Scaling Blockchains Using Pipelined Execution and Sparse Peers

Parth Thakkar*
University of Illinois, Urbana-Champaign
thakkar.parth.d@gmail.com

Senthilnathan Natarajan†
IBM Research, India
senthil.nathan.7@gmail.com

ABSTRACT

Large cloud providers such as AWS and IBM now provide managed blockchain platforms, showcasing an active interest in blockchains. Unfortunately, blockchains provide poor performance and scalability. This is true even for the `Execute-Order-Validate` (`EOV`) style of blockchains which improves over the traditional `Order-Execute` architecture. We experimentally show that `EOV` platforms scale poorly using both vertical and horizontal scaling approaches. We find that the throughput is bottlenecked by the Validation and Commit phases, which poorly utilize the resources, limiting performance and scalability.

We introduce three ideas to improve the performance, scalability, and cost-efficiency of `EOV` platforms. We first propose a *provably correct* way to pipeline the Validation and Commit phases. We then introduce a new type of node called *sparse peer* that validates and commits only a subset of transactions. Finally, we propose a technique that makes it possible for these systems to elastically scale as per the load. Our implementation provides 3.7× and 2.1× higher throughput than Hyperledger Fabric and FastFabric for the same infrastructure cost while providing their peak throughputs at 87% and 50% lower cost. It also makes dynamic horizontal scaling practical by providing 12-26× faster scale-up times.

ACM Reference Format:

Parth Thakkar and Senthilnathan Natarajan. 2021. Scaling Blockchains Using Pipelined Execution and Sparse Peers. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–5, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3472883.3486975

*Work done at IBM Research, India

†Committer, Hyperledger Fabric

SoCC '21, November 1–5, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11...$15.00
https://doi.org/10.1145/3472883.3486975

1 INTRODUCTION

Blockchains are becoming popular as data management systems in the enterprise setting. They allow mutually distrusting organizations to share a common data platform in a tamper-proof and auditable manner. Every organization controls its servers, each of which stores a copy of the shared data. Blockchain protocols ensure that this data is modified in a consistent and agreed-upon manner. Unfortunately, blockchains suffer from poor performance, scalability, and cost-efficiency due to serial transaction processing and excessive redundancy.

A blockchain allows modifying the shared data by executing transactions against it. In a typical blockchain, the servers/nodes first agree upon an order of such transactions using a consensus protocol. Then every node follows that order to *serially* execute and commit transactions on its copy of data. Such blockchains are called `Order-Execute` (OE) blockchains as transactions are first ordered and then executed. The serial execution of transactions is important here, as every node should have the same state after executing n transactions.

Hyperledger Fabric [6] introduced a novel architecture called `Execute-Order-Validate` (`EOV`) to overcome many limitations of the `OE` architecture. Here, a transaction is first optimistically executed (but not committed) by a subset of nodes, say one from every organization. The transaction is then ordered among other such pre-executed transactions. Finally, every node validates and commits these transactions in the same order. This design allows parallel execution of transactions. As parallel execution may lead to conflicts between two transactions, the validation stage uses the transaction order to abort the later transaction. This ensures every node is at the same state after executing n transactions.

`EOV` is particularly well-suited to the enterprise setting. A typical enterprise setting has no more than tens of distrusting parties, thus reaching a consensus on the transaction order is not expensive. In this setting, the throughput is bottlenecked by transaction *processing*. As a result, EOV's ability to execute transactions in parallel gives it an edge over the traditional OE model. Unlike OE, an EOV node can execute more transactions if we provide it more CPU cores. The same fact also implies horizontal scalability — an organization can add more nodes to improve its transaction execution capacity, as they all can execute in parallel. This is not possible in OE, as transactions must be serially executed, and adding more nodes would only lead to wasted resources.

Overall, EOV is a step in the right direction for enterprise blockchain platforms. However, as we show in this paper, it misses several opportunities to improve the performance, scalability and cost-efficiency of blockchain networks. We identify 3 broad problems with this approach, and then propose techniques to address them.

First, we show that the validation and commit phases bottleneck the throughput. These phases largely use different hardware resources and yet execute serially, leading to poor resource utilization. The validation phase is compute-heavy as it performs multiple signature verifications, while the commit phase is IO-heavy as it performs synchronous disk IO. They execute serially because validating block $N+1$ may require reading states committed in block N, but reading them while block N is being committed may lead to incomplete or stale reads. To address this, we introduce a provably correct mechanism to pipeline these phases. We show that this doubles the CPU utilization and provides 40% higher throughput.

Second, we show that even though EOV networks *can* scale horizontally, they only do so poorly. This is because adding more nodes to an organization only improves its transaction *execution* capacity. However, every added node duplicates the expensive validation and commit phases even though they belong to the same trust domain (i.e., the organization). As these phases largely dictate the throughput, more nodes do not translate to *large* throughput improvement. To address this, we introduce a new type of node called *sparse peer*, which selectively commits transactions. Thus, nodes within an organization can share validation and commit workload, improving resource utilization and horizontal scalability. We show that this improves throughput by 2.7×.

Finally, we show that newly added nodes take a long time to synchronize states, as they validate and commit every transaction. This requires the networks to be provisioned upfront to handle transient *peak* load, implying a higher infrastructure cost than a system that can elastically scale with load. We address this by introducing a technique to quickly split a full peer into multiple *sparse peers*, and to merge multiple *sparse peers* into a full peer. This enables a network to elastically scale with workload,

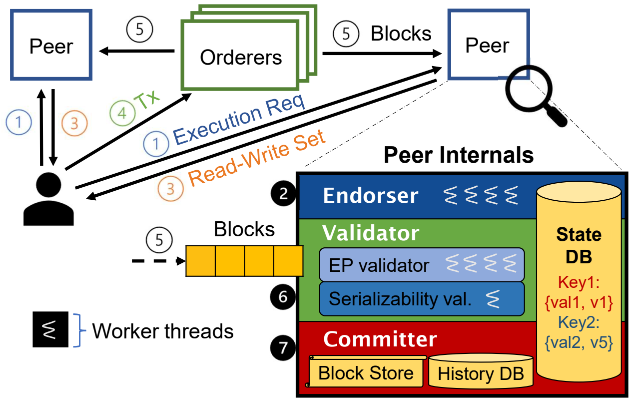

Figure 1: Transaction flow and peer internals

which can help reduce infrastructure cost. Our technique reduces the scale-up time of a network by 12-26×.

We implement our ideas by modifying Hyperledger Fabric [6], a popular EOV blockchain. We compare it with Fabric and FastFabric [10] using the Smallbank and YCSB workloads. Altogether, our approach provides 3.7× and 2.1× higher throughput than Fabric and FastFabric respectively, for the same infrastructure cost. It also provides their peak throughputs at 87% and 50% lower cost.

2 BACKGROUND

This section briefly presents an Execute-Order-Validate (EOV) blockchain architecture — Hyperledger Fabric [6]. Fabric consists of three entities: client, peer, and orderer. The transaction flow involves all three entities and comprises four phases — execution, ordering, validation, and commit. Figure 1 shows the transaction flow. We now describe the transaction flow by focusing on each phase.

Phase 1—Execution. In step ①, a client wanting to read or modify the blockchain data requests a peer to execute a specific *smart-contract*. A smart-contract is a program that can read and modify blockchain states. In step ❷, the peer runs the contract while recording its reads and writes but does not *actually* update any state. In step ③, the peer returns the **read-write set** (i.e., the states that the contract read or attempted to modify) along with the peer's signature/endorsement over it. The client can simultaneously request many peers to execute the contract, depending upon the contract's policy [6]. A contract can invoke other contracts, which enables modular application design.

Phase 2—Ordering. Once the client receives the responses in step ③ and their read-write sets match, it bundles these responses into a `transaction` and sends that to an orderer in step ④. Each orderer node receives several such transactions concurrently, and they all employ a consensus protocol such as Raft [13] to order the received transactions and group them into blocks. Each block has a sequence number called *block number*, previous block's hash, current block's hash, ordered transactions (i.e., *commit order*), and the orderer's signature. In step ⑤, the orderer nodes broadcast these blocks to peers in the same order.

Phase 3—Validation. In step ❻, each peer validates every received block. The peers pass every transaction in the block through two validators:
(1) `Endorsement Policy validator` marks a transaction valid only if it has been signed by enough organizations as specified in the contract's policy [6].
(2) `Serializability validator` marks a transaction valid only if the states read by it have not been modified by any preceding valid transaction. Fabric tracks state modifications by tagging every state with a *version*.

Phase 4—Commit. Finally, in step ❼, the *committer* first stores the block in the *block store*, which is a chain of blocks stored in a file system along with the transaction validity information. Second, the *committer* applies the write set of all valid transactions to the *state database*, which maintains all active states. Third, it stores metadata of updates of all valid transactions to the *history database*, which holds the version history (but not the values) of both active and inactive states.

3 MOTIVATION

In this section, we quantify the impact of vertical and horizontal scaling on Fabric's performance. We also show that validation and commit phases are the bottleneck.

Infrastructure Cost in Public Cloud. Throughout the paper, we consider infrastructure costs incurred for the achieved performance. We take the cost to be proportional to the total number of vCPUs allocated in the network because the VM price quotes provided by various cloud providers such as AWS, Azure, Google Cloud, and IBM Cloud [1–4] show the same trend.

Workload and Configuration. We ran our experiments on a cluster of 70 VMs, each with 32 GB RAM, 16 vCPUs of Intel Xeon E5-2683 v3 2.00GHz, SSD storage, and 1 Gbps network bandwidth. Every experiment used four organizations, a Raft [13] ordering service with five nodes, and a block size of 100 transactions. The number of peers depended upon the experiment. We measured the system's peak throughput as the primary performance metric. Table 1 shows the workloads used. Note that for both workloads, every transaction touched two states, chosen as per the mentioned distribution.

Base Case Performance. We use an optimized version of Fabric v1.4, which avoids redundant deserialization operations at various phases within a peer using a cache, as proposed in FastFabric [10].

Workload	Record size	Distribution	Write:Read
Smallbank [5]	10 bytes	Uniform	5:1
YCSB [7]	1 KB	Zipf (S = 0.5)	1:1

Table 1: Workloads used

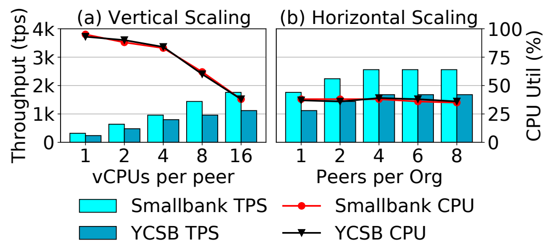

Figure 2: Vertical and horizontal scaling

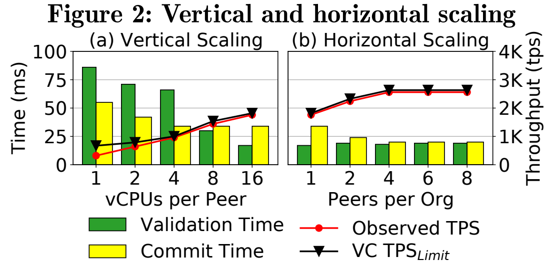

Figure 3: Validation and commit time

3.1 Scaling behaviour of EOV

Vertical Scaling. Figure 2(a) plots the peak throughput and CPU utilization for various allocated vCPUs. Even though the allocated vCPUs and hence the infrastructure cost increased 16×, the throughput under Smallbank increased only 5.5×. The disproportionate improvement in throughput is correlated with poor CPU utilization, which reduced from 95% to 36%. YCSB showed similar results. When the number of vCPUs was small, both endorser and validator contended for the CPU resources, which led to high CPU utilization. Adding more vCPUs reduced the contention leading to higher throughput. However, beyond a point, the CPU is underutilized. This is because, during the commit phase (IO heavy) execution, the validation phase (CPU heavy) does not execute, and vice-versa. The validator cannot validate block $(N+1)$ while block (N) is being committed because the state updates performed during the commit could make the validator read stale or incomplete state.

Takeaway 1. *CPU is underutilized due to the serial execution of validation and commit phases. By pipelining the two phases, we can improve performance, CPU utilization, and thus cost-efficiency.*

Horizontal Scaling. We now study how adding peers to every organization affects overall throughput. We generate load such that if there are N peers per organization and the load on the network is L tps, then each peer receives $\frac{L}{N}$ transactions for execution every second. Figure 2(b) shows an increase in throughput by adding more peers. Thus, unlike `OE` platforms, Fabric *can* scale

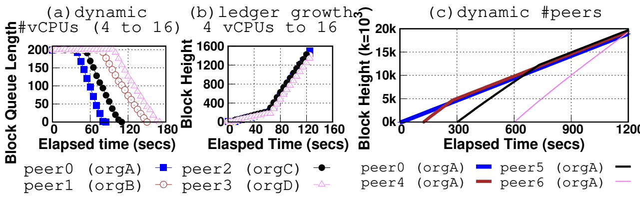

Figure 4: Impact of dynamic scaling by vCPUs & peers.

horizontally. However, the magnitude of scalability is small. Compared to 1 peer per organization, 4 peers give only 1.5× higher throughput (for both workloads) while the infrastructure cost increases 4×. Further, the throughput does not increase beyond 4 peers. This begs the question — why is horizontal scaling so ineffective?

Bottleneck: As the throughput increases by adding more peers, it is clear that the ordering service does not bottleneck the throughput. While peers within an organization share the transaction execution load, they individually validate and commit every block. Given that validation and commit phases are expensive because of multiple signature verifications and synchronous disk IO, we check if these two phases bottleneck the throughput. To do so, we calculate VC TPS_{Limit} as the throughput that validator and committer components can provide together. If a block containing T transactions is validated in V ms and then committed in C ms, then:

$$\text{VC TPS}_{Limit} = T \times \frac{1000\ ms}{V{+}C\ ms}$$

This equation holds because blocks get validated and committed serially. Figure 3 shows that for both vertical and horizontal scaling, the observed throughput equals the maximum throughput validator and committer components can provide. The only exceptions are peers with 1 and 2 vCPUs that are CPU-bottlenecked, having more than 90% CPU utilization, as shown in Figure 2.

Takeaway 2. *A Fabric network's throughput is bottlenecked by the validation and commit phases. Even so, every peer within an organization validates and commits each block. As peers in an organization can trust each other, they need not repeat these expensive operations. Eliminating this excessive redundant work within an organization can improve performance and reduce cost.*

3.2 Mitigating overloaded situation

Dynamic Vertical Scaling. To study the impact of dynamic vCPU scaling on the performance, we overloaded peers in a network by generating more load than it can handle at 4 vCPUs. We then increased the number of vCPUs to 16, one peer at a time, after the block queue's length reached 200. Figure 4(a) plots the time taken to reduce the block queue length from 200 to 0 for each peer across organizations. Though peers scaled immediately, it took around 50 *secs* to 70 *secs* to reduce the queue length to 0. Figure 4(b) plots the ledger block height over time, while the number of vCPUs increased from 4 to 16. The block height is nothing but the last committed block number. As expected, the ledger commit rate increased.

Takeaway 3. *Dynamic scaling by vCPU is efficient as it can quickly react to the increased load. However, this approach is limited by the number of available vCPUs.*

Dynamic Horizontal Scaling. To study the time taken to add a new peer in an existing organization, we ran a peer per organization and then added a new peer at the 2^{nd} minute, 5^{th} minute, and 10^{th} minute. Figure 4(c) plots the time taken by new peers to sync up with the existing peers. We observed that the time taken was proportional to existing peers' block height as the new peer had to fetch all old blocks, validate, and commit them one by one. As there were no endorsement requests on the new peer during the sync up, it could catch up by committing transactions at a rate of 3300 tps (while with endorsement, it was only 1760 tps).

Takeaway 4. *As the new peer validated and committed all blocks, i.e., redundant work, it took a significant amount of time to sync up with existing peers. Instead, we can make peers trust each other within an organization to avoid this redundant work.*

4 PIPELINING VALIDATION & COMMIT

We know that the validation and commit phases together dictate the throughput (takeaway 2), and these two phases execute serially & underutilize the CPU (takeaway 1). Therefore, in this section, we focus on pipelining the two phases.

4.1 As-Is Validation and Commit

Let's take an example to understand validation & commit phases. Suppose two blocks of transactions, as shown in Table 2, are received by a peer. The table shows states read by each transaction under **R-Set** and states written under **W-Set**. Let's assume that each peer's database stores states $\{k_1, k_2, k_7\}$, each at version v_1. On receiving

Block	Txn	R-Set	W-Set	Valid?	Updates
B_1	T_1	(k_1, v_1)	k_2	✓	$(k_2, v_1 \to v_2)$
	T_2	(k_2, v_1)	k_1	✗	-
	T_3	-	k_1	✓	$(k_1, v_1 \to v_2)$
	T_4	-	k_3	✓	$(k_3, \phi \to v_1)$
B_2	T_5	range(k_3, k_5)	k_6	✗	-
	T_6	(k_7, v_1)	k_7	✓	$(k_7, v_1 \to v_2)$

Table 2: Example transactions for validation.

a block, a peer first checks if enough organizations signed each transaction. For our example, we assume this to be true. Then, the peer invalidates any transaction that has

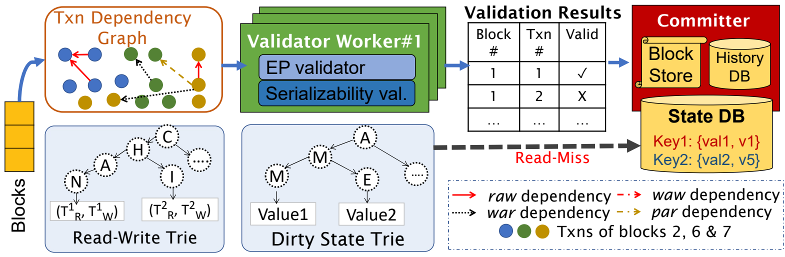

Figure 5: Components involved in pipelined validation and commit

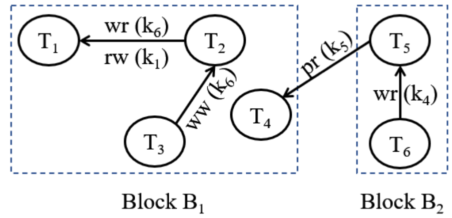

Figure 6: Example dependency graph

read stale data while respecting the transactions commit order. Accordingly, the peer finds T_1 valid as it has read state k_1 at its latest version. T_1's write thus changes k_2's version to v_2. The peer then invalidates transaction T_2 as it has read k_2 at the older version v_1. As T_3 and T_4 do not read any states, they are considered valid. As block B_1 is validated, the peer commits the block and applies the valid transactions' updates to its database.

Next, the peer starts validating block B_2. The first transaction T_5 has performed a *range-query* on the state. This means, during T_5's execution, the smart-contract requested to read all existing keys in the range k_3, k_5. The peer(s) executing T_5 did not have any keys in that range, so T_5's R-Set does not contain any keys. The R-Set only specifies that a range query was performed. However, after committing B_1, the peer's database contains k_3 that belongs to the range. Thus, T_5 has performed a *phantom read* and is marked invalid. After validating T_5 and T_6, the peer commits block B_2.

4.2 Proposed Validation and Commit

There are two limitations to the existing process. First, each transaction is validated *sequentially* because the validity of a prior transaction could affect the validity of a latter transaction, e.g., T_1 being valid made T_2 invalid. However, not all transactions need to wait for prior transactions' validation results. E.g., T_4's validity does not depend on any prior transactions. Second, block B_2's validation could not be started before block B_1 is committed because to validate T_5 correctly, T_4's updates must have been persisted in the database.

We eliminate both limitations to improve CPU utilization by (1) tracking fine-grained dependencies between transactions and validating independent transactions in parallel and (2) maintaining updates of valid transactions in-memory until the block containing those transactions is committed. Figure 5 shows the components involved that help with these. However, before we describe the modified validation procedure in detail, we highlight the correctness requirements. To correctly validate a transaction, we need the *latest* versions of *every* state it reads. We can break this requirement into two parts:

Correctness requirement 1: To validate a transaction T_j that reads a state k, we must first validate all transactions T_i (where $i < j$) that wrote to k.

Correctness requirement 2: Reads and writes to every state are applied in the correct order. E.g., if T_i and T_j (where $i < j$) both write to a state k, then any later transaction that reads k should only see T_j's write.

We now describe our approach that matches these requirements by construction.

4.2.1 Transaction dependency graph.

To satisfy these requirements, we track the following dependencies between transactions T_i and T_j where $i < j$:

(1) ***read-after-write*** ($T_i \xleftarrow{rw(k)} T_j$): T_i writes a new value to state k, and T_j reads it at an older version;
(2) ***write-after-read*** ($T_i \xleftarrow{wr(k)} T_j$): T_j writes a new value to state k, and T_i reads it at an older version;
(3) ***write-after-write*** ($T_i \xleftarrow{ww(k)} T_j$): Both T_i and T_j write to the same state k;
(4) ***phantom-read*** ($T_i \xleftarrow{pr(k)} T_j$): T_j performs a range query, and T_i creates/writes a state k in that range;

Apart from these, we also track dependencies where one transaction modifies the endorsement policy of the contract invoked by another transaction. These are just variants of the rw, wr, and ww dependencies. Note that an edge can only be from a newer transaction to an older transaction, i.e., if $T_i \leftarrow T_j$ then $i < j$, because the commit order is pre-decided. Thus, there are no cycles in the dependency graph. The dependency graph of transactions in Table 2 is shown in Figure 6.

Fate dependencies: While all dependencies are necessary to choose transactions for parallel validation, the rw, pr, and *endorsement policy* rw dependencies also decide the validity of dependent transactions. Hence, these are called *fate dependencies*. When there is a fate dependency from T_j to T_i, and if T_i turns out to be valid, then T_j must be invalid. For example, in Figure 6, $T_1 \xleftarrow{rw(k_1)} T_2$ is a fate-dependency as T_1's validity implies T_2's invalidity but $T_2 \xleftarrow{ww(k_1)} T_3$ is not a fate dependency as both can be valid or invalid independently.

Row	Workers			Dirty State	State DB	Results		Dependency Graph
	w_1	w_2	w_3			Valid	Invalid	
1	T_1	T_4	-	-	(k_1, v_1), (k_4, v_1), (k_6, v_1)	-	-	Same as above
2	T_3	T_6	-	(k_1, v_2), (k_5, v_1)	(k_1, v_1), (k_4, v_1), (k_6, v_1)	T_1, T_4	T_2, T_5	Only T_3 & T_6, no edges
3	-	-	-	(k_1, v_2), (k_5, v_1) (k_6, v_2), (k_4, v_2)	(k_1, v_1), (k_4, v_1), (k_6, v_1)	T_1, T_4 T_3, T_6	T_2, T_5	Empty
Committer commits the block B_1								
4	-	-	-	(k_4, v_2)	(k_1, v_2), (k_4, v_1) (k_5, v_1), (k_6, v_2)	T_6	T_5	-
Committer commits the block B_2								
5	-	-	-	-	(k_1, v_2), (k_4, v_2) (k_5, v_1), (k_6, v_2)	-	-	-

Table 3: Validation process of transactions

Operations exposed to other components: The extractor exposes the following two operations to enable other components to access the dependency graph:

(1) `GetNextTransaction()` $\rightarrow$ T_i: Returns the oldest transaction T_i in the graph that does not depend on any other transactions.

(2) `RemoveFromGraph(`T_i`, isValid)`: Removes T_i from the dependency graph. If `isValid` is *true*, it invalidates all transactions depending on T_i via a *fate dependency* and removes them from the graph.

Our design always ensures the following invariant:

Invariant I1: A transaction is removed from graph only when it has been fully validated.

4.2.2 Validators. The logic of `endorsement policy validator` and `serializability validator` remains the same as discussed in §2 except for the following:

(1) Validators can validate later blocks without waiting for the committer to commit earlier blocks.

(2) Validators read from dirty state buffer before resorting to state DB to avoid reading stale states.

(3) Multiple workers to validate transactions in parallel.

Each free worker calls `GetNextTransaction()` to get the next transaction to be processed. First, the `endorsement policy validator` checks whether the policy is satisfied. On success, the same worker executes the *serializability* check using OCC [11]. If the transaction passes both validations, the worker first applies the write-set to the dirty state and then calls `RemoveFromGraph()` to update the dependency graph. Finally, the worker stores the validation result in a table that the *committer* consumes. To ensure that the validator does not read a stale state when it validates block j before committing block i (where $j > i$), a read request (such as a read of the endorsement policy or the version of a state) would first go to the dirty state. Only on a miss, the read would reach the state DB. We store the dirty state as a trie, with the state's key stored along a path and its value & version stored in a leaf. This trie structure enables the *serializability validator* to validate range queries present in any transaction for a phantom read. Every range query is executed on both the trie and state DB.

4.2.3 Committer. The logic of the commit phase stays the same as discussed in §2, except that now it runs concurrently with the validators. Whenever the committer becomes free, it reads a block from the queue and retrieves the list of transactions. As shown in Figure 5, it then fetches their validation results from the results table. If a validation result is not available, the committer stays blocked until it is available. Once the committer collects the validation results of all transactions, it stores the block in the block-store and applies the valid write-sets to state & history DB as in vanilla Fabric. Finally, it calls the validation manager to remove the dirty state associated with the just committed block as the validator can read those states from the state DB itself.

4.3 Illustration

Let us continue the example from Table 2 and see how this new approach validates transactions in parallel. Say we have three workers, w_1, w_2, and w_3, to validate transactions, as shown in Table 3. Workers w_1 and w_2 would get transactions T_1 and T_4, respectively, by calling `GetNextTransaction()`. As every other transaction in the dependency graph has an out-edge, w_3 would stay idle. The initial database state is shown in the first row.

Each worker first checks if its transaction satisfies the endorsement policy. As assumed before, this check always succeeds. Then each worker checks if its transaction is reading any stale data. As both T_1 and T_4 read the latest version of states, they are marked valid. Each worker then applies its transaction's write-set to the dirty state (see the second row in Table 3). Finally, the workers call `RemoveFromGraph()` to update the dependency graph.

As transaction T_2 has a fate dependency on T_1, it would be marked invalid because T_1 is valid. Similarly, since transaction T_5 has a fate dependency on T_4, T_5 would be invalidated as well. Thus, the dependency graph would have only two transactions (T_3 & T_6) left, and they would have no edges between them. Next, the worker w_1 would pick T_3, and w_2 would pick T_6. As per OCC, both the transactions would be valid, and the dirty state gets updated with their write sets. On calling

`RemoveFromGraph()`, the dependency graph would be left empty. Note that only after the committer commits a block would the dirty state be flushed.

Concurrent to the validation process, the committer would fetch block B_1 and wait for the validation results of $T_1 \cdots T_4$. When they are available, the committer commits B_1, as shown in Table 3. Then, the dirty state and results associated with the block would be deleted. Next, the committer would fetch block B_2 and immediately find validation results for all transactions. Hence, the committer would commit it and update the dirty state.

4.4 Proof of correctness

As outlined in § 4.2, there are two requirements we must meet for the correct validation of transactions. Here we prove that our approach meets both requirements.

Meeting requirement 1: When T_j is added to the dependency graph, edges are created from T_j to each T_i that writes to any state k read by T_j. If T_j has performed a range query, edges are also created to any T_i that writes to any state within that range. Whenever a validator worker is free, it calls `GetNextTransaction()` to get a transaction to be validated. However, T_j will not be returned until it has no out-edges. Thus, T_j will not be validated until every transaction T_i it depends upon (via *rw* and *pr* dependencies) is removed from the graph. Note that by **Invariant I1**, a transaction is only removed from the graph when it is fully validated. Thus, by the time T_j is given to a validator worker, all transactions writing to states it reads would have been validated.

Meeting requirement 2: When T_j is added to the dependency graph, edges are created from T_j to each T_i that either reads or writes to any state k that T_j modifies. The *ww* dependencies ensure that writes to any state k are always applied in the correct order. T_j will not leave the dependency graph until T_i has been removed from the graph, which will happen only after T_i's writes have been applied to the dirty state. Similarly, the *wr* dependencies ensure that until all the earlier transactions T_i that have read k are validated, T_j will not leave the graph, and hence its updates will not be applied to the dirty state till that point. Thus, no T_i will see a later transaction T_j's write.

5 DESIGN OF SPARSE PEER

From takeaway 2 in §3, we know that multiple peers in an organization do redundant work. To avoid this wasteful redundancy, we propose a new peer type called a *sparse peer*.

5.1 Sparseness in Validation and Commit

The idea behind a *sparse peer* is that it can selectively validate & commit transactions. If all *sparse peers* within an organization select a non-overlapping set of transactions, we can avoid redundant work. Towards achieving this, first, we define a deterministic selection logic such that each *sparse peer* selects a different set of transactions. Second, we change the validator and committer to apply the selection logic on the received block. Third, as an option, we make the peer pass the selection logic to the orderer so that it can apply the filter and send only required transactions in a *sparse block*. Thus, both network bandwidth and disk IO usage would reduce.

(1) Transaction selection filter: Each sparse peer owns a *filter* and applies it on a received block to identify which transactions to consider. The *filter* is simply a list of smart-contracts. The admin assigns/updates each peer's *filter* by issuing a request via an RPC to the peer process. The *sparse peer* only validates and commits transactions in a block that invoke a contract specified in the *filter*. If a transaction invokes multiple contracts, even if the filter contains only one of those contracts, the transaction would be considered by the *sparse peer*.

(2) Validation and commit based on a filter. The dependency extractor only considers transactions that invoke a contract present in the *filter*. When the committer gets a block, it marks transactions which do not invoke any contract from the *filter* as "not validated" but stores the whole block in the block store. The rest of the validator and committer logic remains the same. However, the receival and storage of full block would not reduce network bandwidth and disk IO. Since disk IO is in the critical path of peers, this limits the maximum possible throughput. As an optional feature, next, we allow orderers to send *sparse blocks* to peers.

(3) Block dissemination based on filters. If orderers themselves apply the filter and send only appropriate transactions via a *sparse block* to each *sparse peer*, we can save both network bandwidth and disk IO. Hence, each *sparse peer* sends its filter to an orderer to which it has connected. For each block, the orderer applies the filter and sends only the required transactions to the peer. However, this creates a problem with the hash chain and its verification. In vanilla Fabric, the orderer computes a *block hash* at block creation and stores it in the block. The *block hash* is computed using all transactions' bytes within that block and the hash present in the previous block. When a peer receives a block, it can check its integrity by verifying the hash chain. Further, this hash chain is the source of truth of a blockchain network. If we make the orderer send only a sub-set of transactions in a block, the peer cannot verify the hash chain integrity.

(4) *Sparse block.* To fix this problem, we propose a *sparse block* that includes (1) a Merkle tree to represent the *block hash* (as shown in Figure 7); (2) only

Row	Workers						Dirty State			Valid Tx			Invalid Tx			Tx waiting for other peers		
	P_1		P_2		P_3		P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3
	w_1	w_2	w_1	w_2	w_1	w_2												
1	T_1	T_2	T_2	-	T_3	-				-	-	-	-	-	-	-	-	-
2	-	T_2	T_2	-	T_3	-	(k_7, v_2)			T_1	-	-	-	-	-	T_2 awaits P_2	T_2 awaits P_1	T_3 awaits P_1, P_2
P_1 informs P_2 that T_2 is S_1-valid								P_2 informs P_1 that T_2 is S_2-valid								P_3 informs both P_1 and P_2 that T_3 is S_3-valid		
3	-	-	T_3	-	-	-	(k_7, v_2) (k_4, v_2)			T_1 T_2	T_2	-	T_3	-	-	-	-	T_3 awaits P_1, P_2
P_1 informs P_2 and P_3 that T_3 is invalid																		
4	-	-	-	-	-	-	(k_7, v_2) (k_4, v_2)			T_1 T_2	T_2	-	T_3	T_3	T_3	-	-	-

Table 6: Distributed validation of transactions

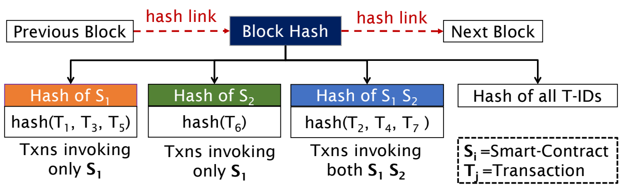

Figure 7: Merkle tree based block hash computation.

a sub-set of transactions after applying the filter; and (3) transaction identifier (T-ID) of each transaction. In the Raft-based consensus, as the leader node constructs blocks, we delegate the responsibility of creating this Merkle tree to it. The followers apply the filter before sending the block to its connected peers. When a *sparse peer* receives a *sparse block*, it can verify the hash chain integrity by verifying the Merkle tree root hash using a sub-set of transactions. A list of transaction identifiers is sent with a *sparse block* to enable the validator to check for duplicates and mark them invalid. In vanilla Fabric, the orderer does not peek into the transaction. We break this agreement, as the orderer needs to find transactions associated with each smart-contract and find transactions that invoke multiple smart-contracts. Since the orderers already have access to the entire transaction, our approach does not weaken the trust model.

5.2 Distributed Execution

In vanilla Fabric, a smart-contract can invoke another contract hosted on the same peer. With sparse peers, contracts would be placed on different peers. Hence, we enable distributed execution in which a contract hosted on one peer can invoke a contract hosted on another by making an RPC request. Each peer in an organization knows every other peer's filter and is used to decide which peer to contact for a given contract. As the distributed execution happens over a network and the *endorser* holds a read lock on the whole state DB [17], the commit operation would get delayed if there are many distributed executions. Hence, we adopt the technique proposed by Meir et al. [12] to remove the state DB's read-write lock.

5.3 Distributed Validation and Commit

Because a transaction invoking multiple contracts may have its contracts hosted on different peers, we now need distributed validation and commit as well. Consider a transaction T invoking smart-contracts S_1, S_2, $\ldots S_n$. Say we have sparse peers P_1, P_2, $\ldots$, P_n where each peer P_i has filter F_i with a single smart-contract S_i. A transaction invoking all n smart-contracts is considered valid when the transaction satisfies both the policy and serialization checks of each contract invocation. Hence, to commit this transaction, we would require an agreement from all n *sparse peers* during the validation phase.

(1) Basic idea. Each peer P_i validates parts of the transaction that invoked contracts $S_i \in F_i$, and then it broadcasts the results to every other peer. Once a peer has received *valid* as a result for all contracts $S_1 \ldots S_k$, it can consider the transaction valid and commit it. If an *invalid* result is received, the peer does not need to wait for any more results and can proceed by invalidating the transaction. As peers within an organization are trusted, there are no security issues.

(2) Illustration. Let's assume we have three sparse peers: P_1, P_2, and P_3 in an organization. Each peer has only one contract in the filter. Let's assume each peer has received a full block with 3 transactions, as listed in Table 4. Table 5 presents the current committed state at each peer's state database. On each peer, the dependency extractor would apply the filter to construct the dependency graph shown in Figure 8. As peer P_1 has contract S_1 in its filter, all three transactions are added while P_2 and P_3 have only T_2 & T_3 and T_3 in the dependency graph, respectively. Even though T_3 has read from and writes to all three contracts, peer P_2 would consider only S_2-states in the read-write set while constructing the dependency graph. The same is true for the other two peers. This does not affect the correctness, as the example below shows.

Let's assume we have two validation workers, w_1 and w_2, per peer. Each worker would pick a transaction with

Tx	R-Set	W-Set
T_1	$S_1\{k_7, v_1\}$	$S_1\{k_7, v_2\}$
T_2	$S_2\ \{k_3, v_1\}$	$S_1\ \{k_4, v_2\}$
T_3	$S_1\ \{k_4, v_1\}$ $S_2\ \{k_5, v_1\}$ $S_3\ \{k_6, v_1\}$	$S_1\ \{k_4, v_2\}$ $S_2\ \{k_3, v_2\}$ $S_3\ \{k_6, v_2\}$

Table 4: Tx. RW-sets

Peer P_1	Peer P_2	Peer P_3
S_1	S_2	S_3
(k_4, v_1)	(k_3, v_1)	(k_6, v_1)
(k_7, v_1)	(k_5, v_1)	

Table 5: StateDB at each peer

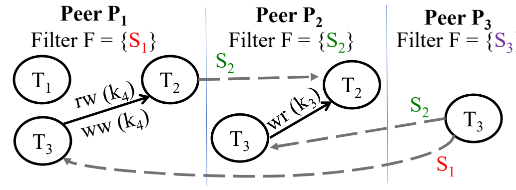

Figure 8: Tx. dependency graphs

no out-edges, as shown in row one of Table 6. The peer P_1 would find T_1 and T_2 to be valid (with respect to S_1) based on OCC [11] check. However, only T_1's write-set would be applied to the dirty state as T_2 needs S_2's validation result from P_2. Similarly, peer P_3 validates T_3 with respect to S_3, but needs validation results for S_1 and S_2 from P_1 and P_2, respectively. Once all results are shared between peers, T_2 would be marked valid on peer P_1, and the dirty state would be updated. However, on peer P_2, the dirty state would not be updated with T_2's write-set as it does not write to S_2. As T_3 has a fate dependency on T_2, peer P_1 would invalidate T_3 and share the result with the other two peers so that they can invalidate it. Row 4 in Table 6 shows the final result.

(3) Correctness. We now show how our distributed validation scheme meets **Correctness Requirement 1**. The proof of meeting correctness requirement 2 is similar to that outlined in §4.4, so we skip that for brevity.

We need to show that a transaction T_j is validated only after the validation of all transactions T_i that it depends upon is over. In §4.4, this was shown to be true, because T_j would only be considered for validation if it had no out-edges, which meant that every transaction that T_j depended upon had been fully validated.

Unfortunately, in Sparse Peers, dependency graphs may only contain a subset of transactions. Thus, even if a transaction T_j has no out-edges in a peer's dependency graph, it may still depend upon a T_i that is present on another peer. E.g., in the above illustration, P_3's T_3 has no out-edges, yet it depends upon T_2 which may not have been validated by P_1 or P_2. However, notice that if T_j depends on T_i, there will be at least one peer which has both T_j and T_i in its dependency graph. We use this to show that T_j's validation can complete only after validation of every T_i it depends upon, even those on remote peers.

Once T_j has no out-edges in a peer P's graph, P can *partially* validate T_j's reads and writes to contracts in its filter F, because by construction P had tracked all writes to those contracts. But it still needs the remaining validation results of T_j from other peers. E.g., P_3 can validate T_3 wrt its filter $\{S_3\}$ right away, but must wait for its partial validation results from P_1 and P_2. Interestingly, other peers can only give T_j's partial validation results when T_j has no out-edges in their own graphs, i.e., when they have validated every T_i that T_j depends upon in their graphs. Since T_j's validation can be *complete* only after *all* partial validations are calculated, and they are calculated only after *every* T_i has been validated, it is thus proved that T_j's validation happens only after validation of every T_i that it depends upon.

(4) Deferred transactions. The approach so far has a significant drawback. Since the peers could be at different block heights due to different block commit rates (because of different workloads or transient performance issues), a transaction requiring distributed validation could take a considerable amount of time. Even if other transactions in the block may have been validated, a single transaction could halt the committer.

To avoid the *committer* getting blocked, we introduce deferred transactions. Now, the committer can commit all local and already-validated distributed transactions (i.e., a partial block commit) without waiting for other distributed transactions' validation results. When the result is available, the deferred transaction is committed and removed from the graph. Any local or distributed transaction with a dependency on a deferred transaction must be deferred.

Correctness with deferred transactions. Deferred transactions do not affect the correctness argument. By **Invariant I1**, a deferred transaction (and thus all its dependents) stays in the dependency graph until it is validated. If the peer does not crash, this is clearly sufficient for correctness. If the peer *does* crash, the dependency graph would be lost, since it is an in-memory construct. However, when committing a block with deferred transactions, the committer persists the fact that those transactions have been deferred. It also persists their partial validation results. On recovery, the peer would query other peers in the organization for the validation results of the transactions it has deferred. Every peer would either readily have their respective partial validation results of those deferred transactions, or they would eventually compute them, and thus the recovering peer would be able to decide on the validity of deferred transactions. If the results aren't readily available, the peer would place the deferred transaction in the dependency graph before processing any new blocks.

(5) Validation result sharing with replicas. To load balance the endorsement requests and to provide high availability, it is required to run multiple replicas of a spare peer. If all replicas do the same work, we end up wasting resources. Hence, we enable the replicas of a sparse peer to share validation results between them. In a replica set, one sparse peer is chosen as the leader who is responsible for validating the transaction and sharing the results with other replicas.

5.4 Auto-Scaling Primitives

From takeaway 4 in §3.2, we know that dynamically scaling a network by adding new peers takes significant time. This is because a newly added vanilla peer copies, validates, and commits all the blocks sequentially to make the State DB to sync with other peers. We propose to dramatically reduce the horizontal scale-up time by directly copying a 'snapshot' of the State DB at a given block B and then continuing regular validation and commit of blocks that come after block B.

Table 7: KV pairs in StateDB

Block	State DB
1	k_1, v_1
2	k_1, v_1, k_2, v_1
3	$k_1, v_1, k_2, v_1, k_3, v_1$
4	k_2, v_2, k_3, v_1

However, Fabric only stores the latest states in the State DB as of the last committed block, which may get overwritten by future blocks. For example, Table 7 presents the states at state DB after committing each block. To copy state DB at block 3, we need to copy the value of k_1, k_2, and k_3 at version v_1. As the peer continues to commit blocks, the latest state changes. After committing block 4, k_1 doesn't even exist in state DB, and k_2 has a new version v_2. In such cases, the older versions of k_1 and k_2 can be obtained from write-sets of transactions present in the block store. To copy states from State DB as of a given block, we add an index of the form {*contractID, block number, transaction number*} $\mapsto$ a list of {*key, isDelete, isDeferred*}. This tracks state modifications done by every transaction. As we would be scaling up by adding *sparse peers*, we would only be copying a part of the State DB. The states to be copied can be identified by doing a range query on this index. For example, if a newly joined sparse peer wants the state of contract S_1 as of block B, the full peer can run the range query [*Start*={S_1, 0, 0}, *End*={S_1, B+1, 0}) on the newly added index, and finds the keys modified in this range. It can then use the states from State DB whose version present was created in the requested block range. If a required state k is not in the State DB because it was deleted or modified by a block $B' > B$, we find the last transaction in the block range $[0, B]$ that wrote k. Then, we extract k's value from the transaction's write-set. Once the State DB is copied, the new peer can start sharing the load. Simultaneously, the admin can update the filter (i.e., removing a contract) of full peer to make it a sparse peer. If the new peer will further split into more sparse peers later, we need to copy the index and block store in the background, which can happen slowly. Instead, if the new peer needs to be spun up to handle transient load spikes, the State DB copy would suffice.

6 EVALUATION

We implemented our proposals by adding 15K lines of GoLang code to Hyperledger Fabric v1.4. Hereafter, we refer to our approach as **SmartFabric**. We compare SmartFabric with vanilla Fabric and FastFabric [10], which claims to achieve 20K tps. A follow-up work [9] showed that the stable implementation [8] of FastFabric achieved 14K tps on a benchmark similar to Smallbank. Briefly, FastFabric separates a peer's roles to 3 separate node types: Endorser Peers (EP), FastPeer (FP) and Storage Peers (SP). Only SP commits both blocks and state on the disk. Other nodes only store state in RAM. Note that we have applied one of their optimizations (block deserialization cache) to both vanilla Fabric (as mentioned in §3) and SmartFabric. First, we focus our evaluation on the following guiding questions:

Q1. Given a fixed cost of infrastructure, how much higher throughput can SmartFabric provide?

Q2. Given a required throughput, by how much does SmartFabric reduce the infrastructure cost?

Next, we inspect the internals to show the efficiency of each proposed optimization. For experiments, we use the same default configurations described in §3.

6.1 Vertical Scaling

To study the impact of vertical scaling on SmartFabric and FastFabric, we considered two scenarios: (1) single peer per organization; (1) four peers per organization. In scenario (1), *sparse peers* and FastFabric are not applicable as they require at least 2 peers per organization.

(1) Single peer per organization. Figure 9(a) plots the impact of number of vCPUs on the throughput and CPU utilization under the Smallbank workload. SmartFabric provided 1.4× higher throughput (on average) than Fabric for the same infrastructure cost. Further, with an increase in the number of vCPUs from 1 to 16, SmartFabric's throughput improved 7× compared to 5.5× improvement with Fabric. Similarly, SmartFabric required only 8 vCPUs to provide higher than Fabric's throughput with 16 vCPUs, i.e., better performance at half the cost. We obtained similar results with YCSB but omitted the plot for brevity. This improvement is

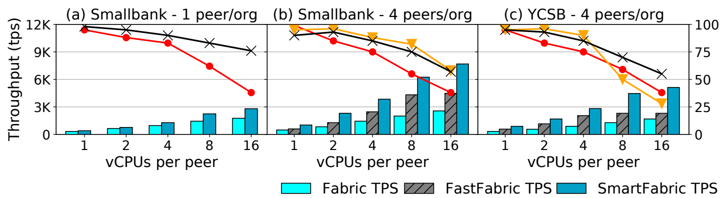

Figure 9: Vertical Scaling

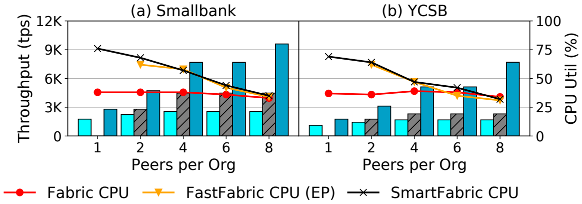

Figure 10: Horizontal Scaling

because Fabric underutilizes the CPU beyond a point while SmartFabric maintains a high utilization. For example, with 16 vCPUs, SmartFabric demonstrates 76% CPU utilization versus 36% utilization with Fabric.

(2) Four peers per organization. Figures 9(b) and 9(c) plot the impact of number of vCPUs on the throughput and CPU utilization for Smallbank and YCSB, respectively. SmartFabric provided 2.7× and 1.7× higher throughput (on average) than Fabric and FastFabric, respectively, for the same infrastructure cost. For example, with 16 vCPUs, SmartFabric provided 7680 tps versus Fabric's 2560 tps and FastFabric's 4480 tps. Further, Fabric's peak throughput with 16 vCPU was met by FastFabric with 4 vCPUs, while SmartFabric provided comparable performance with just 2 vCPUs (87% lower cost). There are two interesting observations: (1) the CPU utilization of FastFabric and SmartFabric is almost the same under Smallbank, yet there is a performance gap; (2) the CPU utilization of FastFabric suddenly dropped when the number of vCPUs > 4 under YCSB. The reasons are explained in the next section.

6.2 Horizontal Scaling comparison

Figure 10 plots the impact of number of peers in an organization on the throughput and CPU utilization. Each peer has 16 vCPUs. For FastFabric, when there are n nodes, we configure them such that there is 1 FP, 1 SP and $n-2$ EP nodes. FastFabric requires a single dedicated FP node. We only keep one SP because, like vanilla Fabric, each SP node would do the same work of committing the entire block to disk. Adding more EP nodes helps by allowing higher transaction endorsement. When there are only 2 nodes, we overlap EP and SP roles on the same node, because FP must be a dedicated node. For SmartFabric, we evenly divide m smart contracts over n nodes. To every node's *filter*, we first add $\lfloor \frac{m}{n} \rfloor$ contracts. We then add one *additional* contract to m mod n nodes. We call these nodes 'unevenly loaded', while others are referred to as 'evenly loaded'. In our case, $m = 8$. So, when $n = 6$, each node gets 1 contract, and two nodes get an additional contract each.

SmartFabric provided 2.7× and 1.8× higher throughput (on average) than Fabric and FastFabric respectively for the same number of peers per organization. Further, with just a single node SmartFabric achieved more throughput than what Fabric achieved by using 4 nodes, i.e., better performance with 75% lower cost. Similarly, SmartFabric provided higher throughput with 2 nodes than FastFabric with 3 or more nodes. Interestingly, both Fabric and FastFabric did not provide higher throughput beyond 4 peers per organization. As we have already explained Fabric's behaviour in Section 3, we explain the behaviour of FastFabric next.

vCPU	1 peer	4 peers	8 peers
1	320 vs 400	480 vs 1020	-
4	960 vs 1280	1440 vs 3840	-
16	1760 vs 2800	2560 vs 7680	2560 vs 9600

Table 8: Throughput (tps) of Fabric vs SmartFabric

FastFabric did not benefit by adding even the fourth peer as it got bottlenecked by the SP node. In § 3, we showed that a vanilla peer that does not endorse transactions could achieve 3300 tps. An SP node is exactly like this, except it gets pre-validated transactions and only commits them. Therefore, its peak throughput increased to 4480 tps. Figure 12 shows the block processing times and the VC TPS$_{Limit}$ of each node. The actual throughput matched the VC TPS$_{Limit}$ curve of SP nodes.

SmartFabric on the other hand gave more throughput with 8 peers, but not with 6 peers. When we have 6 nodes, four are evenly loaded because they manage one contract. The other two are unevenly loaded as they manage two contracts. Thus, the slowest node in this case still similar to the case with 4 nodes (see Figure 11). The VC TPS$_{Limit}$ of evenly loaded peers increased as the number of contracts they manage decreased but the actual throughput was dictated by unevenly loaded peers. With 8 peers, every peer managed one contract. Thus, in reality, the benefit of *sparse peers* would depend upon the number of contracts, their load distribution and the number of peers in the organization.

6.3 Impact of combind scaling techniques

Table 8 compares the throughputs of Fabric with SmartFaric for various configurations. Going down the table

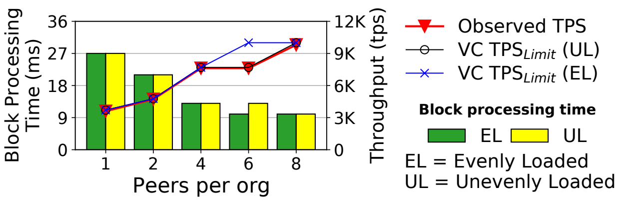

Figure 11: Effect of imbalanced load on Sparse Peers

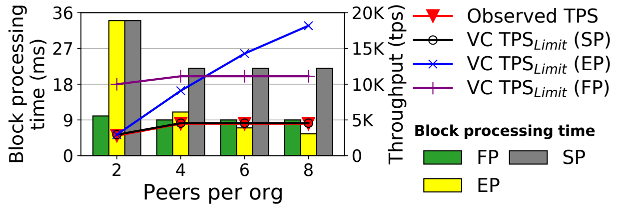

Figure 12: FastFabric's block processing time

displays the vertical scalability, going horizontally showcases the horizontal scalability, and diagonal numbers show the impact of the combination. It is easy to see that SmartFabric scales both vertically and horizontally, showing increasing speedups over Fabric as more resources are added.

6.4 Inspecting internals

Now, we show the efficiency of each optimizattion. All experiments are done with 4 peers per organization (16 peers in total), each having 16 vCPUs.

(1) Pipelined Validation and Commit Phases. In this experiment, all peers were full peers. We first compare the peak performances achieved on the Smallbank workload. Figure 13(a) plots the throughput achieved with the pipelined execution against vanilla Fabric. As expected, there was a 1.4× improvement over vanilla Fabric while increasing the CPU utilization from 40% to 60%. The validation manager was so efficient that the committer never got blocked. The size of the *result-map* was always higher than 200. This is because the time taken by the committer (≈22 ms at 3600 endorsement requests per second—eps) was always higher than the time taken by validators. Further, the end-to-end commit latency (validation + commit) for a block reduced from 39 *ms* (at 2560 eps) to 27 ms (at 3600 eps).

To study the effect of different degrees of dependencies between transactions, in YCSB, we varied the skewness of the Zipf distribution, that was used to select the keys: we went from an s-value=0.0 (uniform) to an s-value=1.5 (highly skewed) in steps of 0.5. Figure 13 plots the throughput and goodput achieved with pipelined execution against vanilla Fabric. The reason for considering goodput is that certain transactions get invalidated due to serializability conflicts. As expected, the pipelined execution outperformed vanilla Fabric when the s-value was low. As the s-value increased, the more transactions were invalidated, thus reducing the goodput and performance gain with pipelining. It is interesting to note that the throughput increased with a decrease in the goodput as the peer did not spend much time on the commit.

(2) Sparse Peer with Full and Sparse Blocks. We evaluate the performance of two variants of sparse peer proposed in §5. Each organization hosted 4 sparse peers where the filter of each sparse peer contained only 2 non-overlapping smart-contracts. Figure 14(a) plots the throughput achieved with both variants of sparse peers against a network where each organization hosted 4 vanilla peers. As expected, the throughput increased significantly by 2.4× with sparse peers for both workloads. Compared to the sparse peer processing full blocks, the sparse peer processing sparse block achieved higher throughput due to the reduced IO operation. Figure 14(b) plots the throughput achieved with the combination of sparse peer and pipelined execution against vanilla Fabric. The throughput increased significantly to 7680 tps, i.e., 3× that of vanilla Fabric.

(3) Distributed Simulation and Validation. When transactions invoke multiple smart-contracts, SmartFabric employs distributed simulation and validation. To study our proposed system's performance in the presence of distributed transactions, we submitted transactions that invoked multiple smallbank contracts. We describe the details of our setup in our technical report [16]. Figure 15 compares the throughput of vanilla Fabric against SmartFabric under a varying mix of cross-contract invocation transactions. We consider scenarios where cross-contract transactions invoke 2 and 3 contracts respectively. As expected, SmartFabric's performance degraded as the percentage of transactions invoking multiple contracts increased. This is because, with more transactions invoking multiple contracts, the amount of work to be done at each peer increased. However, even with 100% cross-contract transactions, SmartFabric outperformed vanilla Fabric by 2× in both scenarios.

Interestingly, the number of transactions deferred per second wasn't substantial either. Even when 100% of transactions invoked two and three contracts, only ≈ 200 tps and ≈ 400 tps respectively got deferred. In other words, merely 5 to 10% of issued transactions were getting deferred even when every transaction required distributed validation. The low deferral rate is because, with pipelined validation and commit phases transactions were getting validated much earlier than their results were required by the committer. The extra latency experienced by deferred transactions ranged from 30 to 50 ms. Given that blockchain transactions already take several hundreds of milliseconds [6], this is quite a small

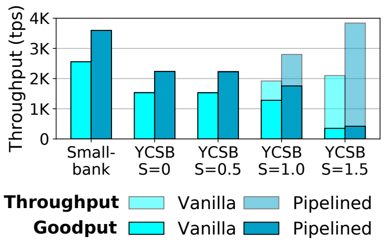

Figure 13: Effect of dependencies

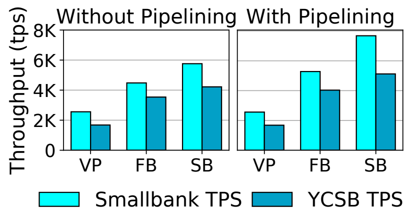

Figure 14: Vanilla Peer (VP) vs Full Blocks (FB) & Sparse Blocks (SB)

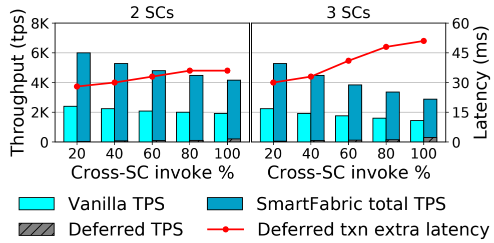

Figure 15: Distributed transactions

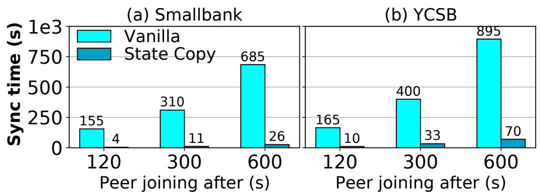

Figure 16: New peer sync time

penalty to pay for a big improvement in throughput. However, note that this duration is of the order of one or two block processing times in SmartFabric. Without deferring, the average block processing duration would increase 2× or worse due to waiting, thus drastically reducing the peak throughput. That would be a significant penalty to pay for not deferring 5 to 10% of transactions. Thus, deferring transactions is key to this performance.

(4) Scaling Up and Down. We evaluate the dynamic scaling approach discussed in §5.4. Figure 16 plots the time taken for a new peer to copy states from another. For a fair comparison with vanilla, we add a new sparse peer with all smart-contracts (the equivalent of a full peer) and generate the same load on existing peers. SmartFabric provided a multifold reduction over a vanilla peer's sync time at various minutes. This is because SmartFabric copied a much smaller amount of data. In general, the size of block store is several times higher than the size of state DB. As we copy only the required states from the state DB, a new peer can join very quickly. In vanilla Fabric, we could turn off a peer for scaling down a network. However, scaling down sparse peers would require copying states from the sparse peers to other peers. The time to copy the states was similar to the scale-up time, which is quite small. Compared to the benefits of sparse peers, this penalty is insignificant.

7 RELATED WORK

Broadly, the past work follows two themes: (1) Improving the raw blockchain throughput and (2) Reducing abort rate in `EOV` blockchains. Our work focuses on the first.

Thakkar *et al.* [17] conducted a comprehensive performance study and found bottlenecks in Fabric v1.0 and provided guidelines to design applications and operate the network to attain a higher throughput. Further, they implemented a few optimizations on the peer process. These optimizations have already been included in Fabric v1.4, and hence our work builds upon this work.

Gorenflo *et al.* [10] proposed FastFabric that includes optimizations such as replacing the state DB with a hash table, storing blocks in a separate server, separating the committer and endorser into different servers, parallelly validating the transactions headers, and caching the unmarshed blocks to reach a throughput of 20000 tps. However, many of these optimizations are not practical for a production environment. For example, a state DB is a must to support range queries and persist all current states (to help a peer recover quickly after a failure).

Sharma *et al.* [15] used ideas from the database literature to reorder transactions within a block by analzing their dependencies, with the goal of reducing transaction aborts. Ruan *et al.* [14] extended [15] to abort unserializable transactions before ordering and to altogether avoid transaction aborts within a block by reordering them. These techniques are orthogonal to our work as we do not modify the ordering of transactions. We focus on pipelined execution of different phases and to avoid redundant work within an organization.

Gorenflo *et al.* [9] extended FastFabric [10] to introduce post-order execution of transactions to reduce the transaction aborts. They constructed a dependency graph at the peer with rw, ww, wr dependencies. Whenever there was a conflict between a transaction, they re-executed the patch-up code passed with the transaction to reduce the transaction aborts. This work is orthogonal to our approach.

8 CONCLUSION

In this paper, we studied the performance of the EOV architecture of blockchains using various scaling techniques and identified two major bottlenecks: (1) serial execution of validation and commit phases in the critical path; (2) duplication of CPU and IO intensive tasks. Toward this, we introduced a pipelined execution of the bottleneck phases and a new peer type called *sparse peer*. Finally, we introduced a mechanism that can make autoscaling realistic in blockchain networks.

REFERENCES

[1] 2021. Amazon AWS Cost Estimator. https://calculator.aws/. (2021). https://calculator.aws/

[2] 2021. Azure Cost Estimator. ihttps://azure.microsoft.com/en-in/pricing/calculator/. (2021). ihttps://azure.microsoft.com/en-in/pricing/calculator/

[3] 2021. Google Cloud Cost Estimator. https://cloud.google.com/products/calculator. (2021). https://cloud.google.com/products/calculator

[4] 2021. IBM Cloud Cost Estimator. https://cloud.ibm.com/estimator/review. (2021). https://cloud.ibm.com/estimator/review

[5] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *2008 IEEE 24th International Conference on Data Engineering*. 576–585. https://doi.org/10.1109/ICDE.2008.4497466

[6] 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. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[8] C. Gorenflo. 2020. FastFabric 1.4 implementation: https://github.com/cgorenflo/fabric/tree/fastfabric-1.4. (2020). https://github.com/cgorenflo/fabric/tree/fastfabric-1.4

[9] C. Gorenflo, L. Golab, and S. Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–9. https://doi.org/10.1109/ICBC48266.2020.9169478

[10] C. Gorenflo, S. Lee, L. Golab, and S. Keshav. 2019. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 455–463. https://doi.org/10.1109/BLOC.2019.8751452

[11] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[12] Hagar Meir, Artem Barger, and Yacov Manevich. 2019. Increasing Concurrency in Hyperledger Fabric. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. ACM, New York, NY, USA, 179–179. https://doi.org/10.1145/3319647.3325841

[13] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[14] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-Order-Validate Blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 543–557. https://doi.org/10.1145/3318464.3389693

[15] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines Between Blockchains and Database Systems: The Case of Hyperledger Fabric. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. ACM, New York, NY, USA, 105–122. https://doi.org/10.1145/3299869.3319883

[16] Parth Thakkar and Senthil Nathan. 2020. Scaling Hyperledger Fabric Using Pipelined Execution and Sparse Peers. arXiv:2003.05113 [cs.DC]

[17] P. Thakkar, S. Nathan, and B. Viswanathan. 2018. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 264–276. https://doi.org/10.1109/MASCOTS.2018.00034