A Transactional Perspective on Execute-order-validate Blockchains

Pingcheng Ruan
National University of Singapore
ruanpc@comp.nus.edu.sg

Dumitrel Loghin
National University of Singapore
dumitrel@comp.nus.edu.sg

Quang-Trung Ta
National University of Singapore
taqt@comp.nus.edu.sg

Meihui Zhang
Beijing Institute of Technology
meihui_zhang@bit.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

Beng Chin Ooi
National University of Singapore
ooibc@comp.nus.edu.sg

ABSTRACT

Smart contracts have enabled blockchain systems to evolve from simple cryptocurrency platforms to general transactional systems. A new architecture called execute-order-validate has been proposed in Hyperledger Fabric to support parallel transactions. However, this architecture might render many invalid transactions when serializing them. This problem is further exaggerated as the block formation rate is inherently limited due to other factors beside data processing, such as cryptography and consensus.

Inspired by optimistic concurrency control in modern databases, we propose a novel method to enhance the execute-order-validate architecture, by reordering transactions to reduce the abort rate. In contrast to existing blockchains that adopt database's preventive approaches which might over-abort serializable transactions, our method is theoretically more fine-grained: unserializable transactions are aborted before reordering and the rest are guaranteed to be serializable. We implement our method in two blockchains respectively, FabricSharp on top of Hyperledger Fabric, and FastFabricSharp on top of FastFabric. We compare the performance of FabricSharp with vanilla Fabric and three related systems, two of which are respectively implemented with one standard and one state-of-the-art concurrency control techniques from databases. The results demonstrate that FabricSharp achieves 25% higher throughput compared to the other systems in nearly all experimental scenarios. Moreover, the FastFabricSharp's improvement on FastFabric is up to 66%.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3389693

CCS CONCEPTS

• **Security and privacy** → *Distributed systems security*; • **Information systems** → **Database transaction processing**;

KEYWORDS

Concurrency Control; Blockchain; Transaction; Database

ACM Reference Format:
Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3318464.3389693

1 INTRODUCTION

Blockchains have stricken the world like a storm. The concept of *blockchain*, originating from Nakamoto's Bitcoin whitepaper [20], proposes to employ a hashed chain of blocks to batch historical monetary transactions. This chain is distributed across a network of mutually distrusting nodes that run a *proof-of-work* (PoW) consensus to consistently replicate the chain and synchronize the state. The consensus groups the transactions into blocks, and the nodes serially execute the transactions in a block to update their state. While Bitcoin only supports cryptocurrency operations, Ethereum was designed to support *Turing-complete* smart contracts that encode *arbitrary* data processing logic [29]. With Ethereum, blockchains evolved from cryptocurrency platforms to distributed transactional systems.

Blockchain systems can be classified into *permissionless* (*public*), such as Bitcoin and Ethereum, and *permissioned* (*private*), such as Hyperledger Fabric [4]. In public blockchains, the data and transactional logic are transparent to the public, hence, are subject to private data leakage. Due to their openness, public blockchains use expensive PoW consensus. This, together with the serial transaction execution limit these systems' capacity. Addressing the limitations of public blockchains, Hyperledger Fabric is a private blockchain that

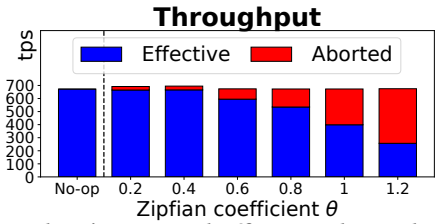


Figure 1: Fabric's raw and effective throughput under both no-op transactions and single modification transactions with varying skewness

supports *concurrent* transactions [4]. A Fabric blockchain requires its members to enroll through a trusted membership service in order to interact with the blockchain. In this paper, we focus on permissioned blockchains as they are more suitable for supporting applications such as supply-chain, healthcare and resource sharing, and in particular, we use Fabric as the underlying blockchain system.

Fabric supports a new transaction execution architecture called execute-order-validate (EOV). In this architecture, a transaction's lifecycle consists of three phases. In the first phase, *execution*, a client sends the transaction to a set of nodes, or peers, specified by an endorsement policy. The transaction is executed by these peers in parallel and its effects in terms of read and written states are recorded. Moreover, transactions from different clients may be parallelized during the execution. In the second phase, *ordering*, a consensus protocol is used to produce a totally ordered sequence of endorsed transactions grouped in blocks. This order is broadcast to all peers. In the third phase, *validation*, each peer validates the state changes from the endorsed transactions with respect to the endorsement policy and serializability.

The new EOV architecture limits the execution details of a transaction to the endorsing peers to enhance confidentiality and exploit concurrency. But such concurrency comes at the cost of aborting transactions that do not abide serializability. We evaluate the impact of concurrency control in Fabric on the setup described in Section 5. We measure both the raw and effective peak throughputs under both no-op transactions, with no data access, and update transactions, with varying skewness controlled by the zipfian coefficient. The raw throughput represents the in-ledger transaction rate, while the effective throughput represents committed transactions by excluding the aborted transactions from raw throughput. In Figure 1, a bar reports the raw throughput, while its blue part reports the effective throughput. The raw throughput is constant (677 tps) despite the workload type and request skewness. But with higher skewness, a larger proportion of transactions are aborted for serializability.

There are two notable directions attempting to solve this issue. The first is to improve upon Fabric's architecture to enhance its attainable throughput [21, 25]. For example, FastFabric proposes to split a node's functionality to alleviate the bottleneck and achieves the highest throughput among all improvements of Fabric [15]. However, these approaches are implementation-specific and might not generalize well to other blockchains. The second direction is to abstract out the transaction lifecycle to reduce abort rate. For example, Fabric++ [24] uses well-established concurrency techniques from databases to early abort transactions or reorder them to reconcile the potential conflicts.

Our work corresponds to the second direction, as a major attempt to *databasify* blockchains. Here, we take a principled approach to learn from transactional analysis techniques in databases with optimistic concurrency control (OCC) and apply them to enhance transaction processing in blockchains. We formally analyze the behavior of the current implementations of Fabric and Fabric++, and discover that both achieve *Strong Serializability* [5] (as described in Section 3.2). In fact, these implementations are more stringent than *One-Copy Serializability* (or simply Serializability), as prescribed by the original Fabric protocol [4]. Both systems employ a preventive approach which might over-abort transactions that are still serializable. In contrast, our proposal consists of a novel reordering technique that eliminates unnecessary abort due to in-ledger conflicts, with the serializability guarantee established on our theoretical insights. Our approach does not change Fabric's architecture, therefore it is orthogonal to the aforementioned optimizations, such as FastFabric [15]. In summary, our paper makes the following contributions:

- We theoretically analyze the resemblance of transaction processing in blockchains with EOV architecture and databases with optimistic concurrency control (Section 3.1). Based on this resemblance, we analyze the transactional behavior of state-of-the-art EOV blockchains, such as Fabric and Fabric++ (Section 3.2).
- We propose a novel theorem to identify transactions that can never be reordered for serializability (Section 3.3). Based on this theorem, we propose efficient algorithms to early filter out such transactions (Section 3.4), with the serializability guarantee for the remaining after reordering. We also discuss the security implications of our proposal (Section 3.5).
- We implement our proposed algorithms on top of two existing blockchains. First, we use Hyperledger Fabric v1.3 as the base and name our implementation FabricSharp (or Fabric# for short). Second, we start from FastFabric [15], which obtains the highest throughput among all optimizations of Fabric, and name our implementation FastFabricSharp (or FastFabric# for short). We have released FabricSharp for public use [1].
- We extensively evaluate FabricSharp by comparing it with the vanilla Fabric, Fabric++, and two other implementations based on database concurrency control

Table 1: The transaction summary in Figure 2. Staled reads and installed writes are marked in red and blue colors. The symbols ✓, x, N.A. respectively indicate committed, aborted, or not-allowed transactions.

		Txn1	Txn2	Txn3	Txn4	Txn5
Readset	Key	B C	A **B**	B	**C**	**C**
	Version	1,2 2,1	1,1 **1,2**	2,1	**2,1**	**2,1**
Writeset	Key	C	C	**C**	B	A
	Value	301	302	**303**	304	305
Commit status	Fabric	**N.A.**	x	✓	x	x
	Fabric++	x	x	x	✓	✓

techniques from one standard approach [9] and a recent proposal by Ding et al [11]. The experimental results show that the throughput of FabricSharp is more than 25% higher compared to the other systems. In addition, the FastFabricSharp's improvement over FastFabric is up to 66%.

The remaining of this paper is structured as follows. Section 2 provides background on EOV blockchains and OCC techniques. Our theoretical analysis follows in Section 3, ending with our reordering algorithm. Section 4 describes the implementation of our approach. Section 5 reports our experimental results. We review the related work in Section 6, before concluding in Section 7.

2 BACKGROUND

2.1 EOV architecture in Fabric and Fabric++

Hyperledger Fabric [4] is a state-of-the-art permissioned blockchain that features a modular design based on the EOV architecture. Fabric++ [24] is an optimization of Fabric, which reorders transactions after consensus to reduce the abort rate. A Fabric/Fabric++ blockchain is run by a set of authenticated nodes, whose identity is provided by a membership service. A node in this blockchain has one of the following three roles: (i) *client* which submits a transaction proposal for execution, (ii) *peer* which *executes* and *validates* transaction proposals, or (iii) *orderer* which *orders* transactions and batches them in blocks. Transaction order is determined collectively by all orderers in the blockchain based on a consensus protocol.

The state of a blockchain after forming a block is maintained by a versioned key-value store. Each entry in this store is a tuple (`key`, `ver`, `val`), where `key` is a unique name representing the entry, and `ver` and `val` are the entry's latest version and value, respectively. Moreover, `ver` is a pair consisting of the sequence number of the block and the transaction that updated the entry. For example, in Figure 2a, the entry (C, (2, 1), 201) in the state after block 2 indicates that the key C contains the latest value 201 which was lastly updated by the 1st transaction in block 2.

In Fabric/Fabric++, the workflow of a transaction consists of three phases: execution, ordering, and validation. We elaborate on these phases below, using the example in Figure 2a.

Execution. In this phase, clients propose transactions consisting of smart contract invocations to a set of endorsing peers, which are selected by an endorsement policy. Each endorsing peer executes transaction proposals concurrently and speculatively and returns the simulation results together with its endorsement signature. The results contain two value sets called the *readset* and the *writeset* which respectively represent the version dependencies (all keys read along with their version numbers) and the state updates (all keys modified along with their new values) produced by the simulation. For example, the readset and writeset of transactions in Figure 2a are summarized in Table 1. Throughout the execution, a transaction holds a read lock on the state database to guarantee that read values are the latest. Transactions that read across blocks, such as `Txn1` in Figure 2a, are not allowed in Fabric. In contrast, Fabric++ optimistically removes this lock for more parallelism but aborts transactions that read across blocks. After a client collects enough identical simulation results as required by the endorsement policy, it packages them into a single transaction and submits it to orderers.

Ordering. In this second phase, orderers receive transactions and sequence them into a total order to form a block, as shown in Figure 2b. Each orderer may belong to different administrative domains and receive different transaction proposals from various clients. But all orderers rely on a single consensus protocol to establish a common transaction order. Fabric/Fabric++ outsources this consensus service to Kafka. With the consistent transaction stream from the consensus, each orderer employs the same block formation protocol to batch transactions into blocks, and consequently delivers them to peers. A block is formed when the number of pending transactions reach the threshold or a timeout triggers. For example, in Figure 2a, Orderer1 receives `Txn5` and Orderer2 receives `Txn2`, `Txn3`, and `Txn4`. They send the transactions to the consensus service and receive the same transaction order. Based on this order, both orderers package the transactions into identical blocks, i.e., block 3 with `Txn2` to `Txn5`, given that the protocol limits the maximum number of transactions per block to 4.

Validation. This phase is executed by each peer after a block has been retrieved from orderers. Transactions in a block are sequentially validated based on the corresponding endorsement policy and transaction serializability. The serializability of a transaction is tested by inspecting the staleness of its readset. The transaction is marked as invalid if it reads a key whose version at the read time is inconsistent (or older) than the latest version. For example, in Figure 2a, transaction `Txn2` in block 2 is unserializable since it reads key B with version (1, 2) from block 1, which is inconsistent

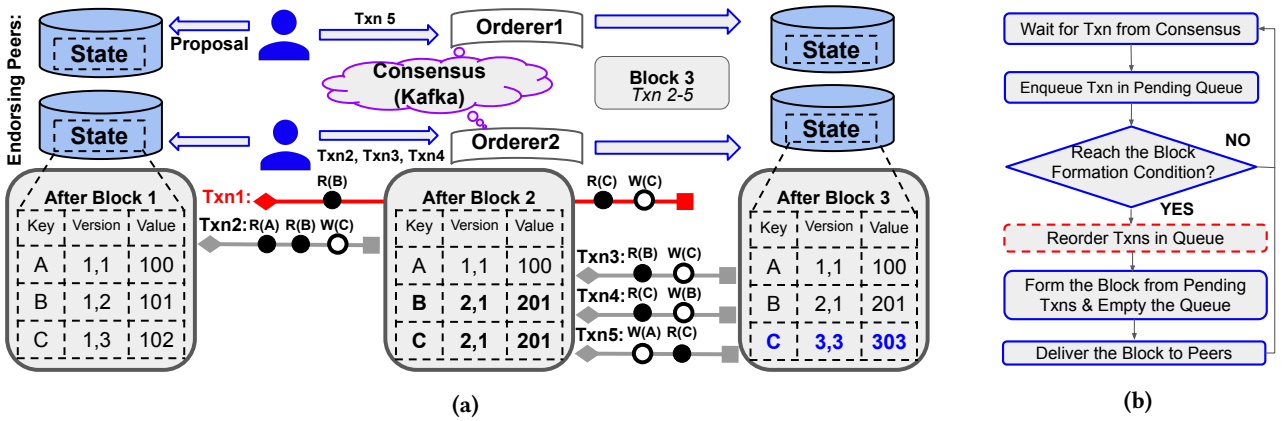


Figure 2: (a) Example of transaction workflow in Fabric. An arrow represents the lifespan of a transaction's execution (simulation), e.g, Txn1 starts its execution immediately after block 1 and finishs its simulation after block 2. (b) Procedures replicated in each Fabric Orderer. Fabric++ introduces a reordering step before the block formation to reduce the transaction abort rate.

with the latest version (2, 1) in block 2. Suppose that Txn3 passes the serializability test and updates the version of key C from (2, 1) to (3, 3) in block 3. Then, transactions Txn4 and Txn5 become invalid, since they both read an inconsistent version of key C in block 2. Hence, after this validation phase, only transaction Txn3 in block 3 is committed, while transactions Txn2, Txn4, Txn5 are aborted. To satisfy the serializability constraint, Fabric++ introduces a reordering step immediately before block formation but after consensus. The reordering is based on the commit order determined by the consensus and the accessed records in the transactions. For example, each orderer in Fabric++ puts Txn3 behind Txn4 and Txn5. Then, Txn4 and Txn5 are committed while Txn3 is aborted. Hence, Fabric++ commits one more transaction than Fabric.

2.2 Optimistic Concurrency Control in Databases

Unlike pessimistic concurrency control, the OCC technique does not hold locks to regulate transactional interference. Instead, each transaction has a unique *start timestamp* assigned to it from a global atomic clock. All queries reflect the state snapshot of the database at the start timestamp, without observing later changes. Each transaction is also assigned an *end timestamp*. Before committing, the database system checks the validity of a transaction based on these two timestamps and the accessed records. OCC can easily achieve *Snapshot Isolation*, which disallows concurrent transactions updating the same key [6]. Considering the fact that Snapshot Isolation suffers anomalies such as Lost Update and Write Skew, a number of attempts have been made to transform Snapshot Isolation to Serializable level [8, 14, 31].

3 THEORETICAL ANALYSIS

In this section, we first describe the resemblance of transaction processing techniques in EOV blockchains and OCC databases. Then, we use the transactional analysis method of OCC databases to reason about the serializability behavior of EOV blockchains, such as Fabric and Fabric++. Finally, we propose a reordering-based concurrency control algorithm for ordering serializable transactions in EOV blockchains, along with the discussion on its security implications.

3.1 Resemblance in Transaction Processing

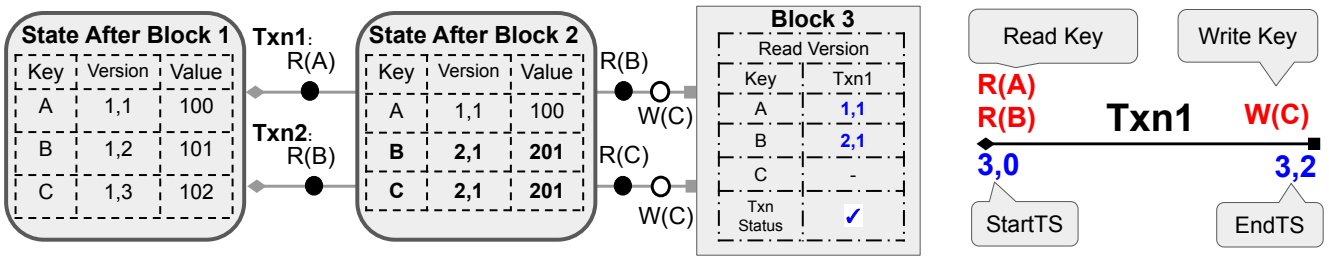
Similar to database systems where the concept *database snapshot* is used to describe a read-only, static view of a database [19], in blockchains, we can define the similar concept of *blockchain snapshot* as follows.

Definition 3.1 (Blockchain snapshot). A blockchain snapshot is the state of a blockchain after a block is committed. Let M be the sequence number of the committed block, then the corresponding snapshot is denoted as M and is said to have the sequence number $(M{+}1, 0)$ [1].

Definition 3.2 (Snapshot consistency). A transaction is snapshot consistent if there exists a blockchain snapshot M from which all the transaction's records are read.

Transactions in Fabric satisfy snapshot consistency since Fabric uses a lock to ensure the simulation is done against the latest state. Fabric++ optimistically removes the lock but early aborts transactions which read across blocks. Hence, it also satisfies the snapshot consistency. However, eliminating

[1]We use the two-value tuple with 0 fixed for the second element. This is to facilitate the ordering relations $<$ of sequence numbers of blockchain snapshots and transaction timestamps.



(a) Txn1, which reads across blocks, is snapshot consistent and can be scheduled with serializability. Txn2 is not as its early-read key B is updated before its execution ends.

(b) Concise notation to represent a transaction

Figure 3: Example of transactions reading across blocks

transactions based on cross-block reading might lead to over-aborting snapshot consistent transactions.

Example 3.3. In Figure 3a, Txn1 reads key A of version (1, 1) in snapshot 1 and key B of version (2, 1) in snapshot 2. These versions are the same as the versions of keys A and B in snapshot 2. Hence, Txn1 is *snapshot consistent* with block snapshot 2. In contrast, transaction Txn2, which also reads across blocks, does not achieve snapshot consistency because the value of previously read key B changes in block 2.

Proposition 3.4. *There exist snapshot-consistent transactions that read across blocks. For such a transaction, its block snapshot is determined by its last read operation.*

Proof. Txn1 in Figure 3a is a witness example. We have described in Example 3.3 that Txn1 reads across blocks 1 and 2, and it is still consistent with block snapshot 2. □

Proposition 3.4 shows that a legitimate transaction in an EOV blockchain can read across blocks, if their states are consistent. This makes the EOV blockchain similar to an OCC database, as the latter also reads from consistent states determined by the transaction's start timestamp. We also observe that the blockchain's sequence numbers have similar properties with databases' timestamps, such as atomicity, monotony, total order, and unique mapping to snapshots. Therefore, we define the timestamps of blockchain transactions using their sequence numbers.

Definition 3.5 (Start timestamp). The start timestamp of transaction Txn, denoted by StartTs(Txn), is the sequence number of its read snapshot.

Definition 3.6 (End timestamp). The end timestamp of transaction Txn, denoted by EndTs(Txn), is its sequence number in the block, determined by the consensus.

For example, in Figure 3a, Txn1 has StartTs(Txn1) = (3, 0) and EndTs(Txn1) = (3, 1), since it lastly reads from block 2 and occupies the first position in block 3. For brevity, in later paragraphs, we use the notation presented in Figure 3b to denote a transaction. Moreoever, the sequence numbers of transactions' start or end timestamps are lexicographically ordered, e.g., (2,1) < (2,2) = (2,2) < (3,0).

Definition 3.7 (Concurrent transactions). Two transactions Txn1 and Txn2 are said to be concurrent if their executions overlap. To be specific, if Txn1 ends earlier than Txn2 (i.e., EndTs(Txn1) < EndTs(Txn2)), then Txn2 must start before Txn1 ends (i.e., StartTs(Txn2) < EndTs(Txn1)). Otherwise, if Txn2 ends earlier than Txn1 (i.e., EndTs(Txn2) < EndTs(Txn1)), then Txn1 must start before Txn2 ends (i.e., StartTs(Txn1) < EndTs(Txn2)).

Proposition 3.8. *Each pair of transactions in the same block are concurrent.*

Proof. Suppose two transactions Txn1 and Txn2 are committed in the same block M at position p and q, respectively, where $p < q$. Since the latest block that Txn2 can read from is $M-1$, we have that: StartTs(Txn2) $\leq (M, 0) <$ EndTs(Txn1) $= (M, p) <$ EndTs(Txn2) $= (M, q)$. Hence, Txn1 and Txn2 are concurrent. □

Proposition 3.9. *The reverse of Proposition 3.8 is not true: there are concurrent transactions not belonging to the same block.*

Proof. We present a witness example in Figure 4, where transactions Txn1 and Txn2 respectively belong to block M and $M+1$. However, Txn2 reads from a block earlier than M. Hence, we have: StartTs(Txn2) $\leq (M, 0) <$ EndTs(Txn1) $= (M, 1) <$ EndTs(Txn2) $= (M+1, 1)$. Therefore, Txn1 and Txn2 are concurrent. □

From the above two propositions, concurrency does not only occur between transactions within the same block. Fabric++ fails to consider dependencies among transactions across blocks. Hence, its reordering effect is limited.

3.2 Serializability Analysis

Figure 5 shows all six scenarios of canonical transaction dependency (or conflict) between snapshot transactions, as

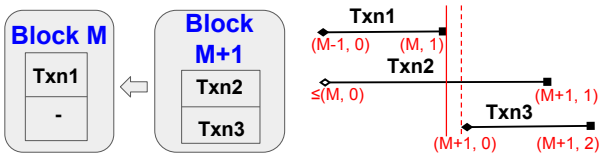


Figure 4: Txn2 and Txn3 are in the same block and concurrent. Txn1 and Txn2 are in different blocks, but they are still concurrent. Txn1 and Txn3 are not concurrent.

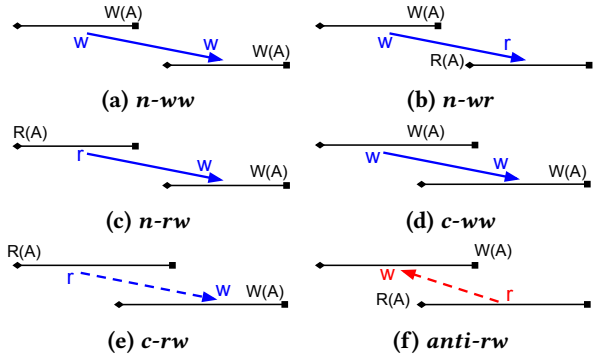


Figure 5: Six canonical dependencies between snapshot transactions. Here, (a), (b), and (c) are non-concurrent and (d), (e), and (f) are concurrent.

described by [14]. Among them, three dependencies, namely *n-ww*, *n-wr*, and *n-rw* are between non-concurrent transactions. The other three dependencies, namely *c-ww*, *c-rw*, and *anti-rw* are between concurrent transactions. According to the conflict serializability theorem in [28], the effect of a serializable transaction schedule is equivalent to any serialized transaction history that respects dependency order. Note that the dependency graph of the serializable transaction schedule must be acyclic.

Definition 3.10 (Strong Serializability). A schedule of transactions is *Strong Serializable* if its effect is equivalent to the serialized history, which conforms to the transactions' commit order determined by their end timestamps.

Theorem 3.11. *A schedule of transactions without anti-rw achieves Strong Serializability.*

Proof. We first prove that any transaction schedule without *anti-rw* achieves Serializability. By contradiction, suppose that such a transaction schedule does not achieve Serializability. Then, in the schedule there must be a subset of transactions with a dependency cycle, in which the last committed transaction is denoted by Txn. Then Txn must exhibit an *anti-rw* dependency because *anti-rw* is the only one among all six dependencies that relates later transactions to earlier ones. But this contradicts our assumption. Hence, the transaction schedule is serializable. Next, we prove that it also achieves Strong Serializability. Since the order of the five remaining dependencies is consistent to their commit order, the serialized history that respects the commit order also respects the dependency order. According to the conflict serializability theorem in [28], this serialized transaction history has the equivalent effect of the serializable schedule. Hence, the transaction schedule is Strong Serializable. □

We remark that Fabric/Fabric++ do not allow *anti-rw* between two transactions because the latter transaction would read an old version of the updated key, hence, it must be aborted. Based on Theorem 3.11, transactions in Fabric/Fabric++ satisfy Strong Serializability, which is more stringent than Serializability [4]. This opens up the opportunity to reduce the transaction abort rate.

3.3 Reorderability Analysis

Under Serializability instead of Strong Serializability, we formally analyze the reorderability of transactions in EOV blockchains. We focus on determining a serializable schedule by switching the commit order of pending transactions.

Lemma 3.12. *In blockchains, reordering can only happen between concurrent transactions.*

Proof. Assume transaction reordering occurs between two non-concurrent transactions. These transactions are committed in different blocks, due to the contra-positive of Proposition 3.8. Switching their order means changing a previously committed block, which is impossible in blockchains due to their immutability. □

Lemma 3.13. *A transaction does not change its concurrency relationship with respect to others after reordering.*

Proof. Assume the next block's sequence number is M. For any pending transaction Txn, we have: StartTs(Txn) $\leq (M, 0) <$ EndTs(Txn). Other transactions are classified into three cases. (i) For any non-concurrent transaction Txn1, we have: EndTs(Txn1) $<$ StartTs(Txn). Since reordering does not affect StartTs(Txn), the non-concurrency between Txn and Txn1 still holds. (ii) For any concurrent transaction Txn2 committed earlier than block M, we have: StartTs(Txn) $<$ EndTs(Txn2) $< (M, 0) <$ EndTs(Txn). Since reordering cannot move the commit time of Txn before $(M, 0)$, Txn2 and Txn remain concurrent. (iii) For any pending transaction Txn3, we have either StartTs(Txn) $< (M, 0) <$ EndTs(Txn3) $<$ EndTs(Txn), or StartTs(Txn3) $< (M, 0) <$ EndTs(Txn) $<$ EndTs(Txn3). Hence, Txn and Txn3 remain concurrent after reordering. □

The above Lemma 3.12 ensures that reordering does not impact non-concurrent transactions and their dependencies. Lemma 3.13 ensures that non-concurrent transactions are not

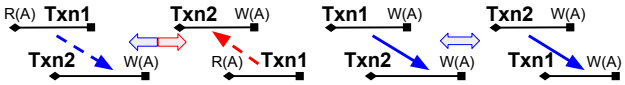


Figure 6: Dependency order preserves between *c-rw*, *anti-rw* but not *c-ww* when switching commit order

introduced by reordering. Therefore, we restrict our analysis to concurrent dependencies. We describe the dependency order of concurrent transactions using the two lemmas below.

Lemma 3.14. *If two transactions* Txn1 *and* Txn2 *exhibit c-rw or anti-rw dependency, switching their commit order does not affect their dependency order.*

Proof. When Txn1 and Txn2 exhibit *c-rw* (or *anti-rw*) dependency, if we switch their commit order, they will exhibit *anti-rw* (or *c-rw*) dependency, as illustrated in the left side of Figure 6. Consequently, in both two cases, their dependency order remains the same, i.e., Txn1 reads a key which will be written later by Txn2. □

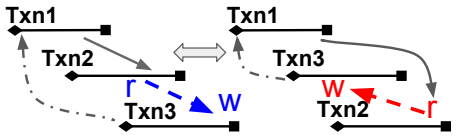
Lemma 3.15. *If two transactions* Txn1 *and* Txn2 *exhibit c-ww dependency, switching their commit order flips their dependency order.*

Proof. When Txn1 and Txn2 exhibit *c-ww* dependency, Txn1 writes to a key which will be over-written by Txn2. If their commit order is switched, then Txn2 and Txn1 will exhibit *c-ww* dependency, as illustrated in the right side of Figure 6. Now, Txn2 writes to a key which will be over-written by Txn1. Consequently, the dependency order of Txn1 and Txn2 is flipped. □

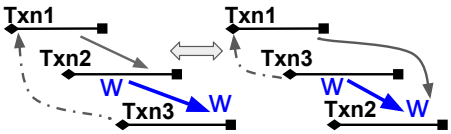
Finally, we present a theorem on reordering transactions containing a dependency cycle. This theorem is utilized in Section 3.4 to design our novel fine-grained concurrency control algorithm.

Theorem 3.16. *A transaction schedule cannot be reordered to be serializable if there exists a cycle with no c-ww dependencies involving pending transactions.*

Proof. We classify the dependencies in the cycle into two categories. The first category includes those involving at least one committed transaction in the dependecy. Due to the immutability of blockchains, reordering does not impact these dependencies, because the relative commit order of two transactions is fixed. The second category includes all dependencies between a pair of pending transactions. For each dependency, its corresponding transactions must be concurrent, otherwise, the preceding transaction would be committed. Due to the fact that the pending transactions are concurrent and the absence of *c-ww*, the order switching can only happen between conflicting transactions with *c-rw*



(a) An unreorderable schedule without *c-ww*



(b) A reorderable schedule with *c-ww*

Figure 7: Transaction schedule reorderability

or *anti-rw*. Their dependency order preserves despite being reordered (Lemma 3.14). Hence, the cyclic schedule remains unserializable, as shown in Figure 7a. □

However, a transaction schedule can be reordered to be serializable if there exists a cycle with one *c-ww* conflict between pending transactions. Due to Lemma 3.15, their dependency order can be flipped. We present this scenario in Figure 7b, where a cyclic schedule formed by Txn1, Txn2 and Txn3 becomes serializable by switching the commit order of Txn2 and Txn3, which exhibit *c-ww* dependency.

3.4 Fine-grained Concurrency Control

Theorem 3.16 states that a cyclic transaction schedule without *c-ww* among pending transactions can never be serializable despite reordering. Based on this insight, we formulate the following three steps for fine-grained concurrency control in EOV blockchains.

- For a new transaction, we first consider all dependencies, except *c-ww*, among all pending transactions (including the new transaction). Then, we directly drop the new transaction if there is a dependency cycle.
- On block formation, we retrieve the pending transaction order that respects all the computed dependencies.
- Finally, we restore *c-ww* dependencies on pending transactions based on the retrieved schedule.

Note that *c-ww* dependency restoration is still necessary, as future unserializable transactions may encounter a cycle with a *c-ww* dependency which involves committed transactions. But both their commit and dependency order are already fixed. Hence, the dependency graph remains acyclic after the restoration.

We outline our fine-grained concurrency control in Algorithms 1, 2, and 3, while the implementation details are presented in Section 4. We use the notation $A \cup= B$ to represent the self-assignment with union $A := A \cup B$. Here, we argue that the topological sort in Algorithm 3 always

Algorithm 1: Contract simulation

Input: Contract invocation context.
Output: *readset*, *writeset* are simulation results, b is the number of the block simulated on.

```
1 b := fetch the number of the last block;
2 readset, writeset := simulate the contract invocation
    on Block b snapshot;                          // Section 4.2
```

Algorithm 2: On the arrival of a transaction

Data: G is the transaction dependency graph with nodes U and edges V, and P is the pending transaction set.
Input: t is the transaction identifier, b is the number of the block simulated on, and *readkeys*, *writekeys* are accessed keys during simulation.
Output: *reorderable* property of t.

```
1 dep := Compute t's dependency except c-ww among P
    based on G, b, readkeys, writekeys;           // Section 4.3
2 reorderable := true if no cycle is detected in G with
    respect to dep, or false otherwise;           // Section 4.4
3 if reorderable then
4 |   P ∪= {t};
5 |   G.U ∪= {t};
6 |   G.V ∪= dep;
```

Algorithm 3: On the formation of a block

Data: G is the transaction dependency graph, and P is the pending transaction set.
Output: s is the commit order of pending transactions.

```
1 s := Topologically sort P based on reachability in G;
2 ww := Compute c-ww among P with s;
3 G.V ∪= ww;                                      // Section 4.5
4 P := ∅
```

has a solution since the transaction dependency graph G is guaranteed to be acyclic by Algorithm 2. Even the sub-graph containing only the pending transactions, P, is a directed acyclic graph and, hence, must have a topological order.

Compared to the reordering algorithm in Fabric++, ours is more fine-grained because the unserializable transactions are aborted before ordering and the remaining transactions are guaranteed to be serializable without being aborted. Our reordering is no longer limited to a block's scope. Another notable difference is that we determine the block snapshot at the start of the simulation, while Fabric and Fabric++ determine it based on the last read operation. We allow block commit during the contract simulation for more parallelism, but this may introduce stale snapshots when previously read records are updated by committed transactions during the simulation.

3.5 Security Analysis

Our reordering algorithm serves as a part of the ordering process and needs to be replicated on each honest orderer to form the ledger after the consensus service has established the transaction order. We assume the safety and liveness of the original consensus service under its security model, either crash-failure or byzantine-failure. We now discuss whether both properties preserve after our reordering.

Safety. In the original Fabric design, there are four safety properties: *agreement, hash chain integrity, no skipping*, and *no creation* [4]. These properties require honest orderers to sequentially deliver consistent, untampered blocks in a ledger. We claim that our approach preserves *hash chain integrity* and *no skipping* as we do not change the block formation procedure. Next, *no creation* holds because we do not introduce new transactions. Lastly, we achieve *agreement* because we fully replicate the reordering on each orderer. Moreover, we do not introduce non-determinism which may lead to execution bifurcation. As long as honest orderers perform the reordering individually from a consistent transaction stream, they shall produce identical ledgers.

Liveness. Fabric defines liveness in terms of the *validity* property, which mandates all broadcasted transactions to be included in the ledger. Our algorithm may compromise this liveness property as aborted transactions are excluded from the ledger. However, we propose the following approach to prevent abusive usage. To be specific, in the consensus protocol, the transaction order is tentatively proposed by a leader node. When this order is accepted by the other nodes, it becomes the input of our reordering approach. Hence, the order is controlled by the leader, which may hinge on the publicly available reordering algorithm to maliciously defer certain transactions. Suppose the malicious leader detects an undesirable transaction `TxnT` which reads and writes a record against the state snapshot of block N. The leader, using both a proxy peer and a proxy client, can immediately prepare another transaction `TxnT'` which reads and writes the same record against block N. Next, the leader places `TxnT'` ahead of `TxnT` during ordering. The other orderers, unaware of this manipulation, may accept this ordering. Assuming `TxnT'` passes the reorderability test in Algorithm 2, each honest orderer will abort `TxnT`. It is because these two transactions form an unreorderable cyclic schedule, namely `TxnT'` depends on `TxnT` with *c-rw* and `TxnT` on `TxnT'` with *anti-rw*. The crux of the mitigation is to hide the transaction's details, such as accessed records, before the transaction order is established. For example, we allow clients to send only the transaction hash to the orderers. Moreover, clients have

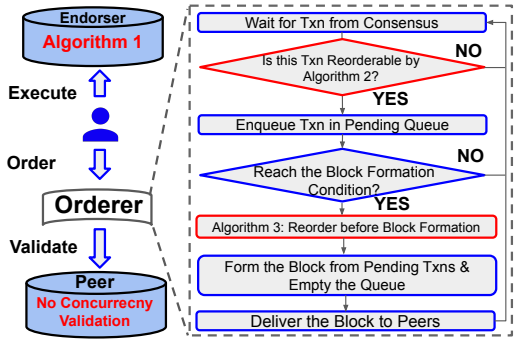


Figure 8: The integration of our approach

incentives to do so to avoid the above manipulation. After the sequence of a transaction hash is decided, its details are then disclosed to orderers for reordering. We remark that this approach also defers malicious clients from exploiting the reordering by mutating the transaction contents. It is because clients have already made a security commitment by publishing the transaction hash.

4 IMPLEMENTATION

4.1 Overview

We illustrate in Figure 8 the integration of our proposed fine-grained concurrency control in the ordering service of an EOV blockchain. In particular, we implemented our approach on top of both Hyperledger Fabric and FastFabric. For simplicity, we only show a single orderer and a single peer in the EOV pipeline, but the algorithms are replicated on each node. While the majority of our implementation is done in orderers, Algorithm 1 is integrated in the peers for snapshot-consistent transaction execution during the endorsement phase. In the ordering phase, we employ Algorithm 2 to test the reorderability of an incoming transaction after the consensus decides its commit order. Algorithm 3 performs the abort-free reordering immediately before pending transactions are batched into a block. We remark that Algorithm 2 and Algorithm 3 are far from implementation-friendly to system developers, as both employ an abstract dependency graph. In light of this, we present the details of designing the dependency graph and efficient operations on it.

4.2 Snapshot Read

We first describe the snapshot mechanism used by Algorithm 1. We rely on the storage snapshot mechanism to ensure each contract invocation is simulated against a consistent state. Specifically, after a block is committed, we create a storage snapshot and associate it with the block number. Each transaction, before its simulation, must acquire the number of the latest block, as shown in Algorithm 1. Staled snapshots without any simulation are periodically pruned. This design allows more parallelism across contract simulation in the Execution phase and block commit in the Validation phase. In contrast, vanilla Fabric uses a read-write lock to coordinate these two phases.

4.3 Dependency Resolution

To compute the dependency graph in Algorithm 2, we introduce two multi-versioned storages in the orderers to identify committed transactions. These storages are implemented in LevelDB and represent *CommittedWriteTxns* (*CW*) and *CommittedReadTxns* (*CR*), respectively. Each key of *CW* consists of the concatenation of the record key and the commit sequence of the transaction updating the value. For example, if `Txn1` with commit sequence (3, 2) writes to key `A`, *CW* has an entry {`A_3_2` : `Txn1`}. Similarly, each key of *CR* consists of the concatenation of the record key and the commit sequence of the transaction reading that key's latest value. For instance, the entry {`A_4_1` : `Txn7`} indicates that `Txn7` is the first transaction in block 4 which reads the latest value of key `A`. In both *CW* and *CR*, we place the record key prior to the commit sequence to efficiently support point query and range query. For example, the query $CW.Before(key, seq)$ returns the last committed transaction updating *key* with the commit sequence earlier than *seq*. Similarly, $CW.Last(key)$ returns the last committed transaction updating *key*. For the range query, $CW[key][seq:]$ returns all committed transactions from *seq* onward that update *key*.

We maintain two in-memory indices, *PendingWriteTxns* (*PW*) and *PendingReadTxns* (*PR*), to respectively store the keys for the write and read sets of pending transactions. Consider a new transaction *txn* that starts at *startTS* with read keys R and write keys W. All the dependencies of transaction *txn* are computed as follows.

$$\begin{aligned} \textit{anti-rw}(txn) &= \bigcup_{r \in R} CW[r][startTS:] \cup PW[r] \\ \textit{rw}(txn) &= \bigcup_{w \in W} CR[w] \cup PR[w] \\ \textit{n-wr}(txn) &= \bigcup_{r \in R} CW.Before(r, startTS) \\ \textit{ww}(txn) &= \bigcup_{w \in W} CW.Last(w) \end{aligned}$$

Note that we ignore *ww* dependencies between pending transactions and do not differentiate whether *ww* and *rw* are concurrent or not. This is because non-concurrent transaction may be part of a cycle. We then compute the predecessor transactions of *txn* as $\textit{ww}(txn) \cup \textit{n-wr}(txn) \cup \textit{rw}(txn)$, and successor transactions as $\textit{anti-rw}(txn)$.

4.4 Cycle Detection

We now discuss how we represent the dependency graph G to detect cycles and achieve serializability. We face two design choices. On the one hand, we could maintain only the immediate linkage information for each transaction and then perform graph traversal for cycle detection. On the

Algorithm 4: Reachability update for transaction *txn*

Data: G is the transaction dependency graph
Input: M is the number of next block to be committed, *pred* is *txn*'s immediate predecessor transactions, and *succ* is *txn*'s immediate successor transactions.

```
1 txn.anti_reachable := ∅;
2 for p in pred do
3 |   p.succ ∪= {txn};
4 |   txn.anti_reachable ∪= p.anti_reachable;
5 for s reachabale from succ in G do
6 |   s.anti_reachable ∪= txn.anti_reachable;
7 |   s.age := M;
```

other hand, we could maintain the entire reachability information among each pair of transactions. But the latter approach shifts the overhead from computation to space consumption. We achieve a sweet spot by maintaining the immediate successors of a transaction (*txn.succ*) and represent all transactions that can reach *txn* with a bloom filter, referred to as *txn.anti_reachable*. Cycle detection becomes straightforward by testing $p.anti\_reachable(s)$ for each pair (p, s) consisting of a predecessor and a successor of *txn*.

We use bloom filters because they are memory efficient and can perform fast union. Union is extensively used to update the reachability information for each transaction, as shown in Algorithm 4. Since a bloom filter internally relies on a bit vector, the set union can be fast computed via the bitwise OR operation. However, bloom filters are known to report false positives [7]. If the filters report such false positives for a pair of adjacent transactions to *txn*, we preventively abort *txn*. If they report negative for all pairs, then *txn* does not belong to any cycle in G.

Algorithm 4 entails the relatively expensive traversal of all reachable transactions from *txn*. However, this cost is bearable, since the traversal is unnecessary when *anti-rw*(*txn*) is empty. This is often the case under non-skewed workloads. Moreover, we reduce the cost of traversal by pruning the dependency graph, as described in Section 4.6.

Another issue in Algorithm 4 is the constant growth of the *anti_reachable* filter. In practice, we observe that the false positive rate of a single bloom filter grows to an intolerable ratio. To address this issue, we use two bloom filters with relay. Each transaction is associated with one bloom filter capturing transactions committed after block M and another bloom filter capturing transactions after block N. Suppose block C is the earliest block which contains a committed transaction in G. We maintain $M < C < N$ and use the first bloom filter for testing reachability. Whenever C grows to $M < N < C$, the first bloom filter is emptied and it starts

Algorithm 5: Restoration of *ww* within pending transactions based on the computed commit sequence

Data: G is transaction dependency graph.
Input: *seq* is committed sequence of pending transactions, *PW* is the index that associates updated keys with pending transactions.

```
1 head_txns := ∅;
2 for (key, txns) in PW do
3 |   Sort txns based on the relative order in seq;
4 |   (txn1, txn2) := the first pair in txns such that
  |     txn1 ∉ txn2.anti_reachable;
5 |   txn2.anti_reachable ∪= txn1.anti_reachable;
6 |   head_txns ∪= {txn2};
7 for txn in the topologically-ordered iteration of all
    txns reachable from head_txns do
8 |   for t in txn.succ do
9 |   |   t.anti_reachable ∪= txn.anti_reachable;
```

to collect transactions from the current block. We then use the second filter for testing reachability. In this manner, we restrict the number of transactions represented by a bloom filter within a certain block range so that the false positive rate remains acceptable. For safety, honest orderers must use the same M and N for exact replication.

4.5 Dependency Restoration

Next, we present our method to install *ww* dependencies into the dependency graph G based on the derived commit sequence, which is a topological order of the pending transactions P according to the reachability in G. One prominent issue is that the reachability of a transaction may be affected by multiple *ww* dependencies from various updated keys. But we want the reachability modification to take place within a single iteration for efficiency. Algorithm 5 outlines the major steps of the restoration of *ww* dependencies. We further explain this algorithm using the example in Figure 9.

For each *key* to be updated by pending transactions (*PW*), we topologically sort its associated transactions and select the first pair that is not yet connected in the reachability filter. In such a pair, the second transaction can be reached from all the predecessors of the first transaction. There can be a scenario where transactions in a pair are already connected in the reachability filter, which makes the restoration redundant. For example, this happens with `Txn0` and `Txn3` in Figure 9. For transactions that are not yet connected, we need to update their successors. To do this efficiently, we keep the transactions in a set (*head_txns*) and update their successors based on the topological order. Thereby, we avoid updating the information multiple times during the iteration in line 2. For example, `Txn8` in Figure 9 is reachable through

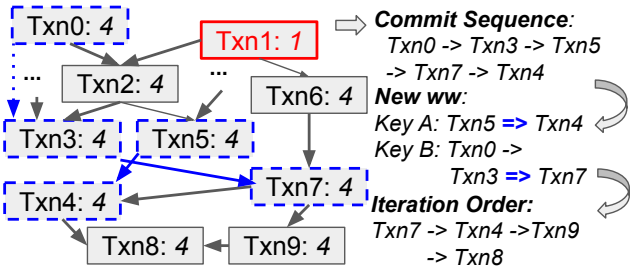


Figure 9: Example of a dependency graph with pending transactions (marked with blue dashed border), the commit sequence, new *ww* dependencies (marked with blue solid line) and the topologically-sorted iteration order. We do not consider the *ww* dependency between Txn0 and Txn3 (marked with blue dotted line), as it is implicit. Txn1 in red is subject to pruning due to staleness. The transaction age is in italic.

the update of both key A and B. Using our algorithm, the reachability information is updated once.

4.6 Dependency Graph Pruning

Since graph G can grow quickly, we prune transactions that either (i) are simulated against very old snapshots or (ii) cannot affect pending transactions. For the first case, we introduce a parameter called *max_span* to limit the block span[2] for a transaction. If the number of the next block is M, we compute the *snapshot threshold* as $H = M - max\_span$. Any transaction simulated against block H or earlier is aborted. For the second case, we define the *age* of a transaction *txn* to be the sequence number of the last committed block containing at least one transaction reachable from *txn* in G. When the snapshot threshold is greater than *txn*'s age, future transactions cannot be concurrent with any transaction that can reach *txn*. In this case, the *anti-rw* dependency will not happen, and this rules out any unserializable schedule containing *txn*. Therefore, *txn* can be safely pruned from G. We facilitate the pruning by arranging all transactions in G into a priority queue weighted by age. For new transaction to be committed in block M, we increase the age of the transactions reachable from it to M during the traversal in Algorithm 4 (line 7). For security, all orderers must use the same value for *max_span*.

5 EXPERIMENTS

5.1 Systems and Setup

First, we implement our approach on top of Hyperledger Fabric 1.3 and name the resulting system *FabricSharp*. We compare FabricSharp with the vanilla Fabric [4], Fabric++ [24], and two new systems which we developed by directly adopting OCC techniques from databases to Fabric. The first system, which we call *Focc-s*, follows the standard serializable OCC approach in [9]. This approach considers a dangerous pattern formed by two consecutive concurrent read-write conflicts with at least one *anti-rw*. We modify our Algorithm 2 such that incoming transactions with a *c-ww* conflict or a dangerous pattern are immediately aborted. Focc-s does nothing on block formation. The second system is *Focc-l*, which uses a recent OCC technique [11], based upon which we construct the read-write dependency graph and apply its *Sort-Based Greedy Algorithm* in Algorithm 3 for reordering. Focc-l does not filter any transactions in Algorithm 2.

Second, we implement our fine-grained concurrency control on top of FastFabric [20], an orthogonal Fabric improvement, and name the resulting system *FastFabricSharp*. We aim to investigate the performance of FastFabricSharp on a real production workload with low contention. In both sets of experiments, we deploy two *orderers*, three Kafka nodes and four *peers*, each on a physical machine with Intel Xeon E5-1650 3.5GHz CPU and 32GB RAM. The machines are connected via 1Gb Ethernet. We configure the smart contract to be endorsed (executed) by a single peer. Any of the four peers can serve as the endorser to spread the workload. The experiments are run at least three times and the average values are reported.

5.2 Workloads and Benchmark Driver

We use the same workloads that evaluate Fabric++ [24], which are based on the Smallbank benchmark. A transaction reads and writes 4 bank accounts, respectively, out of 10k accounts. We set 1% of them as hot accounts. Each read has a certain probability to access the hot accounts, controlled by the *Read hot ratio* parameter. Similarly, writing to hot accounts is controlled by the *Write hot ratio*. We introduce two more workload parameters, namely *Client Delay* and *Read Interval*. The former controls the delay of a client's broadcast to *orderer* after it receives the execution results from a *peer*. This parameter simulates the network transmission delay at the client side. The latter simulates computation-heavy transactions by controlling the interval between consecutive reads. Table 2 tabulates all the parameters, with the default value underlined. We fix *max_span* to 10 and the request rate to 700 tps. This is because Fabric can sustain a maximum raw throughput of around 700 tps on our setup, as shown in Figure 1. Unless otherwise specified, all reported throughputs denote the *effective throughput*, which represents the transactions that pass the serializability check and persist their states.

[2]If a transaction is simulated against block M and committed in block $M+1$, its block span is 1.

Table 2: Experiment parameters

Parameter	Value
# of transactions per block	50, 100, 200, 300, 400, 500
Write hot ratio (%)	0, 10, 20, 30, 40, 50
Read hot ratio (%)	0, 10, 20, 30, 40, 50
Client delay (x100 ms)	0, 1 ,2 ,3, 4, 5
Read interval (x10 ms)	0, 4, 8, 12, 16, 20

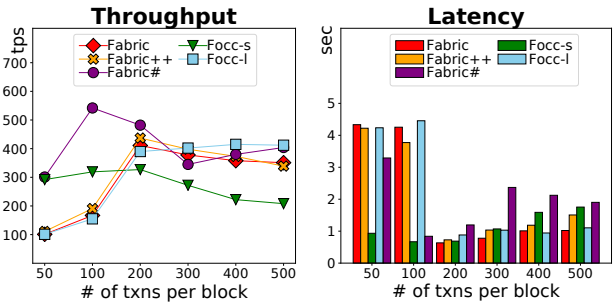


Figure 10: Performance under varying block size

5.3 The Performance of FabricSharp

Block Size. We first determine the block size that leads to the highest throughput for each system, and use these sizes for the remainder of the experiments. Figure 10 shows that the highest throughput (542 tps) is achieved by FabricSharp when the block size is set to 100 transactions. In contrast, Fabric, Fabric++, Focc-s and Focc-l reach their peak performance, 411, 437, 327, and 415 tps, respectively, when a block is limited to 200, 200, 200, and 400 transactions, respectively. Hence, our FabricSharp achieves 25% improvement in throughput compared to the state-of-the-art Fabric++.

Contrary to our expectation, Fabric++ does not achieve higher throughput with larger blocks, even though there are more transactions available for reordering. We attribute it to the longer latency that intensifies the contention, leading to more unserializable transactions. On the one hand, it takes longer to form a block. On the other hand, the reordering before block formation requires more time because there are more transactions. For example, we observe that reordering in Fabric++ takes 4.3ms with 50 transactions per block and 401ms with 500 transactions per block. In contrast, Focc-l takes 0.12ms and 5.19ms, respectively, due to its light-weight approach, even though it similarly constructs a dependency graph. This explains why Focc-l has shorter delay and performs better on larger blocks. But when the block size is smaller than 200, the throughput of both FabricSharp and Focc-s is significantly higher compared to Fabric, Fabric++, and Focc-l. It is because of the preventive abort applied by both FabricSharp and Focc-s during the Ordering

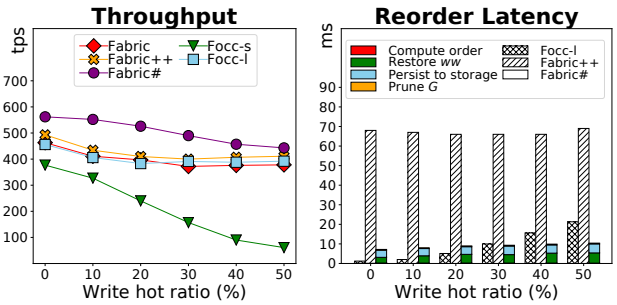


Figure 11: Throughput and reordering latency under varying write hot ratio

phase before reordering. This alleviates the congestion in the Validation phase, which is the bottleneck. In contrast, Fabric, Fabric++, and Focc-l exhibit high latency with smaller blocks because many unserializable transactions are included in the ledger and they overload the Validation phase.

Write Hot Ratio. To evaluate the effect of write-write conflicts, we concentrate more write operations into a fixed number of hot accounts. The throughput of FabricSharp remains the highest among all the systems, as shown in Figure 11 (left). As expected, the throughput of Focc-s drops significantly due to its prevention on *c-ww*. We also observe that the reordering latency of Fabric++ is constantly large, while this delay in Focc-l is smaller and proportional to the increasing skewness. This is because Focc-l iterates through the dependency graph of pending transactions in rounds. In each round, its *Sort-Based Greedy Algorithm* keeps pruning transactions until there are only transactions without dependencies. In contrast, Fabric++ computes all the cycles and determines the transactions to be aborted in batch mode. Hence, its reordering procedure is less sensitive to workload skewness compared to Focc-l. Figure 11 shows that the reordering latency in FabricSharp (Algorithm 3) is low. This is because FabricSharp shifts most of the work (e.g., the dependency graph maintenance) to Algorithm 2 on the transaction arrival. We notice that a large ratio (~50%) of the reordering delay in FabricSharp is due to the restoration of *ww* conflicts, and this ratio increases with higher write hot ratio.

Read Hot Ratio. We increase the read hot ratio to generate more read-write conflicts in the workload. As explained in Theorem 3.16, dependency cycles with these conflicts can never be reordered to become serializable. Consistent to our explanation, we show in Figure 12 that the throughput of all the systems, except Focc-s, decreases at a similar rate. The throughput of Focc-s is greater compared to Fabric and Focc-l when 50% of the read requests are on the hot accounts. This is because Focc-s imposes a more stringent condition for serializability compared to Fabric and Focc-l. Focc-s aborts

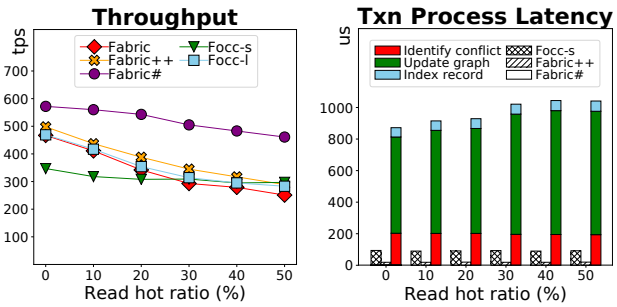


Figure 12: Throughput and transaction processing latency under varying read hot ratio

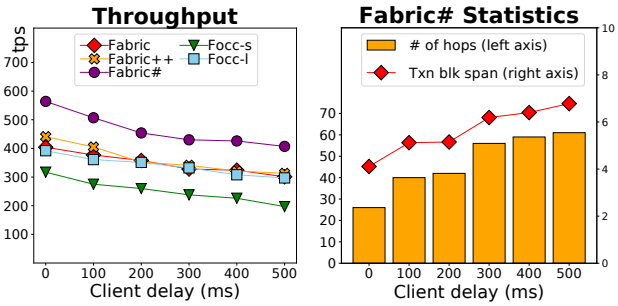


Figure 13: Throughput of all systems (left) and statistics of FabricSharp (right) under varying client delay

transactions if they are forming two consecutive read-write conflicts with at least one *anti-rw*, while the other systems, except FabricSharp, abort immediately when there is a single *anti-rw*. Hence, Focc-s can recover more serializable transactions especially under heavy read-write contention. Figure 12 (right) shows the processing latency breakdown for an incoming transaction. As expected, the reachability update on the dependency graph takes the largest proportion of the delay in FabricSharp, as all the reachable transactions from the incoming one must be traversed. This overhead increases with more dependencies in the workload. Compared to FabricSharp, the transaction processing delay in Focc-s and Fabric++ is almost negligible. However, Focc-s takes a bit longer than Fabric++, as it needs to additionally identify conflicted transactions, instead of only indexing the transactions based on the accessed records as in Fabric++.

Client Delay. Next, we simulate the network transmission latency at the client side, in order to study its impact on a transaction's end-to-end processing. Using the *client delay* parameter, we introduce a delay between the Execution and Ordering phases. As expected, a longer client delay increases the end-to-end latency and the block span of a transaction. In turn, this leads to lower throughput, as show in Figure 13 (left). Moreover, a larger block span leads to more concurrent

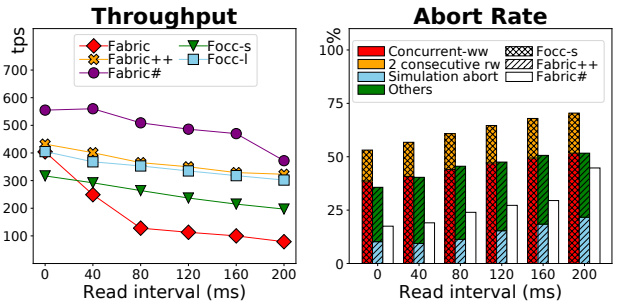


Figure 14: Throughput (left) and abort rate (right) under varying read interval

transactions and more dependencies. As shown in Figure 13 (right), FabricSharp traverses more transactions in the dependency graph to update their reachability (Algorithm 2) when the client delay is higher. Despite this, FabricSharp performs better than all the other systems.

Read Interval. To simulate the scenario where a transaction incurs heavy computations, we increase the interval between consecutive reads during the transaction execution. When a transaction takes longer to execute, there is a higher probability to read across blocks. Fabric++ prevents this scenario by aborting transactions that read across blocks even though some may be serializable. This is evidenced by a larger proportion of transactions that are early aborted during the Execution phase, as shown in Figure 14 (right). Focc-s consistently reads from a valid block snapshot and, hence, the effect of extending a transaction's execution results in a higher end-to-end latency. This leads to more concurrent transactions which, in turn, results in a higher abort rate for both *c-ww* and the dangerous pattern in Focc-s. Notably, the performance of the vanilla Fabric drops drastically with longer transaction execution. We attribute this to the read-write lock used during the simulation and block commit which prevents parallelism.

5.4 The Performance of FastFabricSharp

Next, we analyze the performance of our approach on top of FastFabric [15]. FastFabric separates peers in the same administrative domain into endorser, storage, and validator nodes. Endorsers, storage nodes, and validators are solely responsible for the transaction execution, block persistence, and transactions validation, respectively. These optimizations enable FastFabric to obtain a speedup of 6 compared to vanilla Fabric, as reported in [15].

We implement our reordering techniques on top of FastFabric and name the resulting system FastFabricSharp. In our experimental setup of four peers, we set two peers as endorsers, one as storage, and one as validator. We use two workloads based on the original Smallbank to evaluate the

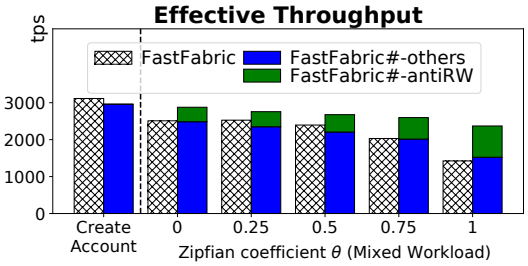


Figure 15: Effective throughput of FastFabric and FastFabricSharp under the contention-free (*Create Account*) and mixed workload

effectiveness of our approach when the workloads exhibit less conflicts compared to the modified Smallbank used in the previous set of experiments. The first workload consists of uniform update-only transactions (*Create Account*), which are all serializable due to the absence of read operations and hence *anti-rw*. The second workload is a mix of 50% read-only transactions (*Query Account*), 30% transactions that modify a single account (*Deposit Checking, Write Check, Transaction Saving*), and 20% transactions that modify two accounts (*Send Payment, Amalgamate*). The skewness of the accessed accounts is controlled by the zipfian parameter θ.

Figure 15 reports the comparison between FastFabric and FastFabricSharp. With the contention-free *Create Account* workload, FastFabric achieves a speedup of 4.5 on our experimental setup compared to the original Fabric, namely 3114 tps versus 677 tps in terms of raw throughput. The reordering overhead in FastFabricSharp is less than 5% and its effective throughput achieves 2960 tps. Under the mixed workload, FastFabricSharp achieves higher throughput compared to FastFabric. Moreover, the gap between FastFabricSharp and FastFabric grows with increasing skewness. When the $\theta = 1$, FastFabricSharp can achieve 2370 tps, 66% more than the 1424 tps of FastFabric. The gain of FastFabricSharp is mostly from the serialized transactions with *anti-rw*, as highlighted in Figure 15, which are all aborted by FastFabric.

We conclude that the benefits of our reordering outweigh the computation overhead, even under a heavy load. From previous experiments, this overhead is mostly due to the reachability update, which depends on the transaction complexity. In practice, if our reordering becomes the bottleneck, we could simply adopt a discriminated approach. For example, complex transactions with more referenced records can be checked with a simple serializability condition, e.g., the existence of *anti-rw* conflicts. On the other hand, simple transactions are passed to the fine-grained reordering.

6 RELATED WORK

Concurrency Control in Databases. Modern hardware with large memory and multi-core architecture opens up new opportunities for redesigning OLTP RDBMSs and their concurrency control [18]. Various works have attempted to either make data access more cache-friendly [26, 33] or extract more execution parallelism [30, 32]. Others strove to streamline the transaction ordering based on workload characteristics and application requirements [11, 16, 27]. Our work is closer to the latter approach, but we focus on blockchain systems. In particular, we learn from the transactional analysis in databases and propose a novel technique to efficiently reduce the abort rate in EOV blockchains.

Concurrency Control in Blockchains. The serial execution of smart contracts has long been the only solution adopted by blockchains due to ease of reasoning. Fabric, with its novel EOV architecture, is the pioneer system which formally introduced concurrency management into blockchains. It sparked a series of related optimizations, such as Fabric++ [24], and enhanced architectures, like OXII [3]. Fabric++ is the closest work to ours. It also aims to reduce the number of aborted transactions under EOV architecture. However, our approach provides a more fine-grained concurrency control which can still serialize transactions aborted by Fabric++.

Bridging the Gap between Database and Blockchain Transactions. Similarities between databases and blockchains have long been observed in their surveys [12, 22] and benchmarks [13]. There are several works addressing the atomicity of cross-chain or cross-shard transactions in blockchains, by transitioning the well-established database techniques, such as the classic Two Phase Commit [10, 17, 34]. Smart contract, as a key enabler for the transactional workload, is extensively optimized for better utility with data lineage [23] or confidentiality across applications [2].

7 CONCLUSIONS

We propose a novel solution to efficiently reduce the transaction abort rate in EOV blockchains by applying transactional analysis from OCC databases. We first draw a theoretical parallelism between EOV blockchains and OCC databases. Then, we propose a fine-grained concurrency control method and implement it in FabricSharp and FastFabricSharp based on Fabric and FastFabric, respectively. Our experimental analysis shows that both FabricSharp and FastFabricSharp outperform other blockchain systems, including the vanilla Fabric, Fabric++, and FastFabric. Unlike databases that achieve high throughput, the blockchains' limited throughput due to factors related to security opens up opportunities for precise transaction management.

ACKNOWLEDGMENTS

This research is supported by Singapore Ministry of Education Academic Research Fund Tier 3 under MOE's official grant number MOE2017-T3-1-007.

REFERENCES

[1] 2020. FabricSharp. https://github.com/ooibc88/FabricSharp

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1385–1398.

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 30.

[5] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment* 7, 3 (2013), 181–192.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[7] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[8] Mihaela A Bornea, Orion Hodson, Sameh Elnikety, and Alan Fekete. 2011. One-copy serializability with snapshot isolation under the hood. In *IEEE 27th International Conference on Data Engineering*. 625–636.

[9] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. 2008. Serializable isolation for snapshot databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. 729–738.

[10] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 123–140.

[11] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment* 12, 2 (2018), 169–182.

[12] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. 2018. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering* 30, 7 (2018), 1366–1385.

[13] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1085–1100.

[14] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 492–528.

[15] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2019. Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 455–463.

[16] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, and Aoying Zhou. 2019. Adaptive optimistic concurrency control for heterogeneous workloads. *Proceedings of the VLDB Endowment* 12, 5 (2019), 584–596.

[17] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. 2019. Cross-chain deals and adversarial commerce. *Proceedings of the VLDB Endowment* 13, 2 (2019), 100–113.

[18] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

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

[20] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://www.bitcoin.org/bitcoin.pdf

[21] Qassim Nasir, Ilham A Qasse, Manar Abu Talib, and Ali Bou Nassif. 2018. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks* 2018 (2018).

[22] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Dumitrel Loghin, Beng Chin Ooi, and Meihui Zhang. 2019. Blockchains and Distributed Databases: a Twin Study. *arXiv:1910.01310* (2019).

[23] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment* 12, 9 (2019), 975–988.

[24] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 105–122.

[25] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 264–276.

[26] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 18–32.

[27] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment* 10, 2 (2016), 49–60.

[28] Gerhard Weikum and Gottfried Vossen. 2001. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier.

[29] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.

[30] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. 2016. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1689–1704.

[31] Maysam Yabandeh and Daniel Gómez Ferro. 2012. A critique of snapshot isolation. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 155–168.

[32] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.

[33] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sanchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: harmonizing concurrency control and caching in a distributed OLTP database management system. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1289–1302.

[34] Victor Zakhary, Divyakant Agrawal, and Amr El Abbadi. 2019. Atomic commitment across blockchains. *arXiv:1905.02847* (2019).