

# Optimizing Validation Phase of Hyperledger Fabric

Haris Javaid\* Chengchen Hu\* Gordon Brebner†

\*Xilinx, Singapore †Xilinx, USA

{harisj, chengchen, gjb}@xilinx.com

**Abstract**—Blockchain technologies are on the rise, and Hyperledger Fabric is one of the most popular permissioned blockchain platforms. In this paper, we re-architect the validation phase of Fabric based on our analysis from fine-grained breakdown of the validation phase’s latency. Our optimized validation phase uses a chaincode cache during validation of transactions, initiates state database reads in parallel with validation of transactions, and writes to the ledger and databases in parallel. Our experiments reveal performance improvements of 2× for CouchDB and 1.3× for LevelDB. Notably, our optimizations can be adopted in a future release of Hyperledger Fabric.

## I. INTRODUCTION

Blockchain technology is increasingly becoming popular, with applications in various domains such as finance, real estate, supply chains, etc. A blockchain is essentially a distributed ledger of transactions, which is maintained by all the participating nodes of the blockchain network. The transactions represent some business logic and are grouped into blocks which are appended to the ledger. Each node in the network updates its own copy of the ledger with the new block, after consensus is reached amongst the nodes.

In public or permissionless blockchains such as Bitcoin [3] and Ethereum [5], anyone can join the network and the consensus mechanism is based on proof-of-work algorithms which are computationally intensive. In permissioned blockchains, the identity of the nodes is known and authenticated cryptographically. The consensus mechanism is delegated to a few selected nodes in order to reduce bottlenecks in the consensus. Examples include Hyperledger Fabric [7], Quorum [9] and Corda [4]. Hyperledger Fabric is one of the most popular platforms as it is open-source and has already been shown to implement many enterprise applications such as food supply chain, healthcare, etc. [13].

In this paper, we focus on performance improvements for Hyperledger Fabric. The transaction flow in Fabric follows the *execute-order-validate* model, where a transaction is executed first, then ordered into a block, which is finally validated and committed to the ledger. Consequently, some nodes in the Fabric network act as a peer to execute/endorse transactions and validate/commit blocks, while other nodes act as orderers to create new blocks. In addition to the ledger, each peer node uses a state database to keep the global state of the blocks committed so far. Two options are available: (1) LevelDB [8] which is an embedded database and allows relatively fast accesses, and (2) CouchDB [1] which provides a client-server model and is accessible through REST API over HTTP. Unlike LevelDB, CouchDB allows rich queries over the global state but the accesses are relatively slow.

Many previous performance studies have highlighted validation phase as one of the major bottlenecks [11], [15], [23],

in addition to the bottlenecks in consensus mechanism [10]. For the validation phase, recent optimizations [15], [23] include validation of transactions in parallel, caching block and identity certificates, and bulk reading from slow CouchDB.

In this paper, we critically examine and thoroughly evaluate the validation phase of Hyperledger Fabric for further improvements. Based on our evaluation and observations, we propose several optimizations around executing various operations in parallel to overlap and hide their latencies. In particular, this paper makes the following contributions:

- Validation phase latency is broken down into six components for fine-grained analysis, and extensive experiments are run to highlight the bottlenecks and areas for improvement.
- Based on our observations, we find that the validation of transactions is much slower with CouchDB when compared to LevelDB. We propose a chaincode cache to speedup lookups for chaincode information (such as endorsement policy, etc.) instead of always accessing it from the state database during validation of transactions.
- We re-architect the validation phase to execute validation of transactions and state database reads in parallel in a way that their subtle dependency is avoided. Furthermore, we execute the ledger writes and state database writes in parallel. Based upon the type of state database used, the writes to history database are also combined with either the ledger write or the state database write operation.

We implemented our optimizations in Hyperledger Fabric v1.1, however they are also valid for v1.4. From our experiments with CouchDB, we show that the commit throughput at the peer nodes improved by 2×. For LevelDB, our optimizations improved performance by 1.3×. Most importantly, the proposed optimizations can be adopted in a future release of Hyperledger Fabric.

The rest of the paper is organized as follows. Section II provides an overview of Fabric with details of its validation phase. Section III presents our evaluation methodology and in-depth analysis of the validation phase, along with our optimized validation phase. Experimental results of our optimizations are discussed in Section IV. Section V describes the related work, and the paper is concluded in Section VI.

## II. HYPERLEDGER FABRIC ARCHITECTURE

### A. Overview

Hyperledger Fabric is an open-source, enterprise-grade implementation platform for permissioned blockchains. A Fabric network consists of different types of nodes, such as peers, orderers, clients, etc., where each node has an identity provided by the *membership service provider*.

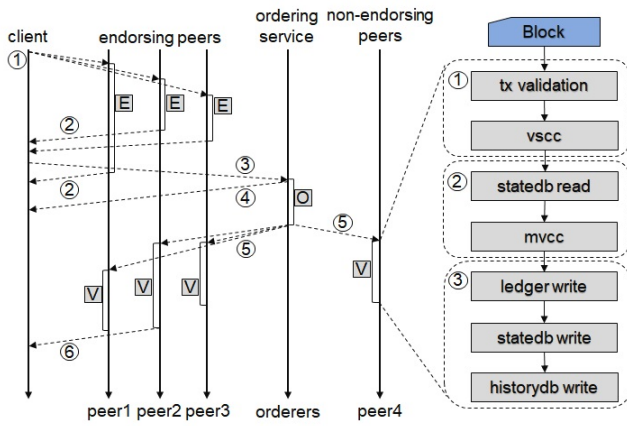


Fig. 1: Transaction Flow in Hyperledger Fabric.

An endorsing peer both executes/endorsees transactions and validates/commits blocks to the ledger. A non-endorsing peer only validates/commits blocks to the ledger. Execution of transactions is enabled by smart contracts or chaincodes, which represent the business logic and are instantiated on the endorsing peers.

The ordering service consists of orderers which use a consensus mechanism to establish a total order for the transactions. A block is created from the ordered transactions, and then broadcast to the peers. Multiple pluggable consensus mechanisms are available, such as Apache Kafka/Zookeeper [2] or Raft [20] based consensus mechanism.

### B. Transaction Flow

A transaction flows through the various nodes of a Fabric network as illustrated in Figure 1 (left-hand side, see [10] for more details). A client creates a transaction and sends it to a number of endorsing peers (step 1). Each endorsing peer executes the transaction against its own state database, to compute the read-write set of the transaction (marked as *E*). The read set is the keys accessed and their version numbers, while the write set is the keys to be updated with their new values. If there are no errors during the execution of the transaction, the peer sends back an endorsement to the client (step 2). After the client has gathered enough endorsements, it submits the transaction with its endorsements to the ordering service (step 3).

The ordering service responds back to the client after the transaction has been accepted for inclusion into a block (step 4). A block is created from the ordered transactions when either a user-configured timeout has expired or user-configured limit on block size has reached. Once a block is created (marked as *O*), the orderer broadcasts it to all the peers (step 5). Each peer validates all the transactions of the block and then commits it to the ledger and state database (marked as *V*). Finally, one of the peers sends a notification to the client that the transaction has been committed (step 6).

Figure 1 shows the operations of the validation phase in more detail on the right-hand side. On receiving the block from the orderer (or another peer) through the Gossip protocol, a peer checks the syntactic structure of the block, and then sends it through a pipeline of various operations. In step 1,

each transaction in the block is syntactically validated. Then, *vscc* (validation system chaincode) is run on each transaction where the endorsements are validated and the endorsement policy of the associated chaincode is evaluated. A transaction is marked as invalid if its endorsement policy is not satisfied.

In step 2, *mvcc* (multi-version concurrency control) check is applied. This check ensures that there are no read-write conflicts between the valid transactions; in other words, it avoids the double-spending problem. The read set of each transaction is computed again by accessing the state database, and is compared to the read set from the endorsement phase. If these read sets are different, then some other transaction (either in this block or an earlier block) has already modified the same keys, and hence this transaction is marked as invalid. For LevelDB, the state database read operation is integrated with the *mvcc* operation. However, for CouchDB, all the keys for all the transactions are read in a bulk operation before starting the *mvcc* operation.

In the final step 3, the block is committed. First, the entire block is written to the ledger with its transactions' valid/invalid flags. Then, the write sets of the valid transactions are committed to the state database. Finally, the history database is updated to keep track of which keys have been modified by which blocks and transactions.

## III. EVALUATION METHODOLOGY AND OPTIMIZATIONS

### A. Fabric Network Setup

We created a Fabric network with two organizations, where each organization had two endorsing peers and a certificate authority. We used Kafka based ordering service with two orderer nodes, four Kafka brokers and three Zookeeper nodes. Each peer is run on a virtual machine which is allocated 16 Intel Xeon 4416 @ 2.1GHz vCPUs with 32GB RAM, 50GB hard disk, and configured with Ubuntu 16.04 LTS. All the machines were connected through a 1Gbps network.

### B. Application

We used the *smallbank* benchmark from Hyperledger Caliper [6] to test our Fabric network. The *smallbank* benchmark is representative of a banking application, where its chaincode implements functions such as creation of a user account, transfer money, deposit cash, etc. For each experiment, the clients were configured to create random transactions from the pool of available functions. A total of 30,000 transactions were created and sent to the peers at the rate closer to their saturation point [23] (i.e., peer throughput is stable, which we determined empirically through our experiments). A single channel was created on the peers with the endorsement policy of at least one signature from each organization. We used a virtual machine with 8 vCPUs to run the *smallbank* clients.

### C. Metrics

Since our goal is to critically examine the validation phase of Fabric, we use the *commit throughput* and *block validation latency* as the primary metrics for performance evaluation. Commit throughput is defined as the rate at which transactions are committed to the ledger by the peer. The block validation

latency is the total time taken by the peer to validate and commit the entire block. Unlike previous works which consider only the coarse-grained latencies at the validation, *mvcc*, and commit operations [23], we breakdown the validation latency into six components for fine-grained analysis:

- *vscc*: time spent in syntactic validation of the transactions as well as the execution of *vscc* for validation of endorsement policy.
- *statedb\_read*: time spent in reading from the state database. For LevelDB, this is always zero as the state database reads are integrated with the *mvcc* step.
- *mvcc*: time spent in executing the *mvcc* checks. For LevelDB, this latency also includes the time spent in reading from the state database.
- *ledger\_write*: time spent in committing the block to the ledger.
- *statedb\_write*: time spent in committing the write sets of transactions to state database.
- *others*: time spent in miscellaneous operations.

We did not use the Caliper tool to measure the above metrics because (1) It had issues missing block events at higher transaction rates (also reported in [11], [22]), and (2) It can only measure throughput and latency at the client level which does not provide in-depth insights of the validation phase. We instrumented the Fabric code to log timestamps at various points through the validation phase, and then calculated the above metrics after an experiment has finished. Each experiment was repeated 20 times to compute average metrics.

#### D. Analysis of Validation Phase

We run Fabric v1.1 using the setup described above as our baseline for analysis. We changed the number of *vscc* threads and the block size, which are the two most important configuration parameters of the validation phase [11], [23].

1) *LevelDB*: Figure 2 shows the breakdown of validation latency and commit throughput for LevelDB with a block size of 50, and varying *vscc* threads from 16 to 48. We make the following observations here:

- The *vscc* operation even with multiple threads is still the bottleneck. The *vscc* latency reduces with an increase in the number of threads; however, the overall throughput only improves slightly because all the other latencies do not improve with *vscc* threads.
- The block commit is dominated by the writes to the ledger instead of the writes to the state database because LevelDB provides relatively fast accesses.
- The *others* latency is not negligible and in fact, it is comparable to *mvcc* and more than the *statedb\_write* latency. This is because it is dominated by the time spent in writing to the history database.

The results from varying the block size from 50 to 200 are reported in Figure 3, and our earlier observations still hold. The interesting point to note here is that the improvement in throughput is more noticeable because the cost of committing the block to the ledger and state database is better amortized for larger blocks.

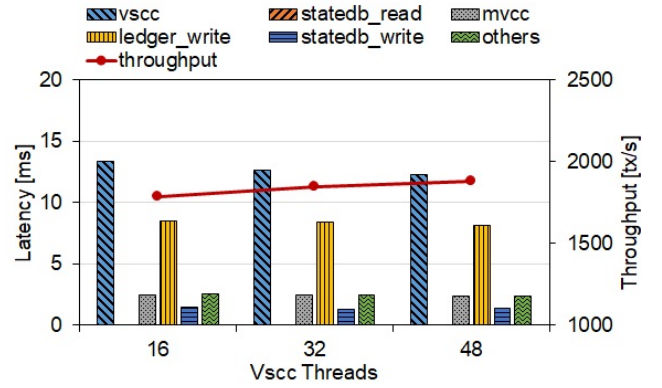


Fig. 2: Original validation phase: LevelDB vs. *vscc* threads.

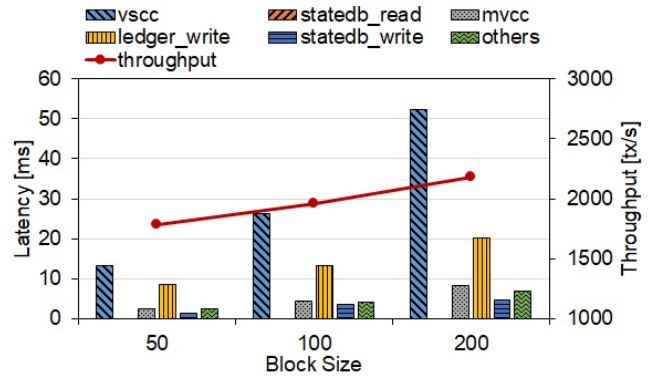


Fig. 3: Original validation phase: LevelDB vs. block size.

2) *CouchDB*: The results for CouchDB with varying *vscc* threads are presented in Figure 4. The distribution of the latencies is quite different compared to LevelDB. The noteworthy observations here are:

- The *vscc* latency is the bottleneck, and is almost 4× that of the *vscc* latency when LevelDB is used. This comes as a surprise since the *vscc* operation should be independent of the type of state database. It turns out that the chaincode information such as version, endorsement policy, etc. is stored in the state database. Since *vscc* operation enforces endorsement policy, for each transaction, it accesses the state database to retrieve this information. Given that CouchDB is accessed through REST API which is slow, the *vscc* latency is much more when compared to LevelDB.
- It is the state database accesses where most of the time is spent. The *statedb\_read* latency is significantly high. Furthermore, the block commit is dominated by the *statedb\_write* latency instead of the *ledger\_write* in contrast to LevelDB.
- The *mvcc* and *others* latencies are not significant, and are more or less the same as LevelDB.

The latency and throughput trends are similar when the block size is changed from 50 to 200, which are shown in Figure 5.

#### E. Optimized Validation Phase

The observations from our analysis can be summarized into two main points: (1) Only the *vscc* operation benefits

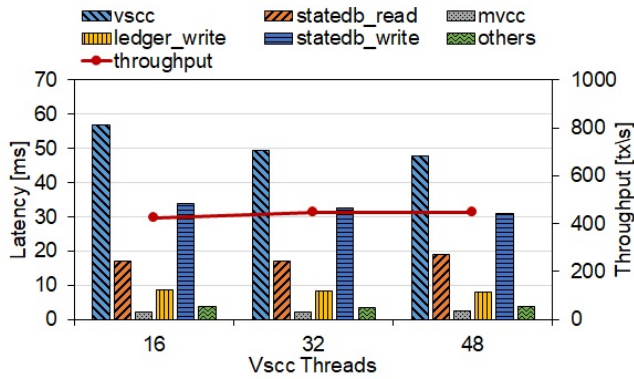


Fig. 4: Original validation phase: CouchDB vs. vscs threads.

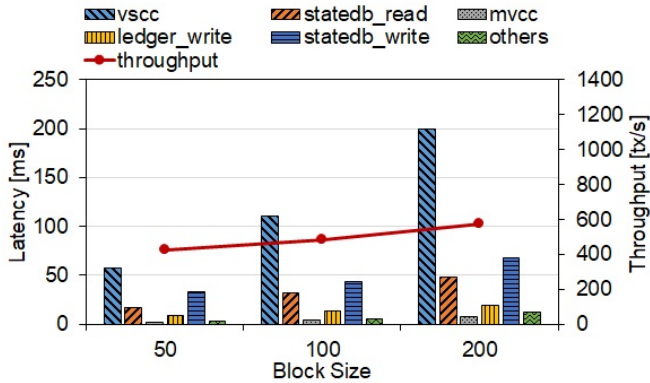


Fig. 5: Original validation phase: CouchDB vs. block size.

from multiple CPUs as it uses multiple threads to validate transactions in parallel. The other operations of the validation phase are sequential, and do not fully utilize the available computational resources, and (2) Accessing state database for information that does not change very often can have a significant negative impact on the performance.

We re-architect the validation phase by executing as many operations of the validation phase in parallel as possible and use a cache for chaincode information. Our optimized validation phase is depicted in Figure 6. Like the original validation phase, in step 1, all the transactions are syntactically validated. If a transaction is ill-formed, then its flag is set to invalid. Afterwards, in step 2, we initiate both the *vscs* and the state database read operations in parallel. Recall that the state database read operation here is only applicable to CouchDB, and is ignored for LevelDB. The motivation is that the latencies of these two operations can be overlapped with each other to reduce the overall block validation latency (see Figure 4).

The *vscs* operation is modified to leverage a chaincode cache (marked as *cc cache*) implemented as a map of chaincode id to its detailed information such as chaincode name, version, endorsement policy, etc. Typically, only a few tens of chaincodes will be instantiated, so their information can be cached for fast accesses by avoiding state database reads. During validation of each transaction, its associated chaincode is searched in the cache. If the cache lookup results in a miss, then state database is accessed to retrieve the chaincode

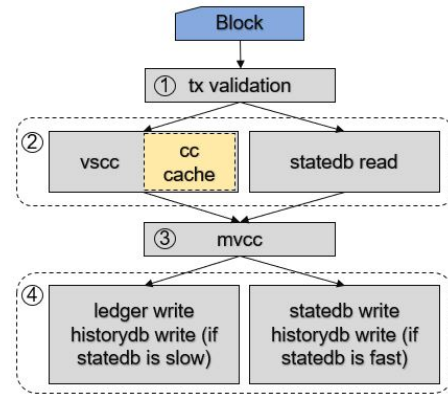


Fig. 6: Optimized Validation Phase for Hyperledger Fabric.

information and the cache is updated. When a chaincode is upgraded, its entry from the cache is deleted so that the new information can be retrieved from the state database again. Another possibility is to clear the chaincode cache at the start of each block (instead of clearing cache at chaincode upgrade). However, for this implementation, the block size should be large enough to amortize the cost of clearing and re-populating the cache for every block.

The state database read operation computes the read sets of all the syntactically valid transactions by reading from the state database. It is possible that some of these syntactically valid transactions might be marked as invalid by the *vscs* operation. This is addressed in step 3, where the *mvcc* checks are only applied to transactions that have been validated by the *vscs* operation, discarding all other invalid transactions. In other words, the *mvcc* operation discards the invalid transactions by ignoring their read sets computed during the state database read operation. The output of the *mvcc* operation will be exactly the same as the original validation phase.

In the final step 4, the block is committed to the ledger and the two databases in parallel. First, the ledger write and state database write operations are executed in parallel to hide the latency of the faster operation (see Figures 2 and 4). Then, for LevelDB (fast database), the write to history database is initiated after the state database write operation. For CouchDB (slow database), history database write operation is combined with the ledger write operation instead.

It is possible that the write to ledger succeeds while the state (or history) database write operation fails, then the peer raises a panic error following the existing recovery mechanism [19]. On restart, the peer fetches the failed/missed blocks from other peers to reconstruct the ledger and the databases. In our optimized validation phase, it is also possible that the writes to either one or both the databases succeeds while the ledger write operation fails, then the databases will be inconsistent with the ledger. In this scenario, we propose to retry ledger write for a number of times before reporting a panic error. As explained above, the peer will reconstruct the ledger and databases on restart. Alternatively, the databases could be rolled back by reverting to previous version/snapshot or re-writing the old values. Note that an in-depth study of crash/fault tolerance of a peer in case of write failures will be addressed in a future work.

## IV. EXPERIMENTAL RESULTS

### A. Implementation

We used the same setup as described in Section III to evaluate the optimized validation phase. In our implementation of the optimized validation phase, we did not separate the syntactic validation from *vscc* validation due to the significant code refactor effort required. This did not affect our experiments as all the transactions were well-formed. Likewise, we did not implement the cache update mechanism when a chaincode is upgraded as our experiments used the same chaincode during the entire run. However, it should be noted that the Fabric code refactor proposal in [17] for the next release will greatly simplify the implementation of our proposals.

All the parallel operations were implemented as goroutines. We added two more latencies, in addition to the ones described in Section III-C:

- *vscc\_statedb\_read*: total time spent in syntactic and *vscc* validation of transactions, and reading from the state database when these operations are executed in parallel.
- *ledger\_statedb\_write*: time spent in committing the block to the ledger, and state and history databases when these operations are executed in parallel.

### B. Results

1) *LevelDB*: The results for LevelDB with block size of 50 and varying *vscc* threads are reported in Figure 7. As expected, the latencies for *vscc*, state database read and *mvcc* operations are similar to the original validation phase (see Figure 2). Furthermore, the *vscc\_statedb\_read* latency is more or less the same as the *vscc* latency, which means the overheads of parallel execution are negligible.

More importantly, the *statedb\_write* latency is completely hidden by the *ledger\_write* latency. The *statedb\_write* latency in the optimized validation phase is more than the original phase because it includes the time taken to write to the history database as well. As a result, the *others* latency is now almost zero. For 16 *vscc* threads, the throughput increased from 1,784 to 2,124 transactions/second which is a  $1.2\times$  improvement.

The validation latency and commit throughput for varying block sizes are presented in Figure 8. The benefits of parallel execution are evident again as the *ledger\_statedb\_write* latency is about 30% lower than that of the sum of *ledger\_write* + *statedb\_write* + *others* latencies from Figure 3 (8.5 ms compared to 12.5 ms for block size of 50). For larger block sizes, the throughput improvement is better ( $1.3\times$  for 200 compared to  $1.2\times$  for 50) because more transactions result in chaincode cache hits than misses. To summarize, our optimized validation phase achieves a  $1.3\times$  improvement in throughput (2,835 compared to 2,178 transactions/second) with 16 *vscc* threads and block size of 200.

2) *CouchDB*: The CouchDB results are even more interesting, and are reported in Figure 9 for block size of 50 and varying *vscc* threads. First, the *vscc* latency decreased significantly, by about  $4\times$ , from 57 ms in Figure 4 to about 14 ms. This is due to the chaincode cache we introduced in Section III, which enables the *vscc* operation with CouchDB to perform on par with LevelDB. Furthermore, the *statedb\_read* latency completely hides the *vscc* latency, making the *vscc\_statedb\_read*

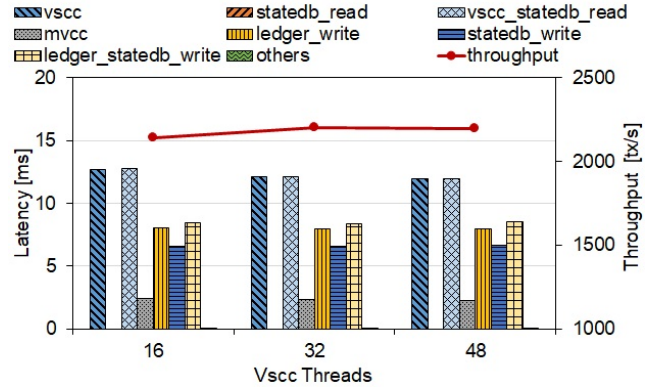


Fig. 7: Optimized validation phase: LevelDB vs. *vscc* threads.

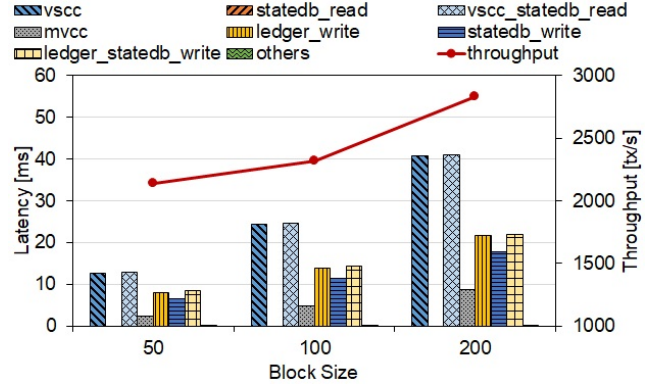


Fig. 8: Optimized validation phase: LevelDB vs. block size.

latency more or less the same as *statedb\_read* latency. An interesting point to note here is that the *statedb\_read* latency is more than its counterpart in Figure 4. The reason is that both the *vscc* and state database read operations are computationally intensive and compete for the available CPU resources. However, the overall *vscc\_statedb\_read* latency is still less than the sum of *vscc* and *statedb\_read* latencies from Figure 4.

On the block commit operations, the *ledger\_write* latency is completely hidden by the *statedb\_write* latency. The *ledger\_write* latency in this case also includes the time taken to write to the history database, and hence is more than the *ledger\_write* latency in Figure 4. Therefore, the *others* latency is almost negligible. Overall, the throughput improvement is about  $2\times$  for 16 *vscc* threads, where the throughput increased from 424 to 841 transactions/second.

Similar improvements are observed across varying block sizes as reported in Figure 10. Most importantly, our optimizations can achieve a commit throughput of about 1,200 transactions/second with a block size of 200. Most of this speedup resulted from the reduction in *vscc* latency and its parallel execution with state database read ( $3\times$  compared to Figure 5) and overlapped writes during block commit ( $1.6\times$  compared to Figure 5). For perspective, the best throughput reported with CouchDB in literature [23] is 700 transactions/second with a block size of 500 transactions and peers with 32 vCPUs (in our setup peers run on 16 vCPU machines).

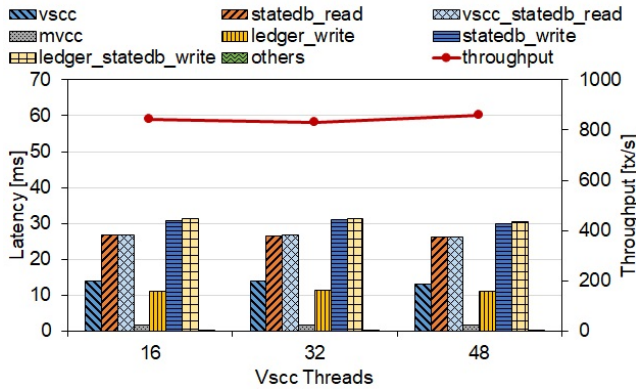


Fig. 9: Optimized validation phase: CouchDB vs. vsc threads.

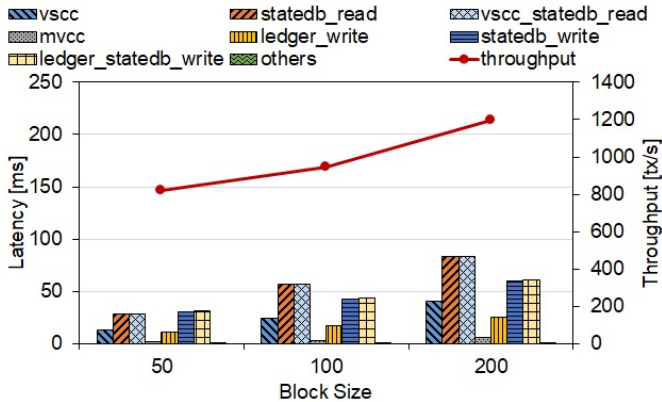


Fig. 10: Optimized validation phase: CouchDB vs. block size.

### C. Remarks

Based on our results, we make the following remarks for the optimized validation phase:

- The chaincode cache in *vsc* operation benefits the most when CouchDB is the state database and higher block sizes are used. The expensive CouchDB accesses are avoided due to the cache hits. Since LevelDB itself is relatively fast, the chaincode cache impact is more noticeable with LevelDB when higher block sizes are used as more database lookups are served from the cache. We believe that such a cache could be generalized to implement a generic caching layer for the state database [16].
- The parallel execution of *vsc* and state database read operations again benefit CouchDB. In general, this optimization is geared towards state databases that are slow and allow bulk read option.
- The parallel execution of writes to ledger, state database and history database during block commit benefits both LevelDB and CouchDB. This is because either the ledger write or the state database write latency can hide the other two latencies depending on whether the state database is slow or fast.

## V. RELATED WORK

Hyperledger Fabric is a recent platform, and thus has been evolving very rapidly. Here, we survey the most relevant work on benchmarking and improving peer performance. For

performance bottlenecks in consensus, readers are referred to [10], [12], [24].

The detailed architecture of Fabric v1.0 was published in [10], where the authors explained their design choices compared to v0.6, and demonstrated the new architecture’s performance and scalability using a cryptocurrency application. Dinh et al. [14] proposed a framework to evaluate private blockchains and compared Fabric v0.6, Ethereum and Parity. Likewise, the authors in [21] compared the performance of Fabric v0.6 and Ethereum. Nasir et al. [18] compared the performance of Fabric v0.6 and v1.0. The authors in [11] evaluated Fabric v1.0 to show that application level parameters such as transaction size, chaincode size, etc. significantly impact performance. Our work differs from these works as they only evaluate the Fabric platform without proposing any architectural changes or optimizations.

Our work is closest to [15], [22], [23]. Thakkar et al. [23] performed an extensive study of the Fabric v1.0, and found that the major bottlenecks are in repeated deserialization of identities/certificates, sequential validation of transactions during *vsc*, and slow CouchDB accesses. They introduced caching identities, parallel *vsc* and bulk read for CouchDB, which were incorporated in the v1.1 release. Hence, our optimizations are on top of their proposals, and we go one step further by caching chaincode information and running operations other than *vsc* in parallel as well.

In a recent work [15], Gorenflo et al. focused on re-architecting both the orderer and peer in Fabric v1.2. For orderers, they proposed to use only the transaction ids instead of full transactions. For peers, they parallelized *vsc* by running it across multiple blocks, cached unmarshalled blocks, and used an in-memory hash table instead of a state database. The authors in [22] proposed re-ordering transactions and early abort mechanism in the orderers to minimize conflicting transactions in a block. Like these works, we also propose architectural changes, however, our optimizations are focused on improving other operations (such as chaincode accesses, and database reads and writes in parallel) which have not been explored before, and can be combined with these existing works. For example, assume that the in-memory hash table from [15] acts as a caching layer for state database [16], then the parallel reads and writes from state database can be combined with [15] to further improve Fabric’s performance.

## VI. CONCLUSION

In this paper, we conduct a fine-grained evaluation of the validation phase of Hyperledger Fabric, and based on our analysis, we re-architect the validation phase. Our optimized validation phase uses a chaincode cache during validation of transactions, executes state database reads in parallel with validation of transactions, as well as writes to the ledger and databases in parallel. We implemented our optimizations in Fabric v1.1 (also valid for v1.4), and show that for CouchDB the throughput improves by  $2\times$  (from 575 to 1,196 transactions/second), while the improvement with LevelDB is  $1.3\times$  (from 2,178 to 2,835 transactions/second). Our optimizations are orthogonal to previous works [15], [23], and thus can be adopted in a future release of Fabric.

## REFERENCES

- [1] Apache CouchDB. <http://couchdb.apache.org/>.
- [2] Apache Kafka. <http://kafka.apache.org/>.
- [3] Bitcoin. <https://bitcoin.org/en/>.
- [4] Corda. <https://www.corda.net>.
- [5] Ethereum. <https://ethereum.org>.
- [6] Hyperledger Caliper. <https://www.hyperledger.org/projects/caliper>.
- [7] Hyperledger Fabric. <https://www.hyperledger.org/projects/fabric>.
- [8] LevelDB in Go. <https://github.com/syndtr/goleveldb/>.
- [9] Quorum. <https://www.jpmorgan.com/global/Quorum>.
- [10] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *EuroSys*, 2018.
- [11] Arati Baliga, Nitesh Solanki, Shubham Verekar, Amol Pednekar, Pandurang Kamat, and Siddhartha Chatterjee. Performance Characterization of Hyperledger Fabric. In *Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2018.
- [12] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the Age of Blockchains. In *CoRR*, *arXiv:1711.03936*, 2017.
- [13] Hyperledger Blog. Forbes Blockchain 50: Half of the biggest companies deploying blockchain use Hyperledger. [https://www.hyperledger.org/blog/2019/04/18/\\_\\_\\_trashed](https://www.hyperledger.org/blog/2019/04/18/___trashed), 2019.
- [14] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.
- [15] Christian Gorenflo, Stephen Lee, Lukasz Golab, and S. Keshav. Fast-Fabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In *CoRR*, *arXiv:1901.00910*, 2019.
- [16] Hyperledger Fabric JIRA. FAB-103 Cache of the World State for Improved Performance. <https://jira.hyperledger.org/browse/FAB-103>.
- [17] Hyperledger Fabric JIRA. FAB-12221 Validator/Committer Refactor. <https://jira.hyperledger.org/browse/FAB-12221?filter=12526>.
- [18] Qassim Nasir, Ilham A. Qasse, Manar Abu Talib, and Ali Bou Nassif. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks*, 2018.
- [19] Senthil Nathan. Failure and Recovery of StateDB in Hyperledger Fabric v1.1.1. <https://blockchain-fabric.blogspot.com/2018/05/failure-and-recovery-of-statedb-in.html>, 2018.
- [20] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC)*, pages 305–320, 2014.
- [21] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. Performance Analysis of Private Blockchain Platforms in Varying Workloads. In *International Conference on Computer Communication and Networks (ICCCN)*, 2017.
- [22] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. How to Databasify a Blockchain: the Case of Hyperledger Fabric. In *CoRR*, *arXiv:1810.13177*, 2018.
- [23] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In *26th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018.
- [24] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016.