMS. Concurrency control is the activity of coordinating concurrent accesses to data [7]. Concurrency control protocols mainly ensure the atomicity and isolation properties. Many techniques have been proposed for concurrency control. Lock-based protocols, e.g., two phase locking (2PL) [10], [16], use locks to control the access to data. Timestamp-based protocols [7], [9] assign a global timestamp before processing where by ordering the timestamp, the execution order of transactions is determined. Optimistic Concurrency Control (OCC) [24] and Multi-Version Concurrency Control (MVCC) [8] are two widely used timestamp-based protocols. Dependency graphs are also, as discussed in section III, used by several recent studies for concurrency control [17], [18], [33].

VII. Conclusion

In this paper, we proposed OXII, an order-execute paradigm for permissioned blockchain to support distributed applications that execute concurrently. OXII is able to handle the workload with conflicting transactions without rolling back the processed transactions or executing transactions sequentially. Conflicts between the transactions of a single application as well as the transactions of different applications are addressed in OXII. We also presented ParBlockchain, a permissioned blockchain system designed specifically in the OXII paradigm. Our experimental evaluations show that in workloads with conflicting transactions, ParBlockchain shows a better performance in comparison to both order-execute and execute-order permissioned blockchain systems.

Acknowledgement

This work is funded by NSF grants CNS-1703560 and CNS-1815733.

References

[1] Chain. http://chain.com.

[2] Corda. https://github.com/corda/corda.

[3] Ethereum blockchain app platform. https://www.ethereum.org.

[4] Hyperledger iroha. https://github.com/hyperledger/iroha.

[5] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[6] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *International Conference on Open and Big Data (OBD)*, pages 25–30. IEEE, 2016.

[7] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[8] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control-theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[9] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. 1987.

[10] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[11] C. Cachin. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, volume 310, 2016.

[12] C. Cachin and M. Vukolić. Blockchain consensus protocols in the wild. In *31 International Symposium on Distributed Computing, DISC*, pages 1–16, 2017.

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

[14] JP Morgan Chase. Quorum white paper, 2016.

[15] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312. ACM, 2017.

[16] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[17] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, 2015.

[18] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 10(5):613–624, 2017.

[19] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[20] Christian Gorenflo, Stephen Lee, Lukasz Golab, and S. Keshav. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. *arXiv preprint arXiv:1901.00910*, 2019.

[21] Gideon Greenspan. Multichain private blockchain-white paper. *URl: http://www. multichain. com/download/MultiChain-White-Paper. pdf*, 2015.

[22] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about eve: Execute-verify replication for multi-core servers. In *OSDI*, volume 12, pages 237–250, 2012.

[23] K. Korpela, J. Hallikas, and T. Dahlberg. Digital supply chain transformation toward blockchain integration. In *proceedings of the 50th Hawaii international conference on system sciences*, 2017.

[24] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[25] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 2014.

[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[27] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[28] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[29] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[30] Ravi Kiran Raman, Roman Vaculin, Michael Hind, Sekou L Remy, Eleftheria K Pissadaki, Nelson Kibichii Bore, Roozbeh Daneshvar, Biplav Srivastava, and Kush R Varshney. Trusted multi-party computation and verifiable simulations: A scalable blockchain approach. *arXiv preprint arXiv:1809.08438*, 2018.

[31] Joao Sousa, Alysson Bessani, and Marko Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58. IEEE, 2018.

[32] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. *arXiv preprint arXiv:1805.11390*, 2018.

[33] Chang Yao, Divyakant Agrawal, Pengfei Chang, Gang Chen, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. Dgcc: A new dependency graph based concurrency control protocol for multicore database systems. *arXiv preprint arXiv:1503.03642*, 2015.