FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second

Christian Gorenflo
University of Waterloo
Waterloo, Canada
Email: cgorenflo@uwaterloo.ca

Stephen Lee
University of Massachusetts
Amherst, USA
Email: stephenlee@cs.umass.edu

Lukasz Golab, S. Keshav
University of Waterloo
Waterloo, Canada
Email: [lgolab, keshav]@uwaterloo.ca

***Abstract*—Blockchain technologies are expected to make a significant impact on a variety of industries. However, one issue holding them back is their limited transaction throughput, especially compared to established solutions such as distributed database systems. In this paper, we re-architect a modern permissioned blockchain system, Hyperledger Fabric, to increase transaction throughput from 3,000 to 20,000 transactions per second. We focus on performance bottlenecks beyond the consensus mechanism, and we propose architectural changes that reduce computation and I/O overhead during transaction ordering and validation to greatly improve throughput. Notably, our optimizations are fully plug-and-play and do not require any interface changes to Hyperledger Fabric.**

I. Introduction

Distributed ledger technologies such as blockchains offer a way to conduct transactions in a secure and verifiable manner without the need for a trusted third party. As such, it is widely believed that blockchains will significantly impact industries ranging from finance and real estate to public administration, energy and transportation [1]. However, in order to be viable in practice, blockchains must support transaction rates comparable to those supported by existing database management systems, which can provide some of the same transactional guarantees.

In contrast to permissionless blockchains, which do not restrict network membership, we focus on *permissioned* blockchains, in which the identities of all participating nodes are known. Permissioned blockchains are suitable for many application domains including finance; e.g., the Ripple[1] blockchain aims to provide a payment network for currency exchange and cross-bank transactions akin to the current SWIFT system.

From a technical standpoint, we observe an important distinction between these two types of blockchains. The trustless nature of permissionless blockchains requires either proofs of work or stake, or expensive global-scale Byzantine-fault-tolerant consensus mechanisms [2]. Much recent work has focused on making these more efficient. On the other hand, permissioned blockchains typically delegate consensus and transaction validation to a selected group of nodes, reducing the burden on consensus algorithms. While even in this setting consensus remains a bottleneck, it is being addressed in recent work [3], [4], motivating us to look beyond consensus to identify further performance improvements.

In this paper, we critically examine the design of Hyperledger Fabric 1.2 since it is reported to be the fastest available open-source permissioned blockchain [5]. While there has been some work on optimizing Hyperledger Fabric, e.g., using aggressive caching [6], we are not aware of any prior work on re-architecting the system as a whole[2]. We hence design and implement several architectural optimizations based on common system design techniques that together improve the end-to-end transaction throughput by a factor of almost 7, from 3,000 to 20,000 transactions per second, while decreasing block latency. Our specific contributions are as follows:

1) *Separating metadata from data:* the consensus layer in Fabric receives whole transactions as input, but only the transaction IDs are required to decide the transaction order. We redesign Fabric's transaction ordering service to work with only the transaction IDs, resulting in greatly increased throughput.
2) *Parallelism and caching:* some aspects of transaction validation can be parallelized while others can benefit from caching transaction data. We redesign Fabric's transaction validation service by aggressively caching unmarshaled blocks at the comitters and by parallelizing as many validation steps as possible, including endorsement policy validation and syntactic verification.
3) *Exploiting the memory hierarchy for fast data access on the critical path*: Fabric's key-value store that maintains world state can be replaced with light-weight in-memory data structures whose lack of durability guarantees can be compensated by the blockchain itself. We redesign Fabric's data management layer around a light-weight hash table that provides faster access to the data on the critical transaction-validation path, deferring storage of immutable blocks to a write-optimized storage cluster.
4) *Resource separation:* the peer roles of committer and endorser vie for resources. We introduce an architecture that moves these roles to separate hardware.

Importantly, our optimizations do not violate any APIs or modularity boundaries of Fabric, and therefore they can be incorporated into the planned release of Fabric version 2.0 [7].

[1] https://ripple.com

[2] Please refer to Section V for a detailed survey of related work.

We also outline several directions for future work, which, together with the optimization we propose, have the potential to reach 50,000 transactions per second as required by a credit card company such as Visa [2].

In the remainder of this paper, Section II gives a brief overview of Hyperledger Fabric, Section III presents our improved design, Section IV discusses experimental results, Section V places our contributions in the context of prior work, and Section VI concludes with directions for future work.

II. Fabric architecture

A component of the open-source Hyperledger project hosted by the Linux Foundation, *Fabric* is one of the most actively developed permissioned blockchain systems [8]. Since Androulaki *et al* [9] describe the transaction flow in detail, we present only a short synopsis, focusing on those parts of the system where we propose improvements in Section III.

To avoid pitfalls with smart-contract determinism and to allow plug-and-play replacement of system components, Fabric is structured differently than other common blockchain systems. Transactions follow an *execute-order-commit* flow pattern instead of the common *order-execute-commit* pattern. Client transactions are first executed in a sandbox to determine their *read-write sets*, i.e., the set of keys read by and written to by the transaction. Transactions are then ordered by an ordering service, and finally validated and committed to the blockchain. This workflow is implemented by nodes that are assigned specific roles, as discussed next.

A. Node types

Clients originate transactions, i.e., reads and writes to the blockchain, that are sent to Fabric *nodes*[3]. Nodes are either *peers* or *orderers*; some peers are also *endorsers*. All peers commit blocks to a local copy of the blockchain and apply the corresponding changes to a *state database* that maintains a snapshot of the current *world state*. Endorser peers are permitted to certify that a transaction is valid according to business rules captured in chaincode, Fabric's version of smart contracts. Orderers are responsible solely for deciding transaction order, not correctness or validity.

B. Transaction flow

A client sends its transaction to some number of endorsers. Each endorser executes the transaction in a sandbox and computes the corresponding read-write set along with the version number of each key that was accessed. Each endorser also uses business rules to validate the correctness of the transaction. The client waits for a sufficient number of endorsements and then sends these responses to the orderers, which implement the ordering service. The orderers first come to a consensus about the order of incoming transactions and then segment the message queue into blocks. Blocks are delivered to peers, who then validate and commit them.

C. Implementation details

To set the stage for a discussion of the improvements in Section III, we now take a closer look at the orderer and peer architecture.

1) Orderer: After receiving responses from endorsing peers, a client creates a *transaction proposal* containing a header and a payload. The header includes the Transaction ID and Channel ID[4]. The payload includes the read-write sets and the corresponding version numbers, and endorsing peers' signatures. The transaction proposal is signed using the client's credentials and sent to the ordering service.

The two goals of the ordering service are (a) to achieve consensus on the transaction order and (b) to deliver blocks containing ordered transactions to the committer peers. Fabric currently uses Apache Kafka, which is based on ZooKeeper [10], for achieving crash-fault-tolerant consensus.

When an orderer receives a transaction proposal, it checks if the client is authorized to submit the transaction. If so, the orderer publishes the transaction proposal to a Kafka cluster, where each Fabric channel is mapped to a Kafka topic to create a corresponding immutable serial order of transactions. Each orderer then assembles the transactions received from Kafka into blocks, based either on the maximum number of transactions allowed per block or a block timeout period. Blocks are signed using the orderer's credentials and delivered to peers using gRPC [11].

2) Peer: On receiving a message from the ordering service a peer first unmarshals the header and metadata of the block and checks its syntactic structure. It then verifies that the signatures of the orderers that created this block conform to the specified policy. A block that fails any of these tests is immediately discarded.

After this initial verification, the block is pushed into a queue, guaranteeing its addition to the blockchain. However, before that happens, blocks go sequentially through two validation steps and a final commit step.

During the first validation step, all transactions in the block are unpacked, their syntax is checked and their endorsements are validated. Transactions that fail this test are flagged as invalid, but are left in the block. At this point, only transactions that were created in good faith are still valid.

In the second validation step, the peer ensures that the interplay between valid transactions does not result in an invalid world state. Recall that every transaction carries a set of keys it needs to read from the world state database (its read set) and a set of keys and values it will write to the database (its write set), along with their version numbers recorded by the endorsers. During the second validation step, every key in a transaction's read and write sets must still have the same version number. A write to that key from any prior transaction updates the version number and invalidates the transaction. This prevents double-spending.

In the last step, the peer writes the block, which now includes validation flags for its transactions, to the file system.

[3] Because of its permissioned nature, all nodes must be known and registered with a membership service provider (MSP), otherwise they will be ignored by other nodes.

[4] Fabric is virtualized into multiple *channels*, identified by the channel ID.

The keys and their values, i.e., the world state, are persisted in either LevelDB or CouchDB, depending on the configuration of the application. Moreover, indices to each block and its transactions are stored in LevelDB to speed up data access.

III. Design

This section presents our changes to the architecture and implementation of Fabric version 1.2. This version was released in July 2018, followed by the release of version 1.3 in September 2018 and 1.4 in January 2019. However, the changes introduced in the recent releases do not interfere with our proposal, so we foresee no difficulties in integrating our work with newer versions. Importantly, our improvements leave the interfaces and responsibilities of the individual modules intact, meaning that our changes are compatible with existing peer or ordering service implementations. Furthermore, our improvements are mutually orthogonal and hence can be implemented individually. For both orderers and peers, we describe our proposals in ascending order from smallest to largest performance impact compared to their respective changes to Fabric.

A. Preliminaries

Using a Byzantine-Fault-Tolerant (BFT) consensus algorithm is a critical performance bottleneck in HyperLedger [2]. This is because BFT consensus algorithms do not scale well with the number of participants. In our work, we chose to look beyond this obvious bottleneck for three reasons:

- Arguably, the use of BFT protocols in permissioned blockchains is not as important as in permissionless systems because all participants are known and incentivized to keep the system running in an honest manner.
- BFT consensus is being extensively studied [12] and we expect higher-throughput solutions to emerge in the next year or two.
- In practice, Fabric 1.2 does not use a BFT consensus protocol, but relies, instead, on Kafka for transaction ordering, as discussed earlier.

For these reasons, the goal of our work is not to improve orderer performance using better BFT consensus algorithms, but to mitigate new issues that arise when the consensus is no longer the bottleneck. We first present two improvement to the ordering service, then a series of improvements to peers.

B. Orderer improvement I: Separate transaction header from payload

In Fabric 1.2, orderers using Apache Kafka send the entire transaction to Kafka for ordering. Transactions can be several kilobytes in length, resulting in high communication overhead which impacts overall performance. However, obtaining consensus on the transaction order only requires transaction IDs, so we can obtain a significant improvement in orderer throughput by sending only transaction IDs to the Kafka cluster.

Specifically, on receiving a transaction from a client, our orderer extracts the transaction ID from the header and publishes this ID to the Kafka cluster. The corresponding payload is stored separately in a local data structure by the orderer and the transaction is reassembled when the ID is received back from Kafka. Subsequently, as in Fabric, the orderer segments sets of transactions into blocks and delivers them to peers. Notably, our approach works with any consensus implementation and does not require any modification to the existing ordering interface, allowing us to leverage existing Fabric clients and peer code.

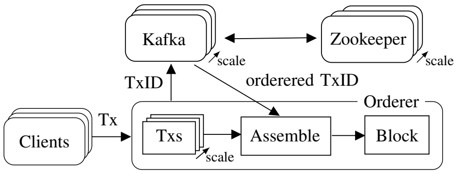

Fig. 1. New orderer architecture. Incoming transactions are processed concurrently. Their TransactionID is sent to the Kafka cluster for ordering. When receiving ordered TransactionIDs back, the orderer reassembles them with their payload and collects them into blocks.

C. Orderer improvement II: Message pipelining

In Fabric 1.2, the ordering service handles incoming transactions from any given client one by one. When a transaction arrives, its corresponding channel is identified, its validity checked against a set of rules and finally it is forwarded to the consensus system, e.g. Kafka; only then can the next transaction be processed. Instead, we implement a pipelined mechanism that can process multiple incoming transactions concurrently, even if they originated from the same client using the same gRPC connection. To do so, we maintain a pool of threads that process incoming requests in parallel, with one thread per incoming request. A thread calls the Kafka API to publish the transaction ID and sends a response to the client when successful. The remainder of the processing done by an orderer is identical to Fabric 1.2.

Fig. 1 summarizes the new orderer design, including the separation of transaction IDs from payloads and the scale out due to parallel message processing.

D. Peer tasks

Recall from Section II-C2 that on receiving a block from an endorser, a Fabric peer carries out the following tasks in sequence:

- Verify legitimacy of the received message
- Validate the block header and each endorsement signature for each transaction in the block
- Validate read and write sets of the transactions
- Update the world state in either LevelDB or CouchDB
- Store the blockchain log in the file system, with corresponding indices in LevelDB

Our goal is to maximize transaction throughput on the critical path of the transaction flow. To this end, we performed an extensive call graph analysis to identify performance bottlenecks.

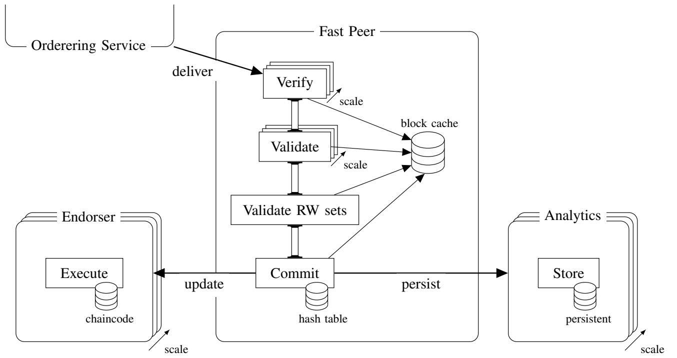

Fig. 2. New peer architecture. The fast peer uses an in-memory hash table to store the world state. The validation pipeline is completely concurrent, validating multiple blocks and their transactions in parallel. The endorser role and the persistent storage are separated into scalable clusters and given validated blocks by the fast peer. All parts of the pipeline make use of unmarshaled blocks in a cache.

We make the following observations. First, the validation of a transaction's read and write set needs fast access to the world state. Thus, we can speed up the process by using an in-memory hash table instead of a database (Section III-E). Second, the blockchain log is not needed for the transaction flow, so we can defer storing it to a dedicated storage and data analytics server at the end of the transaction flow (Section III-F). Third, a peer needs to process new transaction proposals if it is also an endorser. However, the committer and endorser roles are distinct, making it possible to dedicate different physical hardware to each task (Section III-G). Fourth, incoming blocks and transactions must be validated and resolved at the peer. Most importantly, the validation of the state changes through transaction write sets must be done sequentially, blocking all other tasks. Thus, it is important to speed up this task as much as possible (Section III-H).

Finally, significant performance gains can be obtained by caching the results of the *Protocol Buffers* [13] unmarshaling of blocks (Section III-I). We detail this architectural redesign, including the other proposed peer improvements, in Figure 2.

E. Peer improvement I: Replacing the world state database with a hash table

The world state database must be looked up and updated *sequentially* for each transaction to guarantee consistency across all peers. Thus, it is critical that updates to this data store happen at the highest possible transaction rate.

We believe that for common scenarios, such as for tracking of wallets or assets on the ledger, the world state is likely to be relatively small. Even if billions of keys need to be stored, most servers can easily keep them in memory. Therefore, we propose using an in-memory hash table, instead of LevelDB/CouchDB, to store world state. This eliminates hard drive access when updating the world state. It also eliminates costly database system guarantees (i.e., ACID properties) that are unnecessary due to redundancy guarantees of the blockchain itself, further boosting the performance. Naturally, such a replacement is susceptible to node failures due to the use of volatile memory, so the in-memory hash table must be augmented by stable storage. We address this issue in Section III-F.

F. Peer improvement II: Store blocks using a peer cluster

By definition, blocks are immutable. This makes them ideally suited to append-only data stores. By decoupling data storage from the remainder of a peer's tasks, we can envision many types of data stores for blocks and world state backups, including a single server storing blocks and world state backups in its file system, as Fabric does currently; a database or a key-value store such as LevelDB or CouchDB. For maximum scaling, we propose the use of a distributed storage cluster. Note that with this solution, each storage server contains only a fraction of the chain, motivating the use of distributed data processing tools such as Hadoop MapReduce or Spark[5].

G. Peer improvement III: Separate commitment and endorsement

In Fabric 1.2, endorser peers are also responsible for committing blocks. Endorsement is an expensive operation, as is commitment. While concurrent transaction processing on a cluster of endorsers could potentially improve application performance, the additional work to replicate commitments on every new node effectively nullifies the benefits. Therefore, we propose to split these roles.

Specifically, in our design, a committer peer executes the validation pipeline and then sends validated blocks to a cluster of endorsers who only apply the changes to their world state without further validation. This step allows us to free up resources on the peer. Note that such an endorser cluster, which can scale out to meet demand, only splits off the endorsement role of a peer to dedicated hardware. Servers in this cluster are not equivalent to full fledged endorsement peers in Fabric 1.2.

H. Peer improvement IV: Parallelize validation

Both block and transaction header validation, which include checking permissions of the sender, enforcing endorsement policies and syntactic verification, are highly parallelizable. We extend the concurrency efforts of Fabric 1.2 by introducing a complete validation pipeline.

Specifically, for each incoming block, one go-routine is allocated to shepherd it through the block validation phase. Subsequently, each of these go-routines makes use of the go-routine pool that already exists in Fabric 1.2 for transaction validation. Therefore, at any given time, multiple blocks and their transactions are checked for validity in parallel. Finally, all read-write sets are validated sequentially by a single go-routine in the correct order. This enables us to utilize the full potential of multi-core server CPUs.

[5] However, our current implementation does not include such a storage system.

I. Peer improvement V: Cache unmarshaled blocks

Fabric uses gRPC for communication between nodes in the network. To prepare data for transmission, *Protocol Buffers* are used for serialization. To be able to deal with application and software upgrades over time, Fabric's block structure is highly layered, where each layer is marshaled and unmarshaled separately. This leads to a vast amount of memory allocated to convert byte arrays into data structures. Moreover, Fabric 1.2 does not store previously unmarshaled data in a cache, so this work has to be redone whenever the data is needed.

To mitigate this problem, we propose a temporary cache of unmarshaled data. Blocks are stored in the cache while in the validation pipeline and retrieved by block number whenever needed. Once any part of the block becomes unmarshaled, it is stored with the block for reuse. We implement this as a cyclic buffer that is as large as the validation pipeline. Whenever a block is committed, a new block can be admitted to the pipeline and automatically overwrites the existing cache location of the committed block. As the cache is not needed after commitment and it is guaranteed that a new block only arrives after an old block leaves the pipeline, this is a safe operation. Note that unmarshaling only adds data to the cache, it never mutates it. Therefore, lock-free access can be given to all go-routines in the validation pipeline. In a worst-case scenario, multiple go-routines try to access the same (but not yet unmarshaled) data and all proceed to execute the unmarshaling in parallel. Then the last write to the cache wins, which is not a problem because the result would be the same in either case.

Call graph analysis shows that, even with these optimizations, memory (de)allocation due to unmarshaling is still responsible for the biggest share of the overall execution time. This is followed by gRPC call management and cryptographic computations. The last two areas, however, are beyond the scope of this work.

IV. Results

This section presents an experimental performance evaluation of our architectural improvements. We used fifteen local servers connected by a 1 Gbit/s switch. Each server is equipped with two Intel® Xeon® CPU E5-2620 v2 processors at 2.10 GHz, for a total of 24 hardware threads and 64 GB of RAM. We use Fabric 1.2 as the base case and add our improvements step by step for comparison. By default, Fabric is configured to use LevelDB as the peer state database and the orderer stores completed blocks in-memory, rather than on disk. Furthermore, we run the entire system without using docker containers to avoid additional overhead.

While we ensured that our implementation did not change the validation behaviour of Fabric, all tests were run with non-conflicting and valid transactions. This is because valid transactions must go through every validation check and their write sets will be applied to the state database during commitment. In contrast, invalid transactions can be dropped. Thus, our results evaluate the worst case performance.

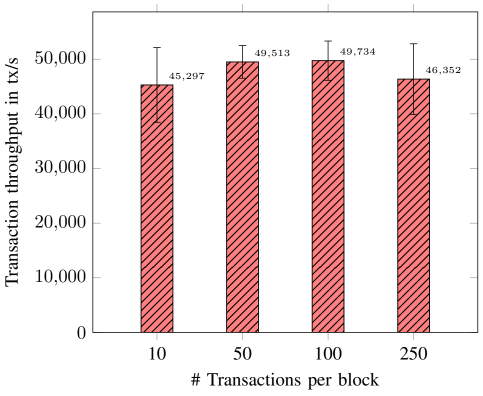

Fig. 3. Throughput via gRPC for different block sizes.

For our experiments which focus specifically on either the orderer or the committer, we isolate the respective system part. In the orderer experiments, we send pre-loaded endorsed transactions from a client to the orderer and have a mock committer simply discard created blocks. Similarly, during the benchmarks for the committer, we send pre-loaded blocks to the committer and create mocks for endorsers and the block store which discard validated blocks.

Then, for the end-to-end setup, we implement the full system: Endorsers endorse transaction proposals from a client based on the replicated world state from validated blocks of the committer; the orderer creates blocks from endorsed transactions and sends them to the committer; the committer validates and commits changes to its in-memory world state and sends validated blocks to the endorsers and the block storage; the block storage uses the Fabric 1.2 data management to store blocks in its file system and the state in LevelDB. We do not, however, implement a distributed block store for scalable analytics; that is beyond the scope of this work.

For a fair comparison, we used the same transaction chaincode for all experiments: Each transaction simulates a money transfer from one account to another, reading and making changes to two keys in the state database. These transactions carry a payload of 2.9 KB, which is typical [3]. Furthermore, we use the default endorsement policy of accepting a single endorser signature.

A. Block transfer via gRPC

We start by benchmarking the gRPC performance. We precreated valid blocks with different numbers of transactions in them, sent them through the Fabric gRPC interface from an orderer to a peer, and then immediately discarded them. The results of this experiment are shown in Figure 3.

We find that for block sizes from 10 to 250 transactions, which are the sizes that lead to the best performance in the following sections, a transaction throughput rate of more that 40,000 transactions/s is sustainable. Comparing this with the results from our end-to-end tests in Section IV-D, it is clear that in our environment, network bandwidth and the

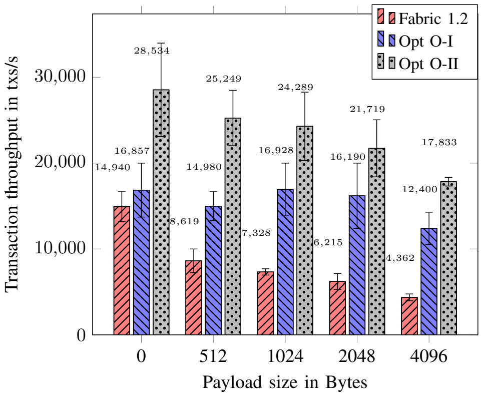

Fig. 4. Effect of payload size on orderer throughput.

1 Gbit/s switch used in the server rack are not the performance bottlenecks.

B. Orderer throughput as a function of message size

In this experiment, we set up multiple clients that send transactions to the orderer and monitor the time it takes to send 100,000 transactions. We evaluate the rate at which an orderer can order transactions in Fabric 1.2 and compare it to our improvements:

- **Opt O-I**: only Transaction ID is published to Kafka (Section III-B)
- **Opt O-II**: parallelized incoming transaction proposals from clients (Section III-C)

Figure 4 shows the transaction throughput for different payload sizes. In Fabric 1.2, transaction throughput decreases as payload size increases due to the overhead of sending large messages to Kafka. However, when we send only the transaction ID to Kafka (Opt O-1), we can almost triple the average throughput (2.8×) for a payload size of 4096 KB. Adding optimization O-2 leads to an average throughput of 4× over the base Fabric 1.2. In particular, for the 2 KB payload size, we increase orderer performance from 6,215 transactions/s to 21,719 transactions/s, a ratio of nearly 3.5x.

C. Peer experiments

In this section, we describe tests on a single peer in isolation (we present the results of an end-to-end evaluation in Section IV-D). Here, we pre-computed blocks and sent them to a peer as we did in the gRPC experiments in Section IV-A. The peer then completely validates and commits the blocks.

The three configurations shown in the figures compared to Fabric 1.2 *cumulatively* incorporate our improvements (i.e., Opt P-II incorporate Opt P-I, and Opt P-III incorporates both prior improvements):

- **Opt P-I** LevelDB replaced by an in-memory hash table
- **Opt P-II** Validation and commitment completely parallelized; block storage and endorsement offloaded to a separate storage server via remote gRPC call
- **Opt P-III** All unmarshaled data cached and accessible to the entire validation/commitment pipeline.

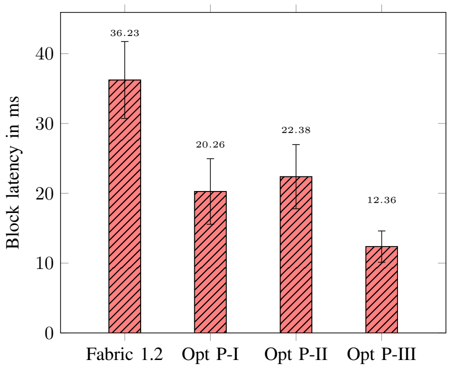

Fig. 5. Impact of our optimizations on peer block latency.

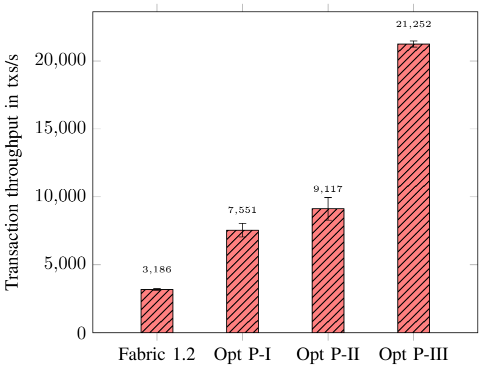

Fig. 6. Impact of our optimizations on peer throughput.

1) Experiments with fixed block sizes: Fig. 5 and 6 show the results from validation and commitment of 100,000 transactions for a single run, repeated 1000 times. Transactions were collected into blocks of 100 transactions[6]. We first discuss latency, then throughput.

Because of batching, we show the latency per block, rather than per-transaction latency. The results are in line with our self-imposed goal of introducing no additional latency as a result of increasing throughput; in fact our performance improvements decrease peer latency to a third of its original value (note that these experiments do not take network delay into account). Although the pipelining introduced in Opt P-II generates some additional latency, the other optimizations more than compensate for it.

By using a hash table for state storage (Opt P-I), we are able to more than double the throughput of a Fabric 1.2 peer from about 3200 to more than 7500 transactions/s. Parallelizing validation (Opt P-II) adds an improvement of roughly 2,000 transactions per second. This is because, as Figure 2 shows, only the first two validation steps can be parallelized and scaled out. Thus, the whole pipeline performance is governed by the throughput of read and write set validation and commitment. Although commitment is almost free when using Opt P-I, it

[6] We experimentally determined that peer throughput was maximized at this block size.

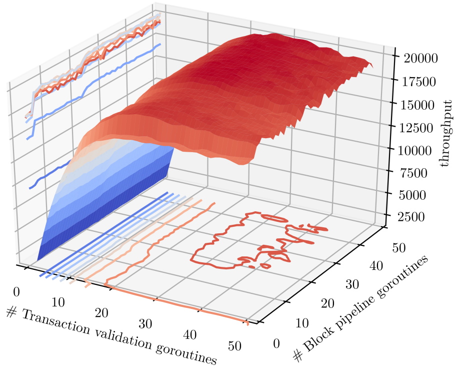

Fig. 7. Parameter sensitivity study for blocks containing 100 transactions and a server with 24 CPU cores. We scale the number of blocks that are validated in parallel and the number of transactions per block that are validated in parallel independently.

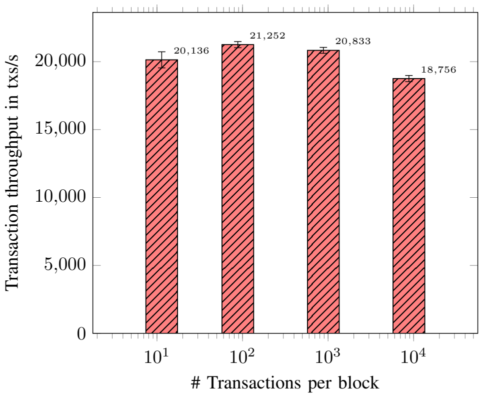

Fig. 8. Throughput dependence on block size for optimally tuned configuration

is not until the introduction of the unmarshaling cache in Opt P-III that Opt P-II pays off. The cache drastically reduces the amount of work for the CPU, freeing up resources to validate additional blocks in parallel. With all peer optimizations taken together, we increase a peer's commit performance by 7x from about 3200 transactions/s to over 21,000 transactions/s.

2) Parameter sensitivity: As discussed in Section IV-C, parallelizing block and transaction validation at the peer is critical. However, it is not clear how much parallelism is necessary to maximize performance. Hence, we explore the degree to which a peer's performance can be tuned by varying two parameters:

- The number of go-routines concurrently shepherding blocks in the validation pipeline
- The number of go-routines concurrently validating transactions

We controlled the number of active go-routines in the system using semaphores, while allowing multiple blocks to concurrently enter the validation pipeline. This allows us to control the level of parallelism in block header validation and transaction validation through two separate go-routine pools.

For a block size of 100 transactions, Figure 7 shows the throughput when varying the number of go-routines. The total number of threads in the validation pipeline is given by the sum of the two independent axes. For example, we achieve maximum throughput for 25 transaction validation go-routines and 31 concurrent blocks in the pipeline, totalling 56 go-routines for the pipeline. While we see a small performance degradation through thread management overhead when there are too many threads, the penalty for starving the CPU with too few parallel executions is drastic. Therefore, we suggest as a default that there be at least twice as many go-routines as there are physical threads in a given machine.

We now investigate the dependence of peer throughput on block size. Each block size experiment was performed with the best tuned go-routine parameters from the previous test. All configurations used around 24 ± 2 transaction validation go-routines and 30 ± 3 blocks in the pipeline. Again, we split 100,000 transactions among blocks of a given size for a single benchmark run and repeated the experiment 1,000 times. We chose to scan the block size space on a logarithmic scale to get an overview of a wide spectrum.

The results are shown in Figure 8. We find that a block size of 100 transactions/block gives the best throughput with just over 21,000 transactions per second. We also investigated small deviations from this block size. We found that performance differences for block sizes between 50 and 500 were very minor, so we opted to fix the block size to 100 transactions.

D. End-to-end throughput

We now discuss the end-to-end throughput achieved by combining all of our optimizations, i.e., Opt. O-II combined with Opt. P-III, compared to our measurements of unmodified Fabric 1.2.

We set up a single orderer that uses a cluster of three ZooKeeper servers and three Kafka servers, with the default topic replication factor of three, and connect it to a peer. Blocks from this peer are sent to a single data storage server that stores world state in LevelDB and blocks in the file system. For scale-out, five endorsers replicate the peer state and provide sufficient throughput to deal with client endorsement load. Finally, a client is installed on its own server; this client requests endorsements from the five endorser servers and sends endorsed transactions to the ordering service. This uses a total of fifteen servers connected to the same 1 Gbit/s switch in our local data center.

We send a total of 100,000 endorsed transactions from the client to the orderer, which batches them to blocks of size 100 and delivers them to the peer. To estimate throughput,

TABLE I
END-TO-END THROUGHPUT

	Fabric 1.2	FastFabric
Transactions/s	3185 ± 62	19112 ± 811

we measure the time between committed blocks on the peer and take the mean over a single run. These runs are repeated 100 times. Table I shows a significant improvement of 6-7× compared to our baseline Fabric 1.2 benchmark.

V. Related Work

Hyperledger Fabric is a recent system that is still undergoing rapid development and significant changes in its architecture. Hence, there is relatively little work on the performance analysis of the system or suggestions for architectural improvements. Here, we survey recent work on techniques to improve the performance of Fabric.

The work closest to ours is by Thakkar *et al* [6] who study the impact of various configuration parameters on the performance of Fabric. They find that the major bottlenecks are repeated validation of X.509 certificates during endorsement policy verification, sequential policy validation of transactions in a block, and state validation during the commit phase. They introduce aggressive caching of verified endorsement certificates (incorporated into Fabric version 1.1, hence part of our evaluation), parallel verification of endorsement policies, and batched state validation and commitment. These improvements increase overall throughput by a factor of 16. We also parallelize verification at the commiters and go one step further in replacing the state database with a more efficient data structure, a hash table.

In recent work, Sharma *et al* [14] study the use of database techniques, i.e., transaction reordering and early abort, to improve the performance of Fabric. Some of their ideas related to early identification of conflicting transactions are orthogonal to ours and can be incorporated into our solution. However, some ideas, such as having the orderers drop conflicting transactions, are not compatible with our solution. First, we deliberately do not send transaction read-write sets to the orderers, only transaction IDs. Second, we chose to keep to Fabric's design goal of allocating different tasks to different types of nodes, so our orderers do not examine the contents of the read and write sets. An interesting direction for future work would be to compare these two approaches under different transaction workloads to understand when the overhead of sending full transaction details to the orderers is worth the early pruning of conflicting transactions.

It is well known that Fabric's orderer component can be a performance bottleneck due to the message communication overhead of Byzantine fault tolerant (BFT) consensus protocols. Therefore, it is important to use an efficient implementation of a BFT protocol in the orderer. Sousa *et al* [3] study the use of the well-known BFT-SMART [15] implementation as a part of Fabric and shows that, using this implementation within a single datacenter, a throughput of up to 30,000 transactions/second is achievable. However, unlike our work, the commiter component is not benchmarked and the end-to-end performance is not addressed.

Androulaki *et al* [16] study the use of channels for scaling Fabric. However, this work does not present a performance evaluation to quantitatively establish the benefits from their approach.

Raman *et al* [17] study the use of lossy compression to reduce the communication cost of sharing state between Fabric endorsers and validators when a blockchain is used for storing intermediate results arising from the analysis of large datasets. However, their approach is only applicable to scenarios which are insensitive to lossy compression, which is not the general case for blockchain-based applications.

Some studies have examined the performance of Fabric without suggesting internal architectural changes to the underlying system. For example, Dinh *et al* use BlockBench [5], a tool to study the performance of private blockchains, to study the performance of Fabric, comparing it with that of Ethereum and Parity. They found that the version of Fabric they studied did not scale beyond 16 nodes due to congestion in the message channel. Nasir *et al* [18] compare the performance of Fabric 0.6 and 1.0, finding, unsurprisingly, that the 1.0 version outperforms the 0.6 version. Baliga *et al* [19] showed that application-level parameters such as the read-write set size of the transaction and chaincode and event payload sizes significantly impact transaction latency. Similarly, Pongnumkul *et al* [20] compare the performance of Fabric and Ethereum for a cryptocurrency workload, finding that Fabric outperforms Ethereum in all metrics. Bergman [21] compares the performance of Fabric to Apache Cassandra in similar environments and finds that, for a small number of peer nodes, Fabric has a lower latency for linearizable transactions in read-heavy workloads than Cassandra. On the other hand, with a larger number of nodes, or write-heavy workloads, Cassandra has better performance.

VI. Conclusions

The main contribution of this work is to show how a permissioned blockchain framework such as Hyperledger Fabric can be re-engineered to support nearly 20,000 transactions per second, a factor of almost 7 better than prior work. We accomplished this goal by implementing a series of independent optimizations focusing on I/O, caching, parallelism and efficient data access. In our design, orderers only receive transaction IDs instead of full transactions, and validation on peers is heavily parallelized. We also use aggressive caching and we leverage light-weight data structures for fast data access on the critical path. In future work, we hope to further improve the performance of Hyperledger Fabric by:

- Incorporating an efficient BFT consensus algorithm such as RCanopus [22]
- Speeding up the extraction of transaction IDs for the orderers without unpacking the entire transaction headers
- Replacing the existing cryptographic computation library with a more efficient one
- Providing further parallelism by assigning a separate ordering and fast peer server per channel
- Implementing an efficient data analytics layer using a distributed framework such as Apache Spark [23]

References

[1] V. Espinel, D. O'Halloran, E. Brynjolfsson, and D. O'Sullivan, "Deep shift, technology tipping points and societal impact," in *New York: World Economic Forum–Global Agenda Council on the Future of Software & Society (REF 310815)*, 2015.

[2] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International Workshop on Open Problems in Network Security*. Springer, 2015, pp. 112–125.

[3] J. Sousa, A. Bessani, and M. Vukolic, "A Byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 51–58.

[4] M. Yin, D. Malkhi, M. K. Reiter, G. Golan Gueta, and I. Abraham, "HotStuff: BFT Consensus in the Lens of Blockchain," *arXiv preprint arXiv:1803.05069*, 2018.

[5] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A Framework for Analyzing Private Blockchains," *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*, pp. 1085–1100, 2017.

[6] P. Thakkar, S. Nathan, and B. Vishwanathan, "Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform," *arXiv*, 2018.

[7] Hyperledger Fabric, "[FAB-12221] Validator/Committer refactor - Hyperledger JIRA." [Online]. Available: https://jira.hyperledger.org/browse/FAB-12221?filter=12526

[8] C. Cachin, "Architecture of the hyperledger blockchain fabric," in *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*, vol. 310, 2016.

[9] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18*, pp. 1–15, 2018.

[10] Apache Foundation, "Apache Kafka, A Distributed Streaming Platform," 2018. [Online]. Available: https://kafka.apache.org/ (Accessed 2018-12-05).

[11] Cloud Native Computing Foundation, "gRPC: A high performance, open-source universal RPC framework," 2018. [Online]. Available: https://grpc.io/

[12] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Consensus in the age of blockchains," *arXiv preprint arXiv:1711.03936*, 2017.

[13] Google Developers, "Protocol Buffers — Google Developers," 2018. [Online]. Available: https://developers.google.com/protocol-buffers/?hl=en

[14] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "How to databasify a blockchain: the case of hyperledger fabric," *arXiv preprint arXiv:1810.13177*, 2018.

[15] A. Bessani, J. Sousa, and E. Alchieri, "State machine replication for the masses with BFT-SMART," in *DSN*, 2014, pp. 355–362.

[16] E. Androulaki, C. Cachin, A. De Caro, and E. Kokoris-Kogias, "Channels: Horizontal scaling and confidentiality on permissioned blockchains," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 111–131.

[17] R. K. Raman, R. Vaculin, M. Hind, S. L. Remy, E. K. Pissadaki, N. K. Bore, R. Daneshvar, B. Srivastava, and K. R. Varshney, "Trusted multi-party computation and verifiable simulations: A scalable blockchain approach," *arXiv preprint arXiv:1809.08438*, 2018.

[18] Q. Nasir, I. A. Qasse, M. Abu Talib, and A. B. Nassif, "Performance analysis of hyperledger fabric platforms," *Security and Communication Networks*, vol. 2018, 2018.

[19] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance Characterization of Hyperledger Fabric," in *Crypto Valley Conference on Blockchain Technology, CVCBT 2018*, 2018.

[20] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, "Performance analysis of private blockchain platforms in varying workloads," in *2017 26th International Conference on Computer Communications and Networks, ICCCN 2017*. IEEE, 7 2017, pp. 1–6.

[21] S. Bergman, "Permissioned blockchains and distributed databases: A performance study," Master's thesis, Linkoping University, 2018.

[22] S. Keshav, W. M. Golab, B. Wong, S. Rizvi, and S. Gorbunov, "RCanopus: Making canopus resilient to failures and byzantine faults," *arXiv preprint arXiv:1810.09300*, 2018.

[23] Apache Foundation, "Apache Spark - Unified Analytics Engine for Big Data," 2018. [Online]. Available: https://spark.apache.org