2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS) | 978-1-6654-3924-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/BRAINS52497.2021.9569780

Partial Order Transactions on Permissioned Blockchains for enhanced Scalability

Krishnasuri Narayanam
IBM Research, India
knaraya3@in.ibm.com

Akshar Kaul
IBM Research, India
akshar.kaul@in.ibm.com

Ken Kumar*
ken.iitkgp.13@gmail.com

Pankaj Dayama
IBM Research, India
pankajdayama@in.ibm.com

***Abstract*—Ordering service in a permissioned blockchain platform like Hyperledger Fabric groups the transactions into various blocks. These blocks are linked together via a hash chain to provide immutability. Transactions with less block height get validated before the transactions with more block height. Within a single block, the transactions get validated in the same order they are arranged inside the block. This structure imposes a total order among transactions. This total ordering ensures that all the peer nodes reach the same final state once all the transactions are validated. This total ordering prevents parallel validation of transactions. A given transaction may not be dependent on all the transactions preceding it. However, such a transaction still cannot be validated before completion of validation for all the preceding transactions. This paper presents a blockchain system to support partial ordering among non-dependent transactions. The proposed blockchain system aims to achieve high transaction throughput by enabling parallel transaction validation and reducing the transaction turnaround time by eliminating the need for the transactions to wait for non-dependent predecessor transactions. The proposed blockchain system ensures that all peer nodes reach the same final state even if the order in which they validate the transactions is different.**

***Index Terms*—Transaction partial order, blockchain, consensus, throughput, latency.**

I. Introduction

Building blockchain systems that can scale well even under high input transaction rates is an important problem that is still evolving. Addressing this scalability problem can help increase blockchain adoption in providing solutions to various real-world business problems.

In this paper, we carry out a study on the scalability of permissioned blockchains with execute-order-validate architecture paradigm (e.g., Hyperledger Fabric [1]). The scalability of such a blockchain network depends on the efficiency of all three phases of the blockchain protocol. Hence, investigating the alternate approaches to enhance the efficiency of the validate phase is as important as the study on the scalability of consensus protocols in the context of increasing the efficiency of the ordering phase (e.g., [2]).

Sharding ([3], [4]) is a technique used to enhance the scalability of blockchains, by splitting the network into multiple committees knows as shards. Since the number of the consensus nodes is reduced within each committee, this results in a reduction in the communication complexity which is key to improving the efficiency of the ordering service. However, there are many challenges during this process like splitting the nodes uniformly into shards such that each shard has a majority of honest nodes with high probability.

Organizing the transactions in a block using a directed acyclic graph (DAG) [5] structure created based on inter-dependencies among the transactions enhances the transaction throughput during the validation phase of the blockchain system by letting parallel validation of transactions within a block. However, since the number of transactions in a block is limited, the throughput improvement obtained is also limited (please note that the use of DAG here is different from the context used in public permissionless blockchains [6], [7]).

Our work enhances the efficiency of the validate phase of blockchains with execute-order-validate architecture. In particular, we address the limitation in the DAG-based approaches in the literature by relaxing the total order among the transactions and enabling parallel validation of the transactions that belong to different blocks of the blockchain as well (i.e., transaction validation parallelism across blocks in addition to transaction validation parallelism within a block). Moreover, the DAG-based approaches require that each of the network peers independently carries out the DAG analysis of the transactions in a block before the block validation. On the other hand, our approach enables transaction validation parallelism across blocks without the need for any such additional processing.

A. Illustrative example

We illustrate the summary of our contributions using a simple example. Consider *four* transactions *Tx1, Tx2, Tx3* and *Tx4* with the following dependencies:

- *Tx2* and *Tx3* depend on *Tx1*
- *Tx4* depends on *Tx2* and *Tx3*
- *Tx2* and *Tx3* are independent

Dependency between various transactions is determined by analyzing their read-write sets.

- Transactions *Ty* and *Tz* are said to be independent if read set of *Ty* is disjoint from write set of *Tz*, and vice versa.
- A transaction *Ty* is said to be dependent on *Tz* if the read set of *Ty* intersects with write set of *Tz*.

Consider that the transactions *Tx1 to Tx4* get submitted to a blockchain platform whose ordering service is configured

* The author contributed during his association with IBM

978-1-6654-3924-4/21/$31.00 ©2021 IEEE

to include one transaction per block. Blockchain platforms, like Hyperledger Fabric, currently support only total ordering among the transactions. Hence, the ordering service cuts blocks either as in Fig.1(a) or in Fig.1(b). In contrast, our solution allows partial ordering among the transactions to enable the ordering service to cut the blocks, as shown in Fig.1(c). The advantages with the proposed approach are *two* fold.

- *Increased transaction throughput:* Peers in the blockchain network which have multiple CPUs can perform the validation of blocks without dependencies in parallel. For example, with ledger as in Fig.1(c), the validation of block *B2* and block *B3* can happen concurrently.
- *Decreased latency/turnaround time:* Peers in the blockchain network do not need to wait for validation of blocks whose transactions are independent of transactions in the current block. E.g., with ledger as in Fig.1(c), the validation of block *B3* does not need to wait for the validation of block *B2*, and vice versa.

Our solution ensures that a peer that validates the blocks in the order *B1, B2, B3 & B4* arrives at the same world state as a peer that validates the blocks via a different order *B1, B3, B2 & B4*.

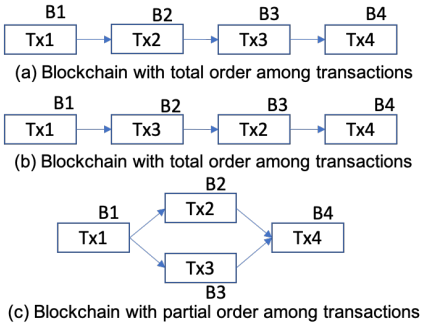


Fig. 1: Non-dependent transactions Tx2 and Tx3 in a blockchain

Our approach is complementary to the work on scalability of consensus protocols and sharding based methods studied in the blockchain literature. Our proposed methods try to improve the efficiency of transaction validation by the network peers during the validation phase of a execute-order-validate architecture based blockchains, by introducing changes to the blockchain structure. We propose block formation in bulk, which we refer to as the blocks of a slot, as against to individual blocks getting cut by the orderer in Hyperledger Fabric. Thus our approach retains partial order among transactions even after transactions are included in the blocks of a blockchain. This approach lets blockchain peers with multiple threads/CPUs validate the blocks that belong to the same slot in parallel, as against to the traditional blockchains with total order among the transactions where blocks are validated in a sequential manner irrespective of the availability of multiple threads with a network peer. We envisage that this approach will enhance the transaction throughput of permissioned blockchains and reduce the transaction latency. The overhead on the ordering service due to the additional processing needed for the block formation in bulk, as proposed in our paper, is small and can be neglected when compared to the gains in block validation with our approach. In other words, the overall scalability of the execute-order-validate architecture based blockchain system increases with our approach even though the methods proposed to enhance the scalability of validation phase have slight overhead on the ordering phase. The same is evident from the preliminary experiments that we have carried out. Moreover, the savings during the block validation phase can be realized by each peer of the blockchain network under many different scenarios (e.g., peer join/reboot, new block validation, etc.), while the overhead due to our approach only impacts the ordering service at the time of block creation. Hence, the savings in validation phase get multiplied as the size of the network grows (i.e., as the number of peers participating the network increases, or as more number of blocks are added, or both).

In the following sections, first, we discuss related work in this domain and how our contributions are different (Section II). Next, we describe the architectural design of our system along with its various components (Section III) and implementation details (Section IV). Finally, we discuss the observations from some preliminary experimental evaluation that we have carried out (Section V), describe few use-cases that get benefitted with the use of our system (Section VI), and provide summary of our contributions (Section VII).

II. Related Work

There has been a lot of work in improving the performance of permissioned blockchain, specifically Hyperledger Fabric. Enhancing transaction execution parallelism during transaction endorsement in Hyperledger Fabric is studied in [1], which is achieved via concurrent transaction simulation by the endorsing peers as per the smart contract logic. This paper proposes to use multiple endorsing peers with the endorsement policy containing an OR predicate to achieve concurrency. Improving the validation performance is studied in [8], [9]. [8] uses a lock-free approach for transaction isolation while [9] suggests improvements in the database configuration used to store the ledger. Enabling parallelism while validating a single transaction by pipelining various operations was explored in [10]–[12]. Enabling parallelism during validation of transactions belonging to a single block was explored in [13], [14]. A consensus protocol for blockchains with order-execute architecture got studied in [15] with the drawback that the multiple blocks proposed in each tournament can have conflicts. None of the prior work explores the role of orderer in enabling parallelism across blocks by enabling partial order support. Work done in the references pointed in this section is complementary to our proposed work and can be integrated with our work to speed up the validation phase even further.

III. System Design

Hyperledger Fabric is a permissioned blockchain platform. The transaction lifecycle in Fabric follows the execute-order-validate model, as summarized below:

- *Execute:* Client submits a transaction to one or more endorsing peers that execute the transaction and reply with digitally signed read-write sets. The client submits the transaction along with the endorsements to the orderer.
- *Order:* Orderer creates blocks of transactions using a (pluggable) consensus protocol and disseminates the blocks to all the peer nodes.
- Validate: Peer nodes validate the transactions in each block using an application-specific endorsement policy and commit to the ledger.

Orderer in Fabric can use *crash fault tolerant* (CFT) or *byzantine fault tolerant* (BFT) consensus protocols. Orderer creates a strict or total ordering among all the transactions submitted on the network. Each peer node has to validate all the transactions of the blocks it receives. To ensure that all the peer nodes reach the same state after the transactions get validated, all the transactions need to get validated in the same order in which they are packed into a block by the ordering service. This results in the validation being a single-threaded process. Even if a peer node has enough resources to validate multiple transactions concurrently, it cannot do so since the resulting state may not be the correct one.

However, peer nodes can analyze the transactions belonging to a block and divide them into various sets such that there is no dependency between the transactions of different sets. The transactions in different sets can then be validated concurrently. This is extra work that each of the peer nodes will have to perform for every block that is received.

Additionally, a peer node needs to wait for validation of all the previous blocks to finish before it can start validation of the current block.

A. Slots

In this paper, we present a solution to enable peer nodes to validate multiple blocks concurrently. The main idea is to group a set of independent blocks into "slots" such that all the blocks in a single slot can be validated in parallel.

Thus, a *slot* consists of a set of blocks such that there is no dependency between transactions belonging to different blocks that are part of the same slot. Transactions within the same block can be dependent on one another.

The non-dependency of transactions between blocks belonging to the same slot guarantees that all the peer nodes will arrive at the same final state irrespective of the order in which the blocks belonging to the same slot are validated. This enables a peer node to validate the blocks in parallel and utilize its resources efficiently.

We illustrate the idea of "slot" using Fig.2. This figure depicts a subset of blocks i.e. blocks B10 to B19, belong to a blockchain. It also depicts the slots to which each of the blocks belongs to. A peer node that validates the blocks belonging to the same slot in parallel (e.g. blocks B14, B15, and B16) will arrive at the same final state which a peer validating the blocks serially (B10, B11 . . . , B19) will arrive at.

In our solution, the task of grouping blocks into appropriate slots is carried out by the orderer. This has the advantage that all peer nodes can take advantage of slots without having to do any additional dependency tracking.

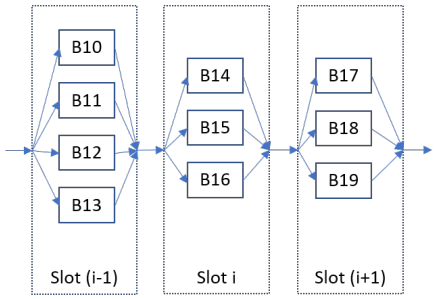


Fig. 2: Slots in blockchain

B. Applicability to other frameworks

Our solution is applicable to any blockchain framework where the transactions are ordered after execution. This is because the read-write sets for any transaction get available to the orderer to carry out the dependency analysis among the transactions and cut the blocks as per our proposed solution. On the other hand, in blockchains with *order-execute* (e.g., Bitcoin or Quorum) or *order-execute-validate* (e.g., Corda or Hashgraph) architecture, our solution is not applicable since the ordering takes place before transaction execution.

IV. Implementation Details

Our solution can be implemented on top of Hyperledger Fabric. We present implementation details of our solution in this section.

A. Tracking Dependencies

To group the blocks into slots, the orderer needs to identify dependencies between the transactions. In Fabric blockchain, endorsed transactions that are submitted to the orderer have associated read-write sets. Orderer can leverage these read-write sets to identify the dependencies between the transactions. Orderer can use one of the below methods to generate the blocks and add one or more blocks to a slot.

1) Set based method: This method ensures that there will not be any inter-block transaction dependencies while there can be intra-block transaction dependencies during slot generation. To achieve this, orderer keeps track of the set of keys accessed by transactions belonging to each block in the current slot. At the beginning of a slot generation, all these block-level key tracking sets get initialized to empty. The orderer then steps linearly through the incoming transactions. For each transaction, it checks if the read-write set of the transaction intersects with any of the current block-level key tracking sets. If there is no intersection, then the transaction gets added to one of the blocks of the current slot, and read-write keys of

that transaction is added to the respective block-level tracking sets. If read-write sets of the transaction intersect with exactly one block-level tracking set, then the transaction gets added to that block and its block-level key set gets updated with the read-write set of the transaction. If read-write sets of the transaction intersect with two or more block-level tracking sets, then a new slot gets created and the transaction gets added to that slot. This method adds very little overhead to the orderer and is applicable even when the transaction rate is very high. However, this method does not provide the most optimized dependency tracking. Note that the orderer can track two sets at each block (read keys and write keys) for a finer-grained dependency tracking.

2) DAG based method: In this method, the orderer maintains a Directed Acyclic Graph (DAG) of the transactions belonging to the current slot. Each transaction is represented by a node in this DAG, with an edge going from the node representing Transaction 1 to the node representing Transaction 2 if the latter is dependent on the former. At the beginning of the slot, the DAG is empty. The orderer then steps linearly through the incoming transactions. For each transaction, it adds a node to the DAG and creates an edge to all the nodes on which it depends. Each connected component in such a DAG represents a set of dependent transactions. These transactions should be placed in the same block if they are to be part of the same slot. Transactions belonging to different connected components in the DAG represent independent transactions. These can be put in the same slot by splitting them into various blocks. This method can optimize the number of transactions that can be put in the same slot. But this method is computationally expensive compared to the *Set based method*. Hence it is useful when the incoming transaction rate is low or moderately high.

Please note that the orderer can choose the dependency tracking method for any slot independently. It can start with the DAG-based method to optimize the number of transactions that can be put in the same slot. If the transaction rate increases and the DAG method starts becoming a bottleneck, the orderer can switch to *Set based method* (ref IV-A1) for future slots. When the transaction rate comes down later, the orderer can switch back to the *DAG based method* to prioritize optimization.

B. Condition for new slot creation

It is the responsibility of the orderer to group blocks into slots. An orderer creates a new slot due to one of the following reasons.

1) Dependent Transaction: When the orderer finds that the current transaction cannot be included in any block of the current slot (ref Sec IV-A1), it creates a new slot and puts the current transaction in the first block of the new slot.
An alternative for the orderer is to defer including the current transaction in the blockchain. This will enable it to include some more transactions in the current slot. The orderer also ensures that the deferred transaction gets eventually added by creating a new slot before some threshold timeout.

2) Max slot size: The orderer has to keep track of the read-write set of all the transactions belonging to the current slot. The memory overhead of the orderer increases linearly with the number of independent transactions in a slot. To limit the memory overhead even when the number of independent transactions grows very big, the orderer can create a new slot once the number of blocks in any slot reaches a predefined number.

C. Slot Identification

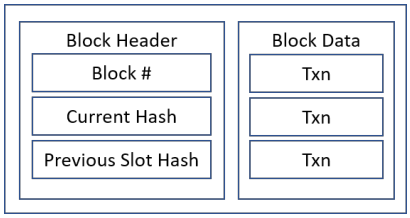


Fig. 3: Block Structure

Fig.3 depicts the modified block structure with slot information in Hyperledger Fabric. Each block consists of *two* parts, Header and Data. We can optionally add a "Slot #" field in the Header. We rename the field "Block Hash" to "Previous Slot Hash" in the block header. The value for this field is obtained by calculating the hash of a string obtained by concatenating the hash of each block header in the previous slot. For example, in Fig.2

$$Previous\_Slot\_Hash(B17) = Hash\,(\,Hash\,(\,Header\_B14\,)\,||\,Hash\,(\,Header\_B15\,)\,||\,Hash\,(\,Header\_B16\,)\,)$$

All the blocks belonging to the same slot have same value for "Previous Slot Hash". For example, in Fig.2

$$Previous\_Slot\_Hash(B17) = Previous\_Slot\_Hash(B18) = Previous\_Slot\_Hash(B19)$$

This enables a peer node to start validating blocks in a current slot even if other blocks in the same slot have not yet arrived.

D. Peer Node Workflow

Once the orderer adds the first block in the current slot, adding more blocks into the predecessor slot is not possible since the "Previous Slot Hash" in the block that got added to the current block should get populated. Moreover, the Fabric channel supports atomic delivery of all messages on the channel to all peers in the same logical order. Hence, a peer carries out the below steps on receiving a broadcast block.

1) If the "Previous Slot Hash" of a new block is equal to the "Previous Slot Hash" of other blocks in the current slot, then the new block belongs to the current slot. In this case, the peer can go ahead and validate the transactions belonging to the new block. E.g., referring to Fig.2, a peer node on receiving the block B15 carries out the block validation concurrently with the validation of the block B14 that belongs to the same slot.

2) If "Previous Slot Hash" of a new block is not equal to the "Previous Slot Hash" of other blocks in the current slot, then the new block should ideally belong to the next slot. Hence the validation of the transactions belonging to the new block needs to wait till the validation of all the blocks in the current slot gets completed. E.g., referring to Fig.2, a peer node having all blocks till B16 and receiving the block B17 waits for the validation of block B17 till the validation of blocks B14, B15, and B16 get completed.

Above scenarios can be extended appropriately to process the blocks that get delivered out of order by the network peers.

Algorithm 1 Compute Slot of Given Block

```
Require: NewBlock : Newly Created Block
  Maintain following across invocations:
  List<Block> CurSlotBlk
  Set<Key> CurSlotKeys
  Byte[] prevSlotHash

  Set<Key> CurBlkKeys = Keys accessed in NewBlock
  Boolean flag = False
  for all Key curKey in CurBlkKeys do
    if CurSlotKeys.contains(curKey) then
      flag = True
    end if
  end for
  if CurSlotBlk.size == MaxBlkPerSlot then
    flag = True
  end if
  if flag then
    prevSlotHash = Compute new slot hash from CurSlotBlk
    CurSlotBlk.clear()
    CurSlotKeys.clear()
  end if
  NewBlock.prevSlotHash = prevSlotHash
  CurSlotBlk.add(NewBlock)
  CurSlotKeys.add(CurBlkKeys)
  return NewBlock
```

E. Co-existence with pluggable consensus methods

Our approach to scale the performance of validation phase can co-exist with various types of pluggable consensus mechanisms (e.g., crash fault tolerant ordering services like Raft [16] and Apache Kakfa [17], or byzantine fault tolerant ordering services like PBFT [18] and Mir-BFT [2], etc.) available in Hyperledger Fabirc, without changes to the consensus mechanism as such. However, it requires a wrapper on top of any such consensus mechanism. The wrapper consists of a pre-processing phase and a post-processing phase to the execution of the consensus mechanism. The pre-processing phase re-arranges the transactions such that the modified sequence of the transactions is better amenable for the methods proposed in this paper. The pre-processing phase typically implements the methods like *Set based method* (refer to the section IV-A1) or *DAG based method* (refer to the section IV-A2) or any other such similar method which primarily tries to select the transactions from the incoming transactions for the block creation. The post-processing phase implements the slot creation methods discussed in section IV-B.

Algorithm 1 presents the pseudo-code for a sample implementation of Set based post-processing phase at the orderer. Following variables track current slot information:- (a) *CurSlotBlk* contains blocks belonging to current slot. (b) *CurSlotKeys* contain all keys that have been accessed(read or written) by transactions belonging to blocks in *CurSlotBlk*. (c) *PrevSlotHash* contains hash of blocks in previous slot. A new slot is created when a key accessed in the *NewBlock* clashes with a key in *CurSlotKeys* or when number of blocks in *CurSlotBlk* reach a predefined block limit in a slot (*MaxBlkPerSlot*).

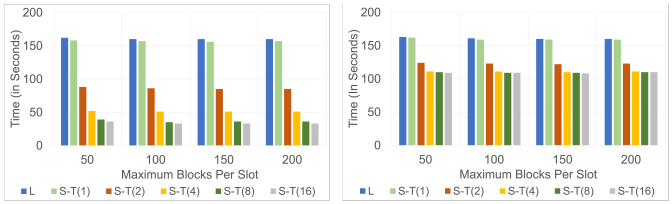
V. Evaluation

This section describes the preliminary experiments done to evaluate the benefits of grouping blocks into slots. The experiments were carried by writing a simulator for major modules impacted by the proposed slot-based approach. Though this simulation does not model a complete blockchain network, it gives a glimpse of how the slot-based approach compares against the current linear model (i.e., blockchain with total order among transactions), by comparing the performance of different independent modules in the slot-based approach with the linear model. The simulation allows running multiple experiments differing in exactly one parameter and hence find out its effect on the system. *Set based method* (Section IV-A1) is used to track transaction dependencies in slot generation.

The number of blocks in a slot are dependent on the keys being accessed by the transactions. In this evaluation we will consider two scenarios :- (a) Uniform Key Access, where the keys are accessed uniformly by the transactions. (b) Normal Key Access, where a subset of keys (20%) are accessed most of the times (80%). All experiments use the following configuration, unless specified otherwise, (a) $1M$ distinct keys (b) Maximum number of transactions per block $= 100$ (c) Maximum number of blocks per slot $= 100$.

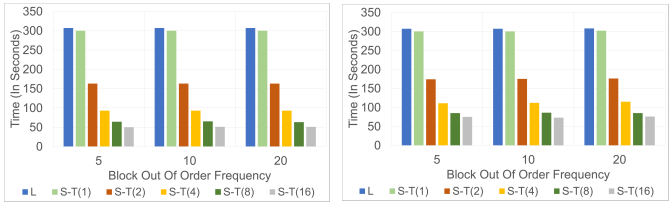
The figures use the following naming convention. The first letter tells the approach used (L for Linear model and S for Slot-based approach), Block out-of-order frequency is specified by F and Maximum number of parallel threads is specified by T.

Fig. 4 shows the time it takes to validate the transactions in a blockchain when all the blocks are already available at the peer node. This can happen when the peer has restarted and wants to recreate the world state from the blocks which it already has. In both Uniform Key Access (Fig. 4a) and Normal Key Access (Fig. 4b), Slot-based approach outperforms the Linear model. It also shows that having more threads/CPUs allows Slot-based approach to create the world state faster. The performance gain is more for Uniform Key Access since it allows more blocks to be placed in the same slot compared to Normal Key Access, which leads to efficient utilization of threads (Table I).



(a) Uniform Key Access (b) Normal Key Access

Fig. 4: Offline Blockchain Validation Time



(a) Uniform Key Access (b) Normal Key Access

Fig. 5: Online Blockchain Validation Time

	1M keys	5M keys	10M keys
Uniform Key Access	19	23	28
Normal Key Access	5	7	10

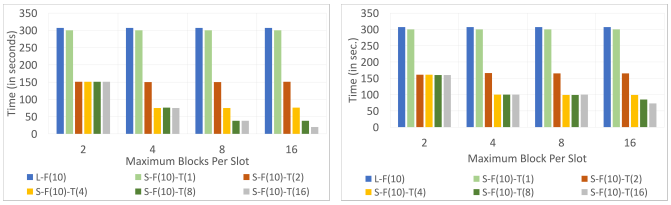
TABLE I: Average number of Blocks Per Slot

The biggest benefit of using slots is realized when blocks arrive out of order but can still be processed since they belong to the current slot. To model this behavior, a parameter *block out-of-order frequency* is introduced into the simulation, which controls the rate at which blocks arrive out of order.

Fig. 5 shows the time it takes to validate the blocks when they arrive periodically with some blocks coming out of order. This experiment simulates a real world scenario where a peer node is receiving new blocks and updating its world state. Slot based approach outperforms the linear model for both Uniform Key Access (Fig. 5a) and Normal Key Access (Fig. 5b). Having more number of threads allows slot based approach to process the blocks faster. The slot based approach is able to maintain its performance even when 20% of blocks arrive out of order. This is due to the fact that as long as the arriving block belongs to the current slot, even though it is out of order, it can be processed immediately.

Fig. 6 shows the time taken to validate the blocks when they arrive periodically (some blocks coming out of order) and the parameter *maximum blocks per slot* is kept very low. It shows that the effect of slots starts to kick in even from a very low value of *maximum blocks per slot*. Slot based approach continues to give better performance as long as *maximum blocks per slot* is more than the number of threads available with the peer. If *maximum blocks per slot* is less than the number of threads available with the peer, then the extra threads remain unused as there are not enough blocks in slot.

This added performance at the peer node comes at a cost as the orderer has to do more work to keep track of the slots. Table II shows the overhead on orderer and benefit at peer. The



(a) Uniform Key Access (b) Normal Key Access

Fig. 6: Online Blockchain Validation Time

orderer incurs less than 20% overhead for slot-based approach compared to the linear model. Moreover, the gain at each peer node is between 45%-85% for various configurations with threads varying from 2 to 16, and can grow with more threads on the peer nodes. An important point to note here is that the overhead is only for orderers whereas the benefits of the slot are for all the peer nodes. Since the number of orderers is extremely low compared to the number of peer nodes, the overhead is amortized leading to a huge overall benefit.

	Uniform Key Access		Normal Key Access	
	Orderer Overhead	Peer Benefit	Orderer Overhead	Peer Benefit
S(50)	7%	50% - 85%	10%	45% - 70%
S(100)	6%	50% - 85%	10%	45% - 70%
S(150)	10%	50% - 85%	11%	45% - 70%
S(200)	16%	50% - 85%	19%	45% - 70%

TABLE II: Orderer Overhead vs. Peer Node Gain using Slots

A direct benefit of Slot-based approach is increased transaction throughput as can be seen from Fig. 7 and 8. Specifically, Fig. 7a and 8a show that Slot-based approach has better transaction throughput than linear model. Fig. 7b and 8b show that Slot-based approach is able to achieve better throughput by increasing the number of threads. And Fig. 7c and 8c show that Slot-based approach is able to maintain its performance even when *block out-of-order frequency* increases.

Using slots lead to reduction in the *average block validation time* as can be seen from Fig. 9 and 10. Specifically, Fig. 9a and 10a show that slot based approach has much lower block validation latency compared to linear model. Fig. 9b and 10b show that slot is able to reduce latency further by increasing the number of threads. Fig. 9c and 10c show that slot is able to maintain its performance even when *block out-of-order frequency* increases.

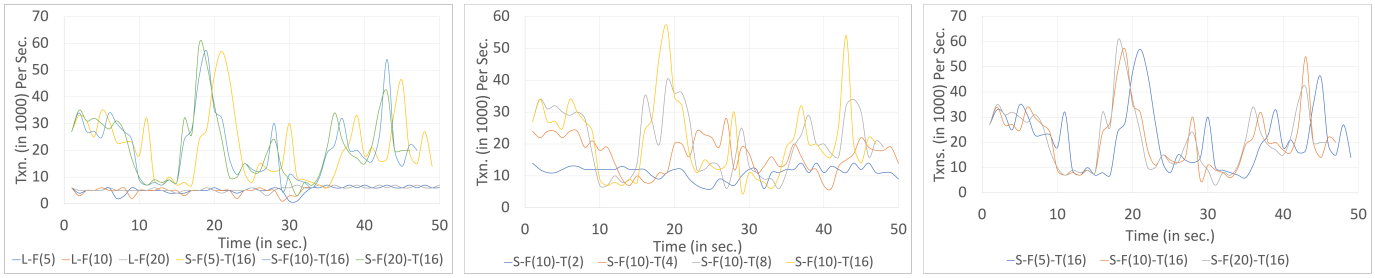
The experimental results clearly show that the slot based approach can increase the performance of blockchain network.

VI. Use Case Scenarios

We describe some use cases of our proposed *blockchain system with support for partial order among transactions* here.

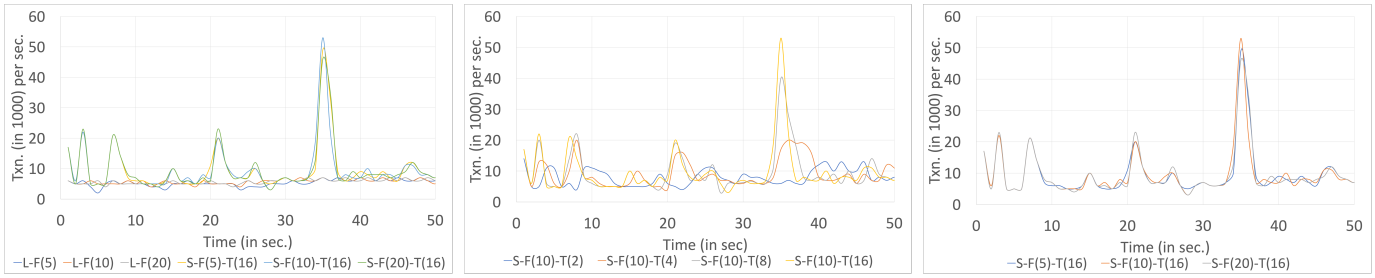
A. Rapid recovery of crashed peers

Currently, when a peer node reboots, all blocks of the Fabric blockchain ledger get validated sequentially. This process leads to long boot times.



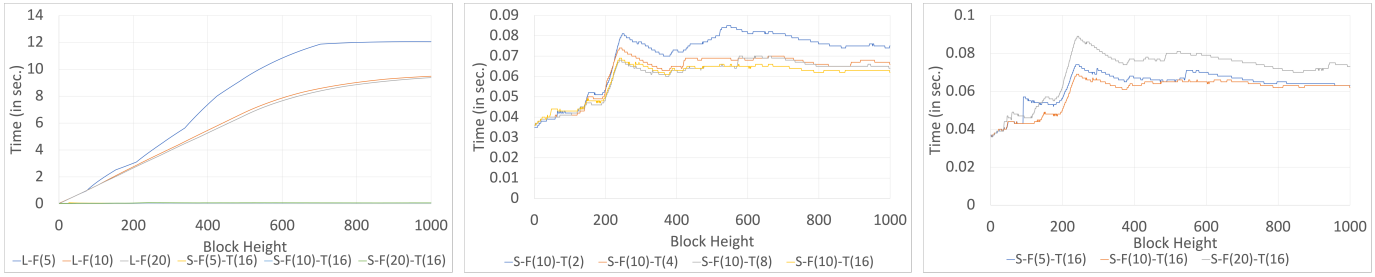
(a) Slot-based approach vs Linear Model (b) Varying number of threads (c) Varying Block Out-of-Order frequency

Fig. 7: Transaction throughput for Uniform Key Access



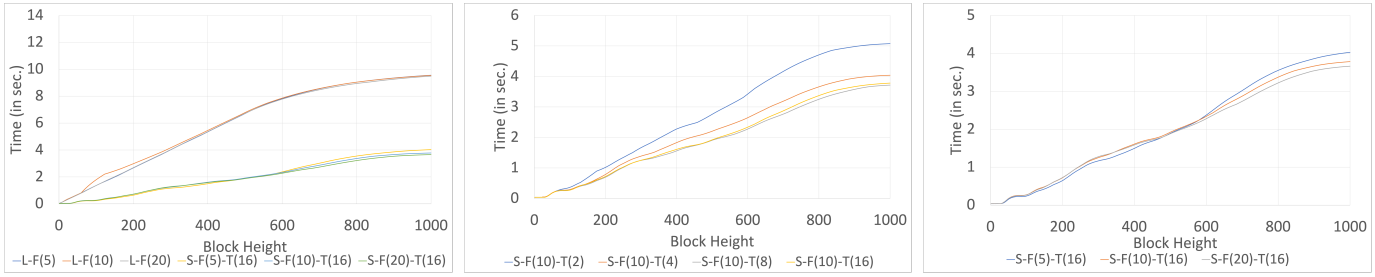
(a) Slot-based approach vs Linear Model (b) Varying number of threads (c) Varying Block Out-of-Order frequency

Fig. 8: Transaction throughput for Normal Key Access



(a) Slot-based approach vs Linear Model (b) Varying number of threads (c) Varying Block Out-of-Order frequency

Fig. 9: Average Block Validation Time for Uniform Key Access



(a) Slot-based approach vs Linear Model (b) Varying number of threads (c) Varying Block Out-of-Order frequency

Fig. 10: Average Block Validation Time for Normal Key Access

In contrast, our solution allows the peer node to validate multiple blocks belonging to the same slot in parallel. This process significantly reduces the time needed to complete validation of the entire blockchain ledger and results into faster recovery of the crashed peer.

B. Improved Query Response Time

Currently, a peer node (including the anchor peer) of the Fabric is required to validate all the previous blocks sequentially before they can respond to the query on the keys which are part of the current block. This leads to a longer query response time being observed by the client applications, even if the current block has no dependency on previous blocks.

In contrast, our solution allows a peer node to validate the current block as soon as it gets received if it is part of the current slot. It also ensures that the update to the keys belonging to the current block is not dependent on any other pending validation and that the peer node can respond to query on keys which are part of the current block as soon as its validation is completed. This significantly reduces the query response time observed by the client application.

C. DAG based Parallelism in Transaction Validation

Our solution allows a peer node to carry out the DAG analysis on the transactions belonging to all blocks in a slot, and identify independent sets of transactions that can be validated in parallel. This improves the resource utilization at each peer node. Without our system, its only possible to carry out the DAG analysis at individual block level only.

This allows massive parallelism at peer nodes but it can lead to an increase in transaction latency since the peer nodes buffer all the blocks of a slot before validation begins. To avoid this overhead in transaction latency, we can apply DAG analysis at block level only on Hyperledger Fabric anchor peers hence the transactions get added to the ledger immediately.

D. Improved new node operational time

In a permissioned blockchain network like Hyperledger Fabric, each peer node is usually associated with a limited set of client applications. Each client application is interested in only a subset of keys stored on the blockchain. All transactions belonging to the application query and modify only these subset of keys. When a new peer node is brought up it has to get all the blocks of the blockchain and validate all the transactions before it can respond to queries on keys of interest which leads to significant delay.

An extension to our solution allows a new node to become operational significantly faster. For this, the orderer has to keep track of the last transaction (i.e. block number and transaction id) which modified a particular key and dependency graph across all transactions in the blockchain. The former can be maintained as a database in the orderer. For the latter, the orderer can use an extended version of DAG based dependency tracking. When a new peer is brought up, it asks the orderer for the last transaction which modified the keys of interest. Then it queries the orderer to get the previous transactions on which these transactions depend. The peer node requests for only a subset of blocks and validates the transactions of interest. At the end of this validation, it can start responding to queries on the keys of interest.

VII. Conclusion

In this paper, we presented a permissioned blockchain system to support partial ordering among non-dependent transactions. It enables a peer node to validate multiple blocks in parallel, allowing efficient resource utilization, while ensuring that all the network peers reach the same final state. Our approach to optimizing the validate phase can co-exist with various types of pluggable consensus mechanisms available in Hyperledger Fabric and can be implemented without changes to the consensus mechanism as such. The benefits of our approach grow as the blockchain network size grows.

References

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. 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. Vukolic, S. W. Cocco, and J. Yellick, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *EuroSys Conference, 2018*.

[2] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukoli, "Mir-bft: High-throughput robust bft for decentralized networks," in *arXiv, 2021*.

[3] G. Yu, X. Wang, K. Yu, W. Ni, J. A. Zhang, and R. P. Liu, "Survey: Sharding in blockchains," *IEEE Access, 2020*.

[4] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," in *ACM Conference on Advances in Financial Technologies, AFT, 2019*.

[5] D. E. Dillenberger and G. Su, "PARALLEL EXECUTION OF BLOCKCHAIN TRANSACTIONS," U.S. Patent 10 255 108.

[6] S. Popov, "The tangle," https://iota.org/IOTA_Whitepaper.pdf, 2016.

[7] Y. Lewenberg, Y. Sompolinsky, and A. Zohar, "Inclusive block chain protocols," in *Financial Cryptography and Data Security - 19th International Conference, FC 2015*.

[8] H. Meir, A. Barger, and Y. Manevich, "Increasing concurrency in hyperledger fabric," in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[9] T. Nakaike, Q. Zhang, Y. Ueda, T. Inagaki, and M. Ohara, "Hyperledger fabric performance characterization and optimization using goleveldb benchmark," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2020.

[10] P. Thakkar, S. Nathan, and B. Viswanathan, "Performance benchmarking and optimizing hyperledger fabric blockchain platform," in *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2018.*

[11] H. Javaid, C. Hu, and G. Brebner, "Optimizing validation phase of hyperledger fabric," in *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2019*.

[12] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "Fastfabric: Scaling hyperledger fabric to 20,000 transactions per second," in *IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2019.*

[13] P. Thakkar and S. Nathan, "Scaling hyperledger fabric using pipelined execution and sparse peers," in *arXiv, 2020*.

[14] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the International Conference on Management of Data, 2019*.

[15] H. Gupta and D. Janakiram, "Colosseum: A scalable permissioned blockchain over structured network," in *Proceedings of the Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts, 2019*.

[16] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference, 2014*.

[17] "*Apache Kafka*," https://kafka.apache.org/intro.

[18] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems, 2002*.