Performance Optimization of High-Conflict Transactions within the Hyperledger Fabric Blockchain

Alexandros Stoltidis and Kostas Choumas and Thanasis Korakis
Dept. of ECE, University of Thessaly, Volos, Greece
Email: stalexandros, kohoumas, korakis@uth.gr

***Abstract*—Hyperledger Fabric (HLF) is a *blockchain (BC)* platform that supports secure high-throughput and low-latency transactions. However, it encounters challenges in managing conflicting transactions that negatively affect throughput and latency. This paper proposes a novel solution to address these challenges and improve performance. Our solution involves reallocating the *Multi-Version Concurrency Control (MVCC)* from the validation phase to a preceding stage in the transaction flow in order to enable early detection of conflicting transactions. Specifically, we propose and evaluate two innovative modifications, called *Orderer Early MVCC (OEMVCC)* and *OEMVCC with Execution Avoidance (OEMVCC-EA)*. Our experimental evaluation results demonstrate significant throughput and latency improvements, providing a practical solution for high-conflict applications that demand high performance.**

***Index Terms*—Blockchain, Hyperledger Fabric, Transaction flow, Conflicting Transactions, Transaction Optimization**

I. Introduction

BC is a *Distributed Ledger Technology (DLT)* [2], maintaining an immutable, distributed, and decentralized ledger of transactions across a network of nodes that reach a consensus on their states without a central authority [3]. DLT encompasses several distributed design paradigms that determine the design of the BC network. The two most prominent paradigms are permissionless or public, like Bitcoin and Ethereum, and permissioned or private, like HLF, Quorum, and Corda. These paradigms govern the access control of participants on the BC network [4].

Our research aims to optimize transaction throughput and minimize latency in applications where high-conflict transactions compete to modify the same assets within HLF. Transaction throughput is the rate at which transactions are committed to the ledger, encompassing the *world state*, the *log history*, and the *BC*. Transaction latency is the time it takes since a client submits a transaction proposal till the transaction gets committed [5]. We thoroughly examine and evaluate the different phases a transaction undergoes in the *Execute-Order-Validation (EOV)* [6] transaction flow of HLF, and based on our comprehensive analysis, we propose and assess various optimizations across its transaction flow.

The remainder of this paper is structured as follows: Section II provides an overview of the HLF architecture and its performance limitations under specific circumstances. Section III explores throughput and latency optimizations proposed in the existing literature. In Section IV, we present our solution, while Section V outlines our evaluation framework, experiments, and results. Finally, Section VI summarizes the content of this paper.

More details on this work are given in [1].

II. System Description - The HLF Architecture

In our research, we employ the latest HLF architecture, which includes the *peers*, the *gateway (GW)* [7], and the *ordering service*. Peers can function as either *endorsing* or *non-endorsing* peers. Endorsing peers use *smart contracts* or *chaincodes* in the execution phase to execute and endorse transaction proposals submitted by the clients. The GW is integrated within each peer, providing a layer of abstraction for the underlying network infrastructure to the clients. The ordering service consists of *orderers* that incorporate a consensus mechanism (Apache Kafka/Zookeeper or Raft [8]) in the ordering phase to order the received transactions into blocks and broadcast them to the peers with a gossip protocol.

Once a peer receives a block, the validation phase commences [9]. Firstly, it verifies the syntactic signature of the block and sends it through a pipeline of operations to validate each transaction individually. After syntactically validating each transaction within the block, it executes the *Validation System ChainCode (VSCC)* to ensure that endorsements adhere to the specified chaincode endorsement policy. Transactions fail VSCC due to inadequate endorsements, endorser signature issues, or mismatches in the read-write set versions between endorsing peers. Transactions failing VSCC are designated as invalid to prevent their commitment to the ledger.

Transactions that successfully pass VSCC go through MVCC, mitigating the double-spending problem. MVCC prevents read-write conflicts by comparing the versions of the keys in the read set during execution with those in the world state. It invalidates transactions upon detecting read conflicts or phantom reads. The former occurs as the read sets contain the versions of the keys during the execution phase, while write operations might increment these versions upon commitment. These discrepancies cause an MVCC read conflict at the validation phase. Phantom read conflicts occur during range queries by checking the entire range of keys for any insertions, deletions, or updates. If any alteration is detected, MVCC

invalidates the transaction. Finally, the peer commits the block to the ledger.

Problems arise when multiple clients with conflicting interests reference identical keys on their transactions. Peers only detect conflicts during the validation phase, impeding throughput and latency. Shifting part of the validation to earlier stages of the transaction flow would improve efficiency and system performance.

III. Related Work

Current research focuses on maximizing transaction throughput and minimizing latency, with limited work on early detection mechanisms for conflicting transactions in HLF. Existing literature mainly addresses conflict mitigation during execution or ordering phases. In [10], a lock-based mechanism detects conflicting transactions. However, its synchronized access to a trusted distributed locking service makes it unsuitable for delay-critical applications. Research in [11] and [12] moves part of the MVCC on peers after the execution phase. Each peer uses a local cache for the read-write sets of endorsed transactions, aborting transactions dependent on cached keys to prevent inevitable MVCC failures. As the validity of transactions depends on their position on the block, the peers should re-evaluate these transactions at the validation phase. In [13] and [14], the authors propose reducing MVCC failures using a conflict graph to reorder and abort conflicting transactions at the ordering phase. Similarly, [15] explores reducing conflicts by minimizing aborted transactions. It uses binary integer programming to group transactions into blocks for parallel processing, filtering obsolete ones, and prioritizing reads. While these methods improve performance, they add overhead and alter transaction sequences, yielding different results from the original HLF transaction flow.

IV. Proposed Solution

Our research proposes two distinct modifications of the EOV transaction flow, denoted as *EOV-Original (EOV-OG)*, named OEMVCC and OEMVCC-EA. These modifications rely on reallocating the MVCC process from the peers to the ordering service. This adjustment enhances concurrency between the ordering and validation phases, detecting invalid transactions as soon as possible while preserving the same order of transactions as the EOV-OG.

In this section, we will delve deeper into the intricacies of OEMVCC and OEMVCC-EA, discussing their differences and advantages in detail. As OEMVCC-EA is an extension of OEMVCC, we will begin by analyzing the logic behind the less intricate inner workings of OEMVCC before examining the more sophisticated OEMVCC-EA.

A. OEMVCC

Implementing OEMVCC requires modifications across the various components of the HLF network. The most significant modification is the reallocation of the MVCC process from the peers to the orderers. When the ordering service receives a transaction from a GW, it orders it into a block using a consensus mechanism. The orderers then asynchronously perform MVCC to classify and mark invalid transactions within a block. When a transaction is invalid, the ordering service promptly notifies the client that the transaction failed through the GW that sent the transaction. When the block reaches its maximum block size or block interval elapses, the ordering service waits until all transactions within the block have undergone MVCC before broadcasting it to the peers.

OEMVCC takes advantage of the latest versions of HLF, which can incorporate VSCC on the GW, invalidating transactions before dispatching them to the ordering service and notifying the client earlier in the transaction flow. Performing VSCC directly after the execution phase on the GW guarantees that valid MVCC transactions on orderers will successfully pass VSCC on the validation phase and eventually change the version of their keys upon block commitment.

OEMVCC implements a cache-based mechanism to incorporate MVCC and avoid direct access to the world state on the ordering service. The MVCC procedure checks the version numbers of the read-sets for each transaction against the cache. A transaction is invalid if one or more versions are outdated compared to the ones recorded in the cache. If all transaction keys succeed MVCC, the transaction is valid, and the orderers can update the cache with the expected versions of the keys upon commitment.

The caching mechanism within the ordering service should account for the logic of the underlying consensus algorithm. Currently, the default consensus algorithm is Raft, consisting of a leader and its followers. It regularly elects a leader orderer who orchestrates the ordering of transactions into blocks for its leadership duration. Followers forward incoming transactions to their leader, ensuring strong consistency between their log entries and those of the leader. In OEMVCC, the leader also performs MVCC, implying that the caching mechanism should incorporate strong consistency in its entries to accommodate elections, as a follower can transition into a leader at any time and undertake the ordering and MVCC. Thus, we implement a distributed cache coordinated by the consensus protocol to maintain this strong consistency and incorporate MVCC in the ordering service.

The behavior of peers during the validation phase also changes in OEMVCC. Contrary to the EOV-OG, where transactions sequentially undergo VSCC and MVCC, peers in OEMVCC bypass the VSCC and MVCC procedures, as the ordering service has already designated all transactions within the block as valid or invalid. Finally, we extend the GW to support receiving and handling notifications by the ordering service regarding invalid transactions.

B. OEMVCC-EA

OEMVCC-EA extends upon OEMVCC, aiming to increase transaction throughput while conserving network and computational resources by dropping invalid transactions earlier in the transaction flow. Similarly to OEMVCC, in OEMVCC-EA, the GW performs VSCC, and the ordering service is MVCC. OEMVCC-EA reduces block sizes in the ordering phase by

discarding invalid transactions after MVCC, ensuring blocks contain solely valid transactions. Consequently, the ordering service proactively informs the peers about the keys in the write sets of valid transactions to optimize their execution and validation phases. More specifically, endorsing peers utilize the write sets of valid transactions to avoid simulating transactions containing these keys since the resulting read-write sets will eventually become outdated, and the transaction will ultimately fail. Finally, as the peers receive blocks containing exclusively valid transactions, they can directly commit the included transactions. Once a transaction gets committed to the ledger, the peer can remove the keys included on the write sets of the transaction, enabling it to endorse upcoming transactions that involve these keys.

V. Evaluation Framework

To evaluate OEMVCC and OEMVCC-EA, we deployed an HLF v2.4.4 network on the NITOS [16] testbed based on the deployment options in [17]. The network includes ten clients, three orderers, and four peer nodes. The orderers are on a dedicated NITOS node and use the Raft consensus with a block size of 10 transactions and a block interval of 2 seconds, while clients and peers are on separate NITOS nodes. A distributed Redis cache supports the caching mechanism for the MVCC in the ordering service.

We enhanced the *asset transfer* [17] application to assess performance under high transaction conflict rates. Clients concurrently submit conflicting transactions to modify similar assets, with conflict rates of 20%, 50%, and 80%. Each transaction proposal is sent to two out of four peers for execution, requiring at least one endorsement before proceeding to the ordering phase.

Figure 1 shows the average duration of the execution phase relative to the percentage of conflicting transactions. Similarly, Figures 2 and 3 depict the average transaction latency and throughput, respectively. Latency is measured from when the client submits a transaction until it receives a notification about its status. Throughput is calculated based on the number of status notifications sent to clients within a specified time frame. To compare performance, we mark OEMVCC, OEMVCC-EA, and EOV-OG with the blue horizontally-lined, the red vertically-lined, and the black diagonally-lined bars, respectively. We also highlight the latency and throughput of the overall, valid, and invalid transactions by the circle-pointed, square-pointed, and triangle-pointed symbols, respectively. These metrics help us evaluate our modifications, offering insights into peer load and additional overhead. We first compare OEMVCC and OEMVCC-EA with EOV-OG and then OEMVCC with OEMVCC-EA.

A. OEMVCC and OEMVCC-EA vs EOV-OG

Figure 1 indicates that OEMVCC and OEMVCC-EA outperform EOV-OG, reducing the execution duration to at least by half, regardless of the conflict rate. As the execution duration closely mirrors the load on the peers, this reduction highlights the importance of early invalidation of transactions within the ordering service. By offloading MVCC to the ordering service and invalidating earlier in the transaction flow, the peers can conserve resources, enabling more efficient transaction simulation during the execution phase.

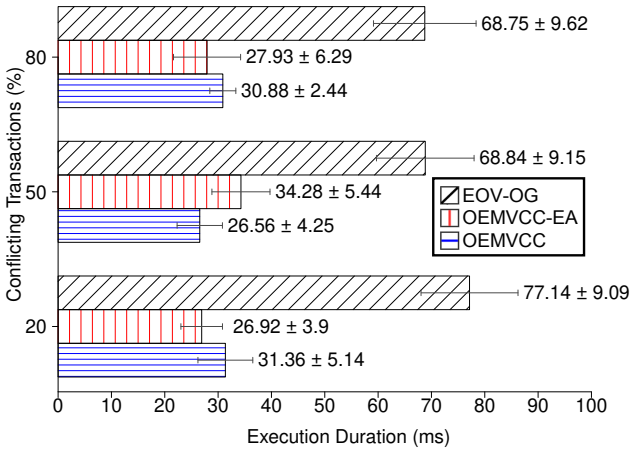

Fig. 1. Execution duration relative to the rate of conflicting transactions.

Figure 2 illustrates that OEMVCC and OEMVCC-EA have significantly lower transaction latency than EOV-OG, regardless of the conflict rate. This improvement is due to reduced latency for both invalid and valid transactions. Invalid transactions are identified earlier, while valid transactions benefit from asynchronous MVCC and ordering, avoiding the sequential ordering and validation as in EOV-OG. The transfer of MVCC to orderers and earlier invalidation, particularly in OEMVCC-EA, also helps peers conserve resources for processing valid transactions, further reducing latency.

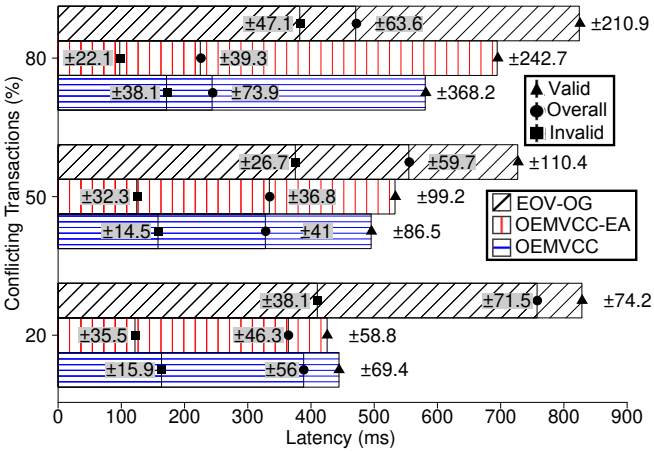

Fig. 2. Transaction latency relative to the rate of conflicting transactions.

Figure 3 demonstrates that OEMVCC and OEMVCC-EA consistently outperform EOV-OG in transaction throughput regardless of the conflict rate. This superiority in throughput is due to the lower latency of transactions in OEMVCC and OEMVCC-EA. However, both OEMVCC and OEMVCC-EA exhibit a higher standard deviation than EOV-OG, as their performance heavily depends on the probability of early invalidation of transactions.

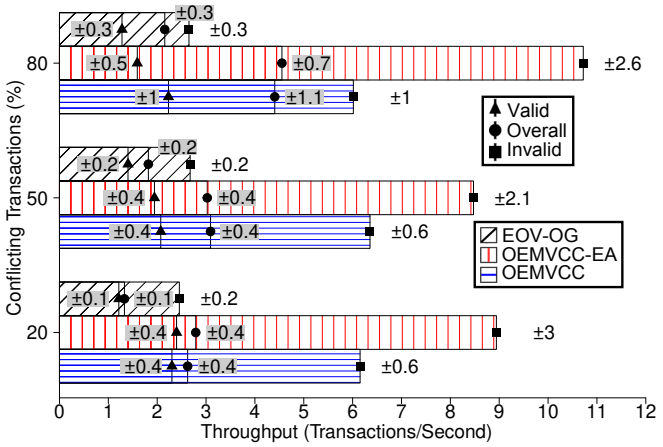

Fig. 3. Transaction throughput relative to the rate of conflicting transactions.

B. OEMVCC vs OEMVCC-EA

OEMVCC-EA outperforms OEMVCC in execution duration, overall latency, and overall throughput at 20% and 80% conflict rates, which is not the case for 50%, as illustrated in Figures 1, 2, and 3, respectively. The advantage of OEMVCC-EA is the earlier invalidation of transactions in the execution phase, conserving resources for potentially valid transactions. However, at a 50% conflict rate, the overhead of its cache-based nature becomes apparent. In OEMVCC-EA, peers update the status of keys within their cache during transaction commitment, avoiding unnecessary simulations but potentially impeding performance. At 20% conflict rates, the performance gains from resource conservation outweigh the overhead. At 80% conflict rates, most transactions are proactively invalidated, increasing performance with earlier notification and conserving enough resources to improve performance despite the concurrent activity. However, at 50% conflict rates, the overhead offsets the benefits of resource conservation, leading to a decline in performance. This tradeoff between the overhead and early invalidation with resource conservation explains the underperformance of OEMVCC-EA in execution duration, overall latency, and overall throughput at 50% conflict rates compared to OEMVCC.

Figures 2 and 3 indicate that OEMVCC-EA outperforms OEMVCC in latency and throughput of invalid transactions regardless of the conflict rate, as it can invalidate transactions earlier in the transaction flow. However, the performance of OEMVCC-EA for valid transactions decreases as the ratio of conflict rate increases, where its overhead also increases. Thus, OEMVCC-EA underperforms in latency and throughput of valid transactions compared to OEMVCC under 50% and 80%, while it excels at a 20% conflict rate where the overhead is minimal. Finally, OEMVCC-EA exhibits a higher standard deviation in the throughput of invalid transactions, primarily its reliance on the timeliness of information within the caches of peers that lack strong consistency.

VI. Conclusion

Our research shows that OEMVCC and OEMVCC-EA outperform EOV-OG in transaction latency and throughput, regardless of conflict rate. Similarly, OEMVCC-EA outperforms OEMVCC with invalid transactions. Conversely, OEMVCC performs better with valid transactions as the conflict rate increases, where the overhead of OEMVCC-EA becomes significant.

Acknowledgments

This project has received funding from the European Union's HE research and innovation programme under grant agreement No 101084188. Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union or the European Research Executive Agency (REA). Neither the European Union nor the granting authority can be held responsible for any use that may be made of the information the document contains.

References

[1] Arxiv. https://arxiv.org/abs/2407.19732.
[2] Ali Sunyaev. *Distributed Ledger Technology*, pages 265–299. Springer International Publishing, February 2020.
[3] Christian Cachin and Marko Vukolić. Blockchain Consensus Protocols in the Wild, 2017.
[4] Yannis Bakos and Hanna Halaburda. Permissioned vs Permissionless Blockchain Platforms: Tradeoffs in Trust and Performance. *NYU Stern School of Business Working Paper*, February 2021.
[5] Thakkar et al. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In *Proc. of IEEE MASCOTS*, 2018.
[6] Androulaki et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proc. of EuroSys*, 2018.
[7] Foschini et al. Hyperledger Fabric Blockchain: Chaincode Performance Analysis. In *Proc. of ICC*, 2020.
[8] Ongaro et al. In Search of an Understandable Consensus Algorithm. In *Proc. of USENIX ATC*, 2014.
[9] Chacko et al. Why Do My Blockchain Transactions Fail? A Study of Hyperledger Fabric. In *Proc. of ACM SIGMOD conf. on Management of Data*, 2021.
[10] Xu et al. Locking mechanism for concurrency conflicts on Hyperledger Fabric. In *Proc. of WISE conf.*, 2020.
[11] Aditya et al. Pathak. Early-stage Conflict Detection in HLF-based Delay-critical IoT Networks. In *Proc. of IEEE CNS*, 2023.
[12] Helmi Trabelsi and Kaiwen Zhang. Early Detection for Multiversion Concurrency Control Conflicts in Hyperledger Fabric, 2023.
[13] Ankur Kumar Sharma et al. Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In *Proc. of ACM SIGMOD conf. on Management of Data*, 2019.
[14] Pingcheng Ruan et al. A Transactional Perspective on Execute-order-validate Blockchains, 2020.
[15] Xu et al. Mitigating Conflicting Transactions in Hyperledger Fabric-Permissioned Blockchain for Delay-Sensitive IoT Applications. *IEEE Internet of Things Journal*, PP:1–1, 01 2021.
[16] NITOS. http://nitos.inf.uth.gr/.
[17] Hyperledger Fabric Samples. https://github.com/hyperledger/fabric-samples.