SHARON: Secure and Efficient Cross-shard Transaction Processing via Shard Rotation

Shan Jiang, Jiannong Cao, Cheung Leong Tung, Yuqin Wang, Shan Wang
Department of Computing, The Hong Kong Polytechnic University

***Abstract*—Recently, sharding has become a popular direction to scale out blockchain systems by dividing the network into shards that process transactions in parallel. However, secure and efficient cross-shard transaction processing remains a vital and unaddressed challenge. Existing work handles a cross-shard transaction via *transaction division*: dividing it into sub-transactions, processing them separately, and combing the processing results. Such an approach is unfavorable for decentralized blockchain due to its reliance on trustworthy parties, e.g., the client or a reference node, to perform the transaction division and result combination. Furthermore, the processing result of one transaction can affect another, violating the important property of transaction isolation. In this work, we propose Sharon, a novel sharding protocol that processes cross-shard transactions via *shard rotation* rather than transaction division. In Sharon, shards rotate to merge pairwisely and process cross-shard transactions when merged. Sharon eliminates reliance on trustworthy parties and provides transaction isolation in nature because transactions are no longer divided. Nevertheless, it poses a scientific question of when and how to merge the shards to improve system performance. To answer the question, we formally define the shard scheduling problem to minimize transaction confirmation latency and propose a novel construction algorithm. The proposed algorithm is proven optimal and runs in polynomial time. We conduct extensive experiments on Amazon EC2 instances using Bitcoin and Ethereum data. The results indicate that Sharon achieves nearly linear scalability, improves the system throughput by 139%, and saves the transaction processing latency by 72.4% compared with state-of-the-art approaches.**

***Index Terms*—Blockchain sharding, cross-shard transaction processing, shard scheduling.**

I. INTRODUCTION

Blockchain technology has been receiving extensive attention from the research community and industries due to its distinctive features of rebuilding trust in trustless environments [1]. The first and most successful blockchain application, Bitcoin, is a cryptocurrency and used only 12 years to reach a US$1 trillion market capitalization [2]. In comparison, Microsoft, Apple, Amazon, and Google used more than 20 years. Besides cryptocurrencies, blockchain technology is widely adopted for secure and trustworthy data storage in many applications, including supply chain management [3] and healthcare information exchange [4]. However, the inferior system throughput of blockchains severely limits a broader range of applications. In particular, the two most successful public blockchains, i.e., Bitcoin and Ethereum, can only process around 9 and 29 transactions per second, respectively [5]. In comparison, modern financial systems process more than $4,000$ transactions per second. Therefore, improving the system throughput of public blockchains is highly demanded.

The primary reason for the limited system throughput of public blockchains is that the whole blockchain network needs to make consensus to keep a strongly consistent data replica [6]. It means more worldwide nodes in the blockchain network make consensus and consistency more complex, thus decreasing the system throughput. Sharding is an essential and promising technique to improve the system throughput of blockchains [7]. A sharding protocol divides the blockchain network into sub-networks, i.e., shards, that process transactions in parallel [8]. Generally, the system throughput will increase with more nodes owing to the parallelization.

However, designing a sharding protocol is non-trivial because it consists of complex steps, including partitioning the network into shards, separating and allocating the transactions to the shards, and balancing the storage [9]. Among the challenges, secure and efficient cross-shard transaction processing is outstanding because it dramatically affects the system security and throughput [10]. Specifically, each shard only stores partial data and cannot validate every transaction [11]. In this case, some transactions cannot be processed by any single shard and need multiple shards to cooperate to handle. They are called cross-shard transactions and significantly decrease the parallelization degree of a sharding system. According to [12], more than 90% transactions are cross-shard.

Fig. 1 depicts three traditional approaches to handling a cross-shard transaction tx. In the first approach [13], the client that submits tx will send tx to corresponding shards, i.e., shards A and B in Fig. 1(a), for validation. Then, shards A and B return the signed validation results to the client so that the client can commit tx. Such an approach relies on the active participation of the client. In practice, client devices have limited resources and unstable networks and are not always online, making such an approach impracticable. Furthermore, the clients can repeatedly validate transactions while refusing to commit them, leading to resource profligacy and performance degradation.

The second approach relies on special shards [12]. A special shard divides tx into sub-transactions tx' and tx'' and sends them to shards A and B for validation, respectively. Then, the special shard can commit tx upon receiving the validation results. Such an approach is similar to the first one but replaces the client with a special shard. However, the special shard has greater power than the others, leading to centralization concerns. Furthermore, it can no longer provide the important property of transaction isolation because the processing result of one transaction affects another. For example, consider two transactions tx_1 and tx_2, transferring some tokens from

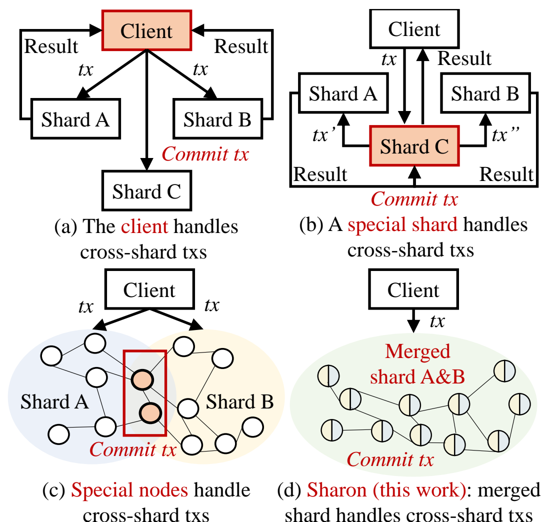

Fig. 1. Comparison between Sharon and traditional sharding protocols when processing a cross-shard transaction tx: a) the client sends tx to the corresponding shards for validation, collects the results, and commits tx, b) a special shard receives tx, sends tx to the corresponding shards for validation, collects the results, and commit tx, c) a group of special nodes joining multiple shards receives, validates, and commits tx, d) Sharon (this work): shards merge to process tx, without reliance on trustworthy parties.

accounts A and B to other accounts. Transaction tx_1 can be invalid because account B does not have enough tokens; however, account A's tokens are enough and locked. The locking can lead to invalidation of tx_2 although it is valid. In this regard, the processing result of tx_1 affects tx_2.

More recently, some researchers propose allowing some nodes to join multiple shards [14], as depicted in Fig. 1(c). Some special nodes, called b-shard nodes, join multiple shards to handle cross-shard transactions. Such an approach is still inadequate. First, only a limited number of b-shard nodes participate in cross-shard transaction processing, raising security and centralization concerns. Second, the b-shard nodes demand manual configuration, e.g., the number of b-shard nodes from which shards. Improper configuration results in resource underutilization and even vulnerabilities.

Some other methods target to reduce cross-shard transactions [15], [16] instead of processing. A popular approach is to model the accounts and transactions as a graph and perform graph partitioning algorithms. The objective is to split the graph into subgraphs so that the edges crossing subgraphs are minimized while the subgraphs' weights are balanced. In this way, the number of cross-shard transactions can be reduced significantly without affecting the high parallelism among shards. However, more than 90% transactions are cross-shard in existing blockchains [14]. These methods can only reduce around half of the cross-shard transactions but are far from complete elimination, which is nearly impossible. Therefore, secure and efficient cross-shard transaction processing mechanisms are still highly demanded.

This work focuses on the key challenge, cross-shard processing, in sharding for public blockchains. We propose a novel protocol, Sharon shown in Fig. 1(d), that processes cross-shard transactions via shard merging and rotation rather than transaction division. In particular, when two shards A and B are merged, the merged shards can handle the transactions crossing A and B straightforwardly. Sharon does not rely on the client, special shard, or special nodes; every node plays an identical role. Meanwhile, Sharon provides transaction isolation because each transaction is considered a whole and never split. Despite the distinctive features of Sharon, it remains a challenge when and how to schedule the merging and rotation among shards. We formally formulate a shard scheduling problem of minimizing transaction confirmation delay to address the challenge. Then, we design a construction algorithm that optimally solves the shard scheduling problem in polynomial time. We prove the optimality and time complexity of the proposed algorithm to be $\mathcal{O}(n^2)$, where n is the number of shards. The main contributions of this work are as follows:

- We propose Sharon, a novel blockchain sharding protocol that processes cross-shard transactions via merging shards rather than dividing transactions. Such an approach eliminates the reliance on trustworthy parties and guarantees transaction isolation in nature.
- We formally define the shard scheduling problem of scheduling shard meetings to make all pairs of shards met in minimum rounds. The target is equivalent to minimizing the maximum transaction confirmation delay.
- We propose an optimal construction algorithm to solve the shard scheduling problem and prove the optimality rigorously. The proposed algorithm runs in $\mathcal{O}(n^2)$ polynomial time where n is that number of shards.
- We conduct extensive performance evaluation on AWS with Bitcoin and Ethereum data. The experimental results indicate that Sharon achieves nearly linear scalability, improves the system throughput by 139%, and saves the transaction processing latency by 72.4% on average compared with state-of-the-art approaches.

II. Related Work

A. Sharding for UTXO-based Transactions

Blockchain sharding has been a hot topic since 2016. Elastico is the first sharding protocol for public blockchains [8]. In Elastico, the blockchain network is uniformly divided into smaller shards that can parallelly process disjoint sets of transactions. Elastico employs proof-of-work to validate the role of blockchain nodes and byzantine agreement protocol for intra-shard consensus. However, Elastico is dedicated to UTXO (unspent transaction outputs)-based transaction model and cannot handle cross-shard transactions securely and efficiently. Furthermore, the performance of Elastico is limited.

Omniledger [13] and RapidChain [12] are two representatives focusing on the UTXO-based transaction model and improving Elastico. In Omniledger, a verifiable random function is proposed to elect leaders efficiently, and a fast consensus protocol is designed assuming partial synchrony [13]. A major limitation of Omniledger lies in its heavy reliance on the clients to confirm cross-shard transactions. That is, the client needs to collect the confirmation results of cross-shard transactions and commit them. RapidChain improves the performance by introducing high-efficiency intra-shard consensus and shard

reconfiguration [12]. However, RapidChain processes cross-shard transactions by dividing them into sub-transactions that individual shards can process. Such an approach increases the number of transactions and relies on trustworthy reference nodes to partition transactions. Furthermore, RapidChain cannot provide transaction isolation because the processing result of one transaction affects another.

In literature, other similar work involves special roles to process cross-shard transactions. For example, ABS considers the satellite-based internet of things scenario and employs the satellites as the rendezvous points to gather and confirm cross-shard transactions [17]. CycLedger introduces leaders and backups to handle cross-shard transactions efficiently [18].

B. Sharding for Account-based Transactions

In the UTXO-based transaction model, the transactions can be evenly partitioned because the public keys are generated randomly, even for the same account. Account-based transaction model is different because an account uses the same public key to sign transactions. To this end, it is challenging, yet an opportunity, to partition the transactions evenly and distribute the portions to shards [16]. Furthermore, a proper partition of transactions based on account numbers can reduce the number of cross-shard transactions.

Allowing one node to join multiple shards is one of the initial ideas to reduce the amount of cross-shard transactions. For example, Pyramid allows nodes with superior hardware to participate in multiple shards to validate and execute the cross-shard transactions without splitting [14]. An optimization framework is proposed to compute the optimal sharding strategy maximizing the system throughput subject to the resource constraint and security level. Such an idea is also used in Prophet [19] and CDT-B [20]. Mizrahi et al. reduced cross-shard transactions based on memory usage rather than transaction partition [21]. OptChain identifies and predicts related transactions and groups them into the same shard to reduce cross-shard transactions [15].

Besides reducing cross-shard transactions, achieving a more balanced workload among shards is another approach to improving the sharding performance for the account-based transaction model. In BrokerChain [16], Huang et al. found that traditional sharding protocols, such as Monoxide [22], lead to an extremely unbalanced workload among shards. The unbalanced workload makes the resources of less-loaded shards underutilized. To this end, BrokerChain constructs a weighted graph for the accounts and transactions and proposes an account segmentation algorithm to allocate accounts to shards, balancing the workload. LB-Chain considers a dynamic setting [23]. More specifically, LB-Chain periodically offloads the workload of active accounts from shards with heavy loads to ones with light loads. Similar ideas appear in Transformers [24] and EfShard [25].

C. Sharding with Special Settings

The related work above concerns the sharding of public blockchains on traditional hardware. Some other work discusses sharding with special settings: permissioned blockchains and trusted hardware. With the special settings, the performance of sharding protocols can be much higher.

A permissioned blockchain is a blockchain that only allows authenticated nodes to join the consensus process. Permissioned blockchains are immune to Sybil attacks due to authentication in nature. Therefore, the procedure of node validation can be skipped, contributing to the reduction of time overhead. Sharper [26] forms the permissioned blockchain ledger as a directed acyclic graph, and each shard is responsible for only a view of the graph. It employs decentralized flattened protocols to handle cross-shard transactions safely. Aeolus [27] introduces an execution master to distribute transactions to shards optimally, considering the available resources of shards. S-store [28] considers the shard reconfiguration issue in sharding permissioned blockchains. When an existing shard is removed, or a new shard is formed, many blockchain nodes need to join new shards and synchronize the data, leading to heavy communication overhead. S-store employs a consistent hashing algorithm to reduce the number of rearranged nodes to reduce the communication overhead.

Trusted hardware can ensure secure data storage, processing, and protection from malicious access. Typical trusted hardware includes Intel SGX and ARM Trustzone. With trusted hardware, many time-consuming procedures can be completed quickly with fewer resources, e.g., secret sharing, transaction verification, and smart contract execution. Dang et al. [29] designed a shard formation protocol leveraging trusted hardware to securely assign blockchain nodes to shards. Benzene [30] uses trusted hardware to process cross-shard transactions securely and efficiently, achieving up to thirty thousand transactions per second.

D. Summary

This work focuses on public blockchain sharding on traditional hardware. Handling cross-shard transactions securely and efficiently is a key challenging issue in this context. Existing solutions attempt to reduce but cannot cross-shard transactions. The state-of-the-art solutions decompose cross-shard transactions into sub-transactions that individual shards can process. Such an approach lacks transaction isolation and relies on a trustworthy party for transaction decomposition. This work proposes a novel sharding protocol that merges shards rather than decomposing transactions to achieve secure and efficient cross-shard transaction processing.

III. Sharon Protocol Overview

This work proposes Sharon for secure and parallel cross-shard transaction processing in public blockchains. As shown in Fig. 2, Sharon starts with shard formation and then proceeds in epochs, in which each epoch consists of intra-shard consensus, cross-shard consensus, and shard reconfiguration.

In shard formation, each node in the blockchain network needs to go through proof-of-work testing with a small difficulty. The testing is to prevent the blockchain system from incurring the Sybil attack. Suppose that m nodes passed the testing. Then the m nodes will agree on the number of shards n and be mapped to shards randomly using random numbers [8].

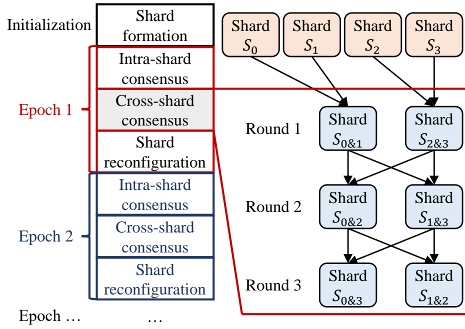

Fig. 2. Overview of the Sharon protocol. Sharon starts with shard formation and runs epoch by epoch. Each epoch consists of intra-shard consensus, cross-shard consensus, and shard reconfiguration. The cross-shard consensus procedure runs round by round. In each round, shards form paired joint shards, and each joint shard makes a consensus to process cross-shard transactions.

In this way, the m nodes have formed n shards with an approximately even size. Note that the number of nodes in each shard will be kept small, i.e., less than 100, to take care of the performance. That is, Sharon will make $\frac{m}{n} < 100$.

The shard reconfiguration procedure consists of two parts: 1) validating existing nodes and incorporating new nodes through proof-of-work and 2) adjusting the mapping between nodes and shards. The first part is straightforward and similar to the shard formation procedure. The second part is to balance the shard size and prevent the whole system from incurring attacks. On the one hand, if a large number of nodes newly join (a larger m), the number of shards should increase (a larger n) to keep the average shard size small (a balanced $\frac{m}{n}$). Similarly, we need to decrease the number of shards when too many nodes leave. On the other hand, it can raise security issues, e.g., slowly adaptive adversary and join-leave attacks [31], if the nodes keep in the same shard for a long time. However, it incurs high communication overhead when a node joins a new shard. Sharon employs the Cuckoo rule [12] and S-Store scheme [28] to rearrange a small number of nodes with few data among shards.

We employ practical byzantine fault tolerance as the consensus algorithm due to its perfect finality and high efficiency in the case of a small number of participating nodes [32]. In the following, we name a *consensus instance* as the execution of the practical byzantine fault tolerance protocol. In intra-shard consensus, each shard will run a consensus instance to confirm the transactions that can be processed by itself. Because each shard only stores partial blockchain data, some transactions may need multiple shards to process them. These transactions are called cross-shard ones and will be processed by the cross-shard consensus procedure.

This work focuses on the security and efficiency of cross-shard consensus. Consider a cross-shard transaction tx that needs to be processed by two shards $\mathcal{S}_0$ and $\mathcal{S}_2$. Most existing sharding protocols deal with cross-shard transactions in three steps. First, tx is divided into two sub-transactions A and B that can be processed by and are sent to shards $\mathcal{S}_0$ and $\mathcal{S}_2$, respectively. Second, shards $\mathcal{S}_0$ and $\mathcal{S}_2$ process A and B, respectively. The processing result can be validated or invalidated. Finally, the two processing results are combined to decide the processing result of the original tx.

Such an approach cannot achieve transaction isolation and rely on the participation of clients or reference nodes. More specifically, consider two transactions tx_a and tx_b, in which tx_a cross shard $\mathcal{S}_i$ and $\mathcal{S}_j$ while tx_b cross shards $\mathcal{S}_j$ and $\mathcal{S}_k$. One of tx_a and tx_b can be validated; however, it is possible that neither tx_a nor tx_b is validated using existing sharding protocols. That is, these protocols fail to provide transaction isolation. Moreover, existing protocols rely on the active participation of clients and reference nodes to combine the processing results of sub-transactions and decide the final result [13], [29]. The approach is vulnerable if the clients and the reference nodes are malicious.

In Sharon, we propose to combine shards instead of dividing transactions as shown in Fig. 1(d). Consider a transaction crossing shards $\mathcal{S}_0$ and $\mathcal{S}_2$, then the transaction can be confirmed when shard $\mathcal{S}_0$ and $\mathcal{S}_2$ are merged. Shard merging can provide the favorable feature of transaction isolation because each transaction is considered a whole rather than many pieces. Moreover, there is no need for the participation of clients or reference nodes. Regarding the transactions that cross more than two shards, Sharon processes them by accumulating intermediate results. For example, consider a transaction tx crossing three shards $\mathcal{S}_0$, $\mathcal{S}_1$, and $\mathcal{S}_2$. When $\mathcal{S}_0$ and $\mathcal{S}_1$ meet, they will process tx and save its intermediate processing result. Next, when $\mathcal{S}_0$ or $\mathcal{S}_1$ meets $\mathcal{S}_2$, the intermediate processing result can be combined with the one from $\mathcal{S}_2$, leading to the final processing result of tx. Despite the advantages, shard merging faces two challenging issues. On the one hand, a large amount of data needs to be synchronized when a node is merged into a new shard. On the other hand, it remains a question of how to schedule the shard merging to optimize the overall system performance.

Regarding shard merging, the objective is to make each pair of shards meet from time to time to process cross-shard transactions efficiently. A running example with four shards in an epoch is shown in Fig. 2. The shard merging proceeds round by round. In round 1, shards $\mathcal{S}_0$ and $\mathcal{S}_1$ form a joint shard and shards $\mathcal{S}_2$ and $\mathcal{S}_3$ form another joint shard; in round 2, shards $\mathcal{S}_0$ and $\mathcal{S}_2$ form a joint shard and shards $\mathcal{S}_1$ and $\mathcal{S}_3$ form another joint shard; in round 3, shards $\mathcal{S}_0$ and $\mathcal{S}_3$ form a joint shard and shards $\mathcal{S}_1$ and $\mathcal{S}_2$ form another joint shard. In this way, every pair of shards have met in three rounds. When a joint shard is formed, it will run a consensus instance to process cross-shard transactions. This work aims to minimize the number of rounds to make every pair of shards met to reduce the time overhead as much as possible. In the example, at least three rounds are demanded to make every pair of shards met. In the following, we formally define the shard scheduling problem, propose an optimal solution, prove the optimality, and analyze the time complexity.

IV. Optimal Shard Scheduling

This section concerns the scheduling of the shard merging and rotation. We first formally define the shard scheduling problem. Then, we present an optimal shard scheduling algorithm. Finally, we analyze the correctness and time complexity of the proposed scheduling algorithm.

A. Problem Formulation

The target of scheduling the shards is to let the shards meet regularly to confirm the cross-shard transactions. Considering a pair of shards, we aim to make the interval of their two consecutive meetings as short as possible. Note that a short interval is preferred because it indicates a short confirmation time for the transactions crossing the pair of shards. As for all shard pairs, we aim to minimize the longest interval.

We consider the shards repeat to meet epoch by epoch. In each epoch, the shards meet in a determined *meeting sequence* that guarantees all pairs of shards will meet. In this way, the longest interval of shard meetings will be no more than the duration of an epoch. Then, our problem is to determine the meeting *sequence*. The longest interval can be minimized if the *meeting sequence* makes each pair of shards meet precisely once with the shortest time. We formulate the *shard scheduling problem* to determine the optimal *meeting sequence*.

Definition 1. ***Shard scheduling problem.*** *Consider $n \geq 2$ shards that meet round by round. In each round, a shard can meet at most one another shard. Schedule the shard meetings so that all pairs of shards have met with the minimum rounds.*

A naive solution to the *shard scheduling problem* is to enumerate the permutations of the shard pairs. For each permutation, we segment it into rounds so that no shard appears twice for all the rounds. In this way, the permutation with the minimum rounds can be found and will be the answer. For example, consider four shards $\{\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$, resulting in six pairs of shards $\{\{\mathcal{S}_0, \mathcal{S}_1\}, \{\mathcal{S}_0, \mathcal{S}_2\}, \{\mathcal{S}_0, \mathcal{S}_3\}, \{\mathcal{S}_1, \mathcal{S}_2\}, \{\mathcal{S}_1, \mathcal{S}_3\}, \{\mathcal{S}_2, \mathcal{S}_3\}\}$. A permutation of the pairs of shards will be $(\{\mathcal{S}_0, \mathcal{S}_2\}, \{\mathcal{S}_1, \mathcal{S}_2\}, \{\mathcal{S}_0, \mathcal{S}_3\}, \{\mathcal{S}_1, \mathcal{S}_3\}, \{\mathcal{S}_0, \mathcal{S}_1\}, \{\mathcal{S}_2, \mathcal{S}_3\})$. It should be segmented into four rounds, i.e., $\{\mathcal{S}_0, \mathcal{S}_2\} \parallel \{\mathcal{S}_1, \mathcal{S}_2\}, \{\mathcal{S}_0, \mathcal{S}_3\} \parallel \{\mathcal{S}_1, \mathcal{S}_3\} \parallel \{\mathcal{S}_0, \mathcal{S}_1\}, \{\mathcal{S}_2, \mathcal{S}_3\}$. In such a segmentation, $\mathcal{S}_1$ meets $\mathcal{S}_2$ at the same time when $\mathcal{S}_0$ meets $\mathcal{S}_3$ in the second round.

However, the naive solution incurs an extremely high time complexity. There are $\frac{n(n-1)}{2}$ pairs of shards when there are n shards. Then, the number of permutations of the shard pairs is $\frac{n(n-1)}{2}!$. Each permutation needs $\mathcal{O}(n^2)$ time to perform the segmentation. Therefore, the naive solution takes up to $\mathcal{O}(n^2 \cdot \frac{n(n-1)}{2}!)$ time, which is exponential and unacceptable.

B. Proposed Solution

In this work, we propose an optimal solution to solving the *shard scheduling problem* in $\mathcal{O}(n^2)$ time. More specifically, we separate the cases when n is odd and even and construct the optimal *meeting sequence* respectively.

First, Algo. 1 depicts the construction algorithm when n is odd. At least n rounds are needed for every pair of n shards to meet when n is odd. The idea is to enumerate the number of rounds r from 1 to n. In round r, a shard $\mathcal{S}_i$ will try to meet shard $\mathcal{S}_{(r-1-i) \bmod n}$. Here, if i equals $(r-1-i) \bmod n$, then shard $\mathcal{S}_i$ will skip the round r, i.e., to meet no shard in round r. Note that in each round, at least one shard should remain unmet because n is odd. In Sec. IV-C, we will prove the optimality of Algo. 1.

Algorithm 1 Optimal shard scheduling when there are an odd number of shards

1: $res \leftarrow$ a list of n sets initiated to be empty sets
2: **for** $r \leftarrow 1$ **to** n **do** ▷ For round r
3: $\mathcal{L} \leftarrow$ a list of n boolean values initiated as FALSE
4: **for** $i \leftarrow 0$ **to** $n-1$ **do** ▷ For the shard $\mathcal{S}_i$
5: **if** $\mathcal{L}_{i+1} =$ FALSE **then** ▷ If $\mathcal{S}_i$ has not yet met
6: $j \leftarrow (r-1-i) \mod n$ ▷ $\mathcal{S}_i$ may meet $\mathcal{S}_j$
7: **if** $i = j$ **then** continue **end if** ▷ Avoid $j = i$
8: $\mathcal{L}_{i+1}, \mathcal{L}_{j+1} \leftarrow$ TRUE ▷ Mark $\mathcal{S}_i$ and $\mathcal{S}_j$ as met
9: $res_i \leftarrow res_i \cup \{\{\mathcal{S}_i, \mathcal{S}_j\}\}$ ▷ $\mathcal{S}_i$ meets $\mathcal{S}_j$
10: **end if**
11: **end for**
12: **end for**
13: return res

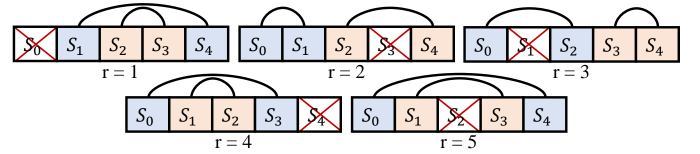

Fig. 3. A running example of Algo. 1 when $n = 5$. Every pair of shards will have met after five rounds.

Algorithm 2 Optimal shard scheduling for 2 shards

1: return $(\{\{\mathcal{S}_0, \mathcal{S}_1\}\})$

Fig. 3 depicts a running example of Algo. 1 when there are five shards ($n = 5$). In the first round ($r = 1$), consider shard $\mathcal{S}_0$ ($i = 0$), then it will try to meet $\mathcal{S}_0$ itself because $(r - 1 - i) \bmod n = 0$. However, a shard cannot meet itself, so shard $\mathcal{S}_0$ will skip the first round, i.e., to meet no shard in this round. Regarding shard $\mathcal{S}_1$ ($i = 1$), it will meet $\mathcal{S}_4$ because $1 - 1 - 1 \bmod 5 = 4$. Shard $\mathcal{S}_2$ will meet $\mathcal{S}_3$ in the first round. Similarly, in round 2, $\mathcal{S}_0$ meets $\mathcal{S}_1$ and $\mathcal{S}_2$ meets $\mathcal{S}_4$. In round 3, $\mathcal{S}_0$ meets $\mathcal{S}_2$ and $\mathcal{S}_3$ meets $\mathcal{S}_4$. In round 4, $\mathcal{S}_0$ meets $\mathcal{S}_3$ and $\mathcal{S}_1$ meets $\mathcal{S}_2$. In round 5, $\mathcal{S}_0$ meets $\mathcal{S}_4$ and $\mathcal{S}_1$ meets $\mathcal{S}_3$. After the five rounds, we can observe that all the pairs of shards, in total $\frac{5 \times 4}{2} = 10$ pairs, have met.

Second, when n is even, at most $\frac{n}{2}$ pairs of shards can meet in each round, leaving no shard alone. We consider the base case when $n = 2$ first. In this case, there is only a single pair of shards, and we can just make shards $\mathcal{S}_0$ and $\mathcal{S}_1$ meet in a round. The corresponding algorithm is shown in Algo. 2.

Third, Algo. 3 depicts the construction algorithm solving the shard scheduling problem when n is even other than 2. Consider a simple case when $n = 6$ first, and the solution can be constructed based on the case when $n = 5$. In Fig. 3, each round leaves a shard alone, and the five shards left in the five rounds are distinct. To this end, we can make the alone shard meet shard $\mathcal{S}_5$ for each of the five rounds. That is, shard $\mathcal{S}_5$ will meet $\mathcal{S}_0$, $\mathcal{S}_3$, $\mathcal{S}_1$, $\mathcal{S}_4$, and $\mathcal{S}_2$ in rounds 1 to 5, respectively. To this end, we can observe that all the pairs of shards, in total $\frac{6 \times 5}{2} = 15$ pairs, have met after five rounds.

The same idea fits any n when n exceeds 2 and is even.

Algorithm 3 Optimal shard scheduling when there are an even number of shards (not two shards)

1: $res \leftarrow$ The optimal shard scheduling result for $n-1$ shards using Algo. 1
2: **for** $r \leftarrow 1$ **to** $n-1$ **do** ▷ For round r
3: $\mathcal{L} \leftarrow$ a list of $n-1$ boolean values initiated as FALSE
4: **for each** $\{\mathcal{S}_i, \mathcal{S}_j\} \in res_r$ **do**
5: $\mathcal{L}_{i+1}, \mathcal{L}_{j+1} \leftarrow$ TRUE ▷ Mark $\mathcal{S}_i$ and $\mathcal{S}_j$ as met
6: **end for**
7: Find an arbitrary i that satisfies $\mathcal{L}_{i+1} =$ FALSE ▷ There is only one i satisfying the constraint
8: $res_r \leftarrow res_r \cup \{\{\mathcal{S}_i, \mathcal{S}_{n-1}\}\}$ ▷ $\mathcal{S}_i$ meets $\mathcal{S}_n$
9: **end for**
10: return res

Algorithm 4 Optimal shard scheduling

1: **if** $n = 2$ **then** return the result of running Algo. 2 **end if**
2: **if** n is odd **then** return the result of running Algo. 1 with the input n **end if**
3: **if** n is even **then** return the result of running Algo. 3 with the input n **end if**

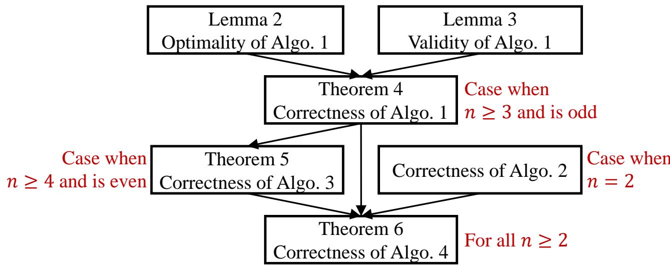

Fig. 4. Correctness proof sketch of Algo. 4. We prove the correctness of the overall algorithm by considering three cases: $n = 2$, $n \geq 3$ and is odd, and $n \geq 4$ and is even.

There will be $n \cdot (n-1)$ pairs of shards. We can construct the solution res of $n-1$ first. The solution res should contain $n-1$ rounds, in which each round contains $\frac{n-2}{2}$ pairs of shards, leaving a single shard alone. The alone shards in the $n-1$ rounds are distinct, and we can make the alone shard meet shard $\mathcal{S}_{n-1}$ when there are n shards. It results in $n-1$ pairs of shards newly met. In total, $\frac{n-2}{2} \cdot (n-1) + (n-1) = n \cdot (n-1)$ pairs of shards will have met after $n-1$ rounds.

Finally, we can synthesize Algo. 2, Algo. 1, and Algo. 3 to get the overall algorithm, Algo. 4, for optimal shard scheduling. Note that the three cases above, i.e., $n = 2$, n is odd, and n is even other than 2, can cover every possible value of n. Therefore, Algo. 4 simply checks how n falls into the three cases and selects the corresponding algorithms.

C. Correctness Proof

This subsection presents the correctness proof of Algo. 4. Particularly, the correctness concerns the validity and optimality: 1) *validity*: in the output, every pair of shards have met, and no shard meets itself or two shards in any round, and 2) *optimality*: the output uses the least number of rounds.

Fig. 4 depicts the correctness proof sketch. We consider Algo. 1 first because Algo. 3 is based on Algo. 1. In order to prove the correctness of Algo. 1, we proves its validity and optimality in Lem. 2 and Lem. 3. Then, the correctness of Algo. 3 is proven owing to the construction based on Algo. 1. Finally, the overall Algo. 4 is proven correct due to three cases: case when $n \geq 3$ and is odd in Algo. 1, case when $n \geq 4$ and is even in Algo. 3, and case when $n = 2$ in Algo. 2.

Lemma 2. *Suppose there are n shards $\{\mathcal{S}_0, \cdots, \mathcal{S}_{n-1}\}$, where $n \geq 3$ and is odd. Algo. 1 will and will only leave a single shard unmet while making all the others met for every round.*

Proof. In any round r ($1 \leq r \leq n$), Algo. 1 will leave a shard not met, or unmatched, with others only when it goes to line 7 and satisfies the condition $i = j$. To this end, we can obtain that a shard $\mathcal{S}_i$ is unmatched if and only if $i \equiv (r-1-i) \mod n$. The condition is equivalent to $r-1 \equiv 2i \mod n$ and called *the condition* in the following. We discuss the parity of r.

If r is odd, then $r-1$ is even, making *the condition* equivalent to $\frac{r-1}{2} \equiv i \mod n$. Because $0 \leq i < n$ and $0 \leq \frac{r-1}{2} < n$, we have the only solution $i = \frac{r-1}{2}$ that satisfies *the condition*.

Otherwise, r is even, making $r-1$ odd and $0 \leq r-1 < n$. If $i <= \frac{n-1}{2}$, then $2i <= n-1$. In this case, *the condition* is equivalent to $r-1 = 2i$. It is impossible to achieve so because the left side is odd while the right side is even. Therefore, only if $i \geq \frac{n+1}{2}$ could it be possible to satisfy *the condition*. In such a circumstance, $n+1 \leq 2i <= 2(n-1)$ and $1 \leq 2i-n \leq n-2$, making the condition equivalent to $r-1 = 2i-n$. To this end, $i = \frac{n+r-1}{2}$ is the only solution that satisfies *the condition*.

To summarize, Algo. 1 will and only will leave the shard $\mathcal{S}_{\frac{r-1}{2}}$ ($\mathcal{S}_{\frac{n+r-1}{2}}$) not met with others for an arbitrary odd (even) round. Except for the unmatched shard, all the others meet in pairs, which proves the lemma. □

Lemma 3. *Suppose there are n shards $\{\mathcal{S}_0, \mathcal{S}_1, \cdots, \mathcal{S}_{n-1}\}$, where $n \geq 3$ and is odd. For any shard $\mathcal{S}_i$, Algo. 1 makes $\mathcal{S}_i$ meet all the other shards.*

Proof. Consider an arbitrary shard $\mathcal{S}_i$ where $0 \leq i < n$. In a round r where $1 \leq r \leq i$ and $r \neq 2i+1 \mod n$, the shard $\mathcal{S}_i$ will meet $\mathcal{S}_j$ where $j = n+r-1-i$ in line 6 of Algo. 1. Because r ranges from 1 to i (except $2i+1 \mod n$ if inside), j will range from $n-i$ to $n-1$ (except i if inside).

Similarly, in a round r where $1+i \leq r \leq n$ and $r \neq 2i+1 \mod n$, the shard $\mathcal{S}_i$ will meet $\mathcal{S}_j$ where $j = r-1-i$. Because r ranges from $i+1$ to n (except $2i+1 \mod n$ if inside), j will range from 0 to $n-i-1$ (except i if inside).

Combining the two cases, we get that $\mathcal{S}_i$ will meet $\mathcal{S}_j$ where j ranges from 0 to $n-1$ except i, which proves the lemma. □

Theorem 4. *Algo. 1 optimally solves the shard scheduling problem when there are n shards, where $n \geq 3$ and is odd.*

Proof. First, we prove the validity of Algo. 1. On the one hand, Lem. 2 indicates that $n-1$ shards will meet in pairs for any round because only one shard will be unmatched, and there are n shards in total. It means no shard will meet more than

one shard in each round. On the other hand, Lem. 3 indicates that Algo. 1 makes all pairs of shards met.

Second, we prove the lower bound of the number of rounds. Note that there will be $\frac{n(n-1)}{2}$ pairs of shards given n shards. In each round, at most $n-1$ shards can meet in pairs, resulting in $\frac{n-1}{2}$ pairs met. We divide the number of shard pairs by the maximum number of shard pairs that can meet in each round to get the minimum possible number of rounds, that is $\frac{n(n-1)}{2}/\frac{n-1}{2} = n$ rounds. We can see that Algo. 1 runs in exactly n rounds, reaching the lower bound.

As a result, Algo. 1 is valid and reaches the lower bound, indicating that it optimally solves the *shard scheduling problem* when there n shards, where $n \geq 3$ and is odd. □

Theorem 5. *Algo. 3 optimally solves the shard scheduling problem when there are n shards, where $n \geq 4$ and is even.*

Proof. First, we prove the validity. Algo. 3 runs Algo. 1 to get the results when there are $n-1$ shards. It means the shard pairs $\{\{\mathcal{S}_i, \mathcal{S}_j\} \mid 0 \leq i, j < n-1, i \neq j\}$ already meet in a valid manner. We still have $n-1$ shard pairs not met, which are $\mathcal{U} = \{\{\mathcal{S}_i, \mathcal{S}_{n-1}\} \mid 0 \leq i < n-1\}$. In Lem. 2, we have proven that Algo. 1 leaves and only leaves the shards $\mathcal{S}_{\frac{r-1}{2}}$ not met in odd rounds and the shards $\mathcal{S}_{\frac{n-1+r-1}{2}}$ not met in even rounds. We range r from 1 to $r-1$ and can obtain that the unmatched shards are indexed $0, 1, \cdots, \frac{n-2}{2}$ and $\frac{n}{2}, \frac{n+2}{2}, \frac{2n-4}{2}$. The indexes are distinct and range from 0 to $n-2$. In lines 7 and 8, Algo. 3 finds the unmatched shard and makes it meet shard $\mathcal{S}_{n-1}$. To this end, Algo. 3 makes the shard pairs in $\mathcal{U}$ meet in a valid manner.

Second, we prove the lower bound of the rounds. There are $\frac{n}{n-1}/2$ shard pairs. Given that n is even, at most $\frac{n}{2}$ shard pairs can meet per round. To this end, it requires at least $\frac{n(n-1)}{2}/\frac{n}{2} = n-1$ rounds to make all the shards meet with each other. We can see that Algo. 3 runs in exactly $n-1$ rounds, reaching the lower bound.

Therefore, Algo. 3 is valid and reaches the lower bound, indicating that it optimally solves the *shard scheduling problem* when there n shards, where $n \geq 4$ and is even. □

Theorem 6. *Algo. 4 optimally solves the shard scheduling problem.*

Proof. Algo. 4 is exhaustive by invoking Algo. 2, Algo. 1, and Algo. 3 when there are two, odd, and even (not two) shards. Because Algo. 2, Algo. 1, and Algo. 3 optimally solve the *shard scheduling problem* in the respective cases, Algo. 4 optimally solves the *shard scheduling problem.* □

D. Time Complexity Analysis

This subsection shows that Algo. 4 runs in $\mathcal{O}(n^2)$ time where n is the number of shards.

Theorem 7. *The time complexity of Algo. 4 is $\mathcal{O}(n^2)$.*

Proof. In the following, we show that Algo. 2, Algo. 1, and Algo. 3 take $\mathcal{O}(1)$, $\mathcal{O}(n^2)$, and $\mathcal{O}(n^2)$ time, respectively. Algo. 4 invokes Algo. 2, Algo. 1, and Algo. 3. Hence, the time complexity of Algo. 4 is $\mathcal{O}(n^2)$.

Algo. 1 has two nested for-loops over n elements in lines 2 and 4. The for-loops take $\mathcal{O}(n^2)$ time because they are nested. The list $\mathcal{L}$ can be implemented using an array, making its operations of initialization, update, and lookup take $\mathcal{O}(1)$ time. Therefore, Algo. 1 runs in $\mathcal{O}(n^2)$ time.

Algo. 3 runs Algo. 1 for the $n-1$-instance and takes $\mathcal{O}(n^2)$ time. Then Algo. 3 goes through two nested for-loops over n elements in lines 2 and 4. The two for-loops also take $\mathcal{O}(n^2)$ time. The list $\mathcal{L}$ can also be implemented using an array with $\mathcal{O}(1)$-time operations. As a result, the time complexity of Algo. 3 is $\mathcal{O}(n^2)$. □

V. Performance Evaluation

A. System Implementation & Experimental Settings

We implement a prototype of Sharon in C++ based on Ethereum [33]. The byzantine fault tolerance consensus primitive is implemented efficiently using Boneh-Lynn-Shacham signature [34] of the C++ implementation [35]. Data synchronization among shards is implemented using the information dispersal algorithm [36]. We implement two benchmark sharding protocols, i.e., RapidChain and Monoxide, with the same consensus primitive. Note that the implementations of the three protocols are different only in cross-shard transaction processing. We run the three protocols on 16 Amazon EC2 c4.4xlarge instances (16 vCPUs and 30GB memory) with parameter settings the same as traditional blockchain platforms. The blockchain nodes are pairwise connected by a link of 20Mbps bandwidth and 100ms latency.

This work considers two performance metrics of blockchains. 1) *System throughput:* the number of transactions that can be processed per second; it is an important performance metric because it relates to the maximum affordable workload of a blockchain system. 2) *Transaction processing latency:* the average time from the submission to confirmation of transactions; a high latency limits the applications of a blockchain system, especially those demanding real-time.

We evaluate the influence of four factors on the system performance. 1) *Shard number.* When the shard size is fixed, more nodes and shards can increase the parallelization ratio of transaction processing. However, it will also make it difficult to synchronize data among shards. 2) *Shard size.* Given a fixed number of nodes, a larger shard size will decrease the number of shards, thus decreasing the parallelization ratio of transaction processing. Meanwhile, the transaction processing logic may be simplified, decreasing the transaction processing latency. 3) *Average transaction step.* A shard only keeps a portion of the blockchain state, so processing cross-shard transactions demands the cooperation of multiple shards. The transaction step of a cross-shard transaction refers to the number of related shards. If a transaction is not cross-shard, then its transaction step is 1. A larger average transaction step indicates that the shards need to cooperate more. 4) *Ratio of cross-shard transactions.* The cross-shard ratio refers to the percentage of cross-shard transactions in all transactions. A higher ratio indicates more transactions are cross-shard and need shard cooperation. In existing blockchains, more than 90% transactions are cross-shard [14].

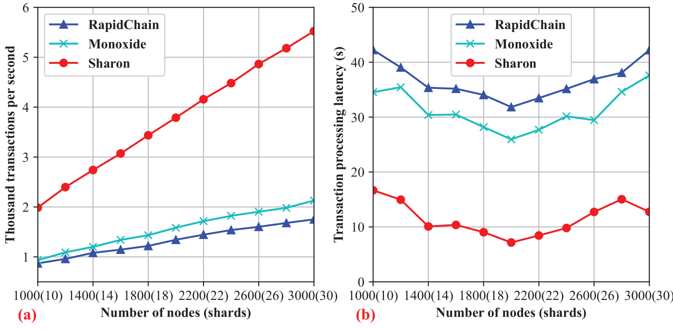

Fig. 5. Influence of number of nodes on system performance.

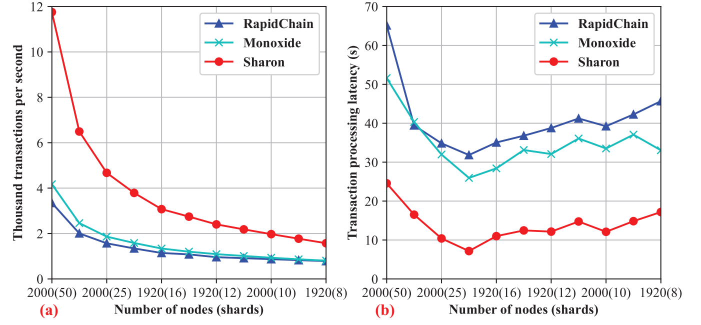

Fig. 6. Influence of shard size on system performance.

We employ real-world Bitcoin and Ethereum transactions as the dataset. Specifically, we collected the Bitcoin data in 2012, containing around 1.9 million transactions, and the Ethereum data in 2016, containing around 14.5 million transactions. The average transaction steps of Ethereum and Bitcoin are 7.48 and 2.93, respectively. We randomly select the transactions from Bitcoin and Ethereum to fit the parameter settings of the average transaction step and ratio of cross-shard transactions.

In the following, we employ the control variable approach to examine the influence of each of the four factors above. We vary a factor each time while fixing the other three and conduct experiments to see how the system throughput and transaction processing latency change according to the varying factor.

B. Influence of Shard Number

In this set of experiments, we fix the shard size, average transaction step, and ratio of cross-shard transactions to be 100, 7.48 (fitting Ethereum), and 90%, respectively, and vary the number of shards from 10 to 30 with a step of 2. The number of nodes will increase from 1000 to 3000. Fig. 5 depicts the experimental results of system throughput and average transaction processing latency.

The system throughput of all three sharding protocols increases linearly with increasing shards. When there are 10 shards, Sharon, Monoxide, and RapidChain can process 1986, 933, and 870 transactions per second, respectively. Sharon achieves 113% and 128% more system throughput than Monoxide and RapidChain. When there are 30 shards, Sharon, Monoxide, and RapidChain can process 5522, 2127, and 1751 transactions per second, respectively. Sharon outperforms Monoxide and RapidChain by 160% and 215%, respectively. Besides high throughput, Sharon enjoys high scalability as well. More specifically, the scalability is measured by $\Delta TPS/\Delta N$, where ΔTPS and ΔN are the changes in system throughput and shard number, respectively. The average scalability of Sharon, Monoxide, and RapidChain can be computed as 0.96, 0.84, and 0.77, respectively. Sharon achieves nearly linear scalability.

Fig. 5(b) shows that the transaction processing latency increases and then decreases with more shards. The reasons are analyzed as follows. On the one hand, more shards improve system throughput, decreasing the average waiting time of transactions. On the other hand, too many shards result in higher complexity of cross-shard transaction processing and higher communication overhead of data synchronization among shards. Sharon, Monoxide, and RapidChain achieve the least average transaction processing latency, i.e., 7.16s, 25.96s, and 31.84s, respectively, when there are 20 shards. Sharon decreases the average transaction processing latency by 72.4% and 77.5% compared to Monoxide and RapidChain.

C. Influence of Shard Size

We fix the number of nodes, average transaction step, and ratio of cross-shard transactions to be 2000, 7.48 (fitting Ethereum), and 90%, respectively, and change the shard size from 40 to 240 with a step of 20. Note that the number of nodes is slightly below 2000 sometimes because the number 2000 is not divisible by some shard sizes, e.g., 60 and 120. Fig. 6 depicts the experimental results of system throughput and average transaction processing latency.

Fig. 6(a) depicts that the system throughput drops dramatically when the shard size increases for all three protocols. The reason is that a larger shard size decreases the number of shards, thus the ability to process transactions in parallel. For example, when the shard size increases from 40 to 60, the number of shards will decrease from $\frac{2000}{40} = 50$ to $\frac{1980}{60} = 33$. With a shard size 40, Sharon, Monoxide, and Sharon can form 50 shards and process 11753, 4173, and 3347 transactions per second. Sharon outperforms Monoxide and Sharon by 182% and 251%, respectively. When each shard consists of 240 nodes, only 8 shards will be formed. In this context, Sharon, Monoxide, and Sharon achieve a system throughput of 1573, 805, and 785 transactions per second, respectively. Sharon outperforms Monoxide by 95.4%, and Monoxide has a similar system throughput compared with RapidChain. Based on Fig. 5(a) and Fig. 6(a), we can infer that the system throughput is primarily affected by the number of shards rather than the number of blockchain nodes.

The influence of shard size on transaction processing latency is shown in Fig. 6(b). The result indicates that the transaction processing latency decreases and then increases with larger shards. When there are 2000 nodes in 50 shards, Sharon, Monoxide, and RapidChain process a transaction on average in 24.57s, 51.61s, and 65.15s, respectively. Sharon saves the transaction processing latency by 52.4% and 62.3% compared to Monoxide and RapidChain, respectively. All three protocols achieve the least transaction processing latency for 2000 nodes in 20 shards. Such a result echos the analysis in Sec. V-B.

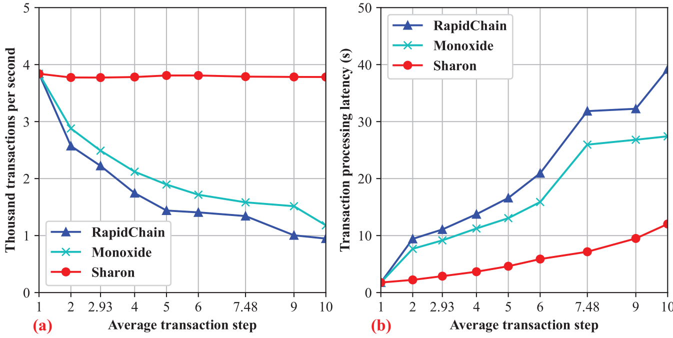

Fig. 7. Influence of average transaction step on system performance.

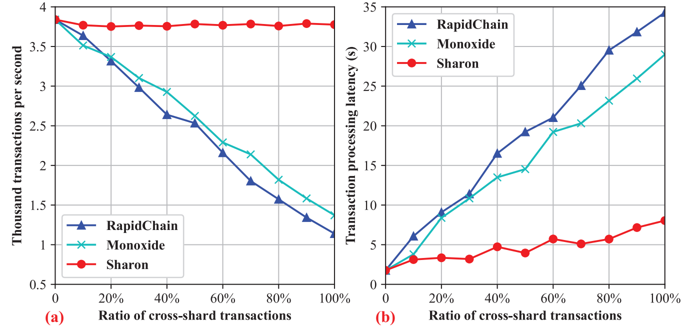

Fig. 8. Influence of ratio of cross-shard transactions on system performance.

D. Influence of Transaction Steps

We fix the number of nodes, shard size, and ratio of cross-shard transactions to be 2000, 100, and 90%, respectively, and change the average transaction step from 1 to 10. We intentionally consider the transaction steps of 2.93 and 7.48, fitting Bitcoin and Ethereum, respectively.

Fig. 7(a) depicts how the system throughput changes according to the average transaction step. When the transaction step is 1, there is no cross-shard transaction. In this case, the three sharding protocols share the same system throughput, i.e., 3839 transactions per second. Sharon outperforms Monoxide and RapidChain with an increasing transaction step. The system throughput of Sharon is not affected by the transaction step because Sharon processes cross-shard transactions via shard merging and rotation regardless of cross-shard transactions. On the contrary, the system throughput of Monoxide and RapidChain decreases when the transaction steps increase due to the performance bottleneck of trustworthy parties to handle cross-shard transactions. Regarding Bitcoin data, when the transactions step is 2.93, Sharon outperforms Monoxide and RapidChain regarding system throughput by 51.5% and 69.7%, respectively. Regarding Ethereum data, when the transaction step is 7.48, Sharon enjoys 139.3% and 182.2% higher system throughput than Monoxide and RapidChain.

The impact of the transaction step on transaction processing latency is shown in Fig. 7(b). Similarly, Sharon, Monoxide, and RapidChain share the same transaction processing latency, i.e., 1.76s, when the transaction step is 1. With an increasing transaction step, all three protocols process transactions slower. However, the transaction step has less impact on Sharon than Monoxide and RapidChain. In particular, when the average transaction step is 2.93 (Bitcoin data), Sharon, Monoxide, and RapidChain process transactions in 2.88s, 9.16s, and 11.07s on average, respectively. Sharon saves 68.6% and 74.0% transaction processing time compared to Monoxide and RapidChain, respectively. When the average transaction step is 7.48 (Ethereum data), Sharon processes a transaction in 7.16s on average and enjoys 72.4% and 77.5% less time compared to Monoxide (25.96s) and RapidChain (31.84s), respectively.

E. Influence of Ratio of Cross-shard Transactions

We fix the number of nodes, shard size, and average transaction step to be 2000, 100, and 7.48 (fitting Ethereum), respectively, and change the ratio of cross-shard transactions from 0% to 100% with a step of 10%.

The influence of the ratio of cross-shard transactions on system throughput is depicted in Fig. 8(a). We can observe that Sharon's system throughput is immune to the ratio of cross-shard transactions because Sharon process cross-shard transactions as normal ones. On the contrary, Monoxide and RapidChain can process much fewer transactions with an increasing ratio of cross-shard transactions. When all transactions are cross-shard, Sharon, Monoxide, and RapidChain process 3775, 1371, and 1140 transactions per second, respectively. In this case, Sharon enjoys 175.3% and 231.1% higher system throughput than Monoxide and RapidChain.

Fig. 8(b) shows the impact of cross-shard transaction ratio on transaction processing latency. Overall, the latency increases for all three protocols with a higher ratio of cross-shard transactions. When all transactions are cross-shard, Sharon can process a transaction in 8.05s on average, compared to 24.29s for Monoxide and 29.00s for RapidChain. In this case, Sharon saves 66.9% and 72.7% transaction processing time compared to Monoxide and RapidChain, respectively.

VI. Conclusion

This work presents Sharon, the first public blockchain sharding protocol that does not rely on trustworthy parties, e.g., clients, special shards, or special nodes. The major innovation of Sharon lies in processing cross-shard transactions through shard merging and rotation rather than transaction division in state-of-the-art protocols. In Sharon, we formally formulate the shard scheduling problem that orders shard meetings minimizing the rounds to confirm cross-shard transactions. We propose a construction algorithm that optimally solves the shard scheduling problem in polynomial time. This work focuses on the bottleneck problem in blockchain sharding, i.e., cross-shard transaction processing, and opens a new direction to solve the problem, i.e., shard merging and rotation.

VII. Acknowledgments

This work was supported by the Research Institute for Artificial Intelligence of Things, The Hong Kong Polytechnic University and the Hong Kong Research Grant Council (RGC) Collaborative Research Fund (CRF) No. C2004-21GF, Research Impact Fund (RIF) No. R5034-18, and Theme-based Research Scheme (TRS) No. T43-513/23-N.

References

[1] M. Xu, Y. Guo, Q. Hu, Z. Xiong, D. Yu, and X. Cheng, “A trustless architecture of blockchain-enabled metaverse,” *High-Confidence Computing*, vol. 3, no. 1, p. 100088, 2023.

[2] J. Zhu, J. Cao, D. Saxena, S. Jiang, and H. Ferradi, “Blockchain-empowered federated learning: Challenges, solutions, and future directions,” *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–31, 2023.

[3] H. Wu, S. Jiang, and J. Cao, “High-efficiency blockchain-based supply chain traceability,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 24, no. 4, pp. 3748–3758, 2023.

[4] S. Jiang, J. Cao, H. Wu, Y. Yang, M. Ma, and J. He, “Blochie: a blockchain-based platform for healthcare information exchange,” in *IEEE International Conference on Smart Computing (SMARTCOMP)*, 2018, pp. 49–56.

[5] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, “A survey of distributed consensus protocols for blockchain networks,” *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020.

[6] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, “Blockchain challenges and opportunities: A survey,” *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.

[7] M. Dotan, Y.-A. Pignolet, S. Schmid, S. Tochner, and A. Zohar, “Survey on blockchain networking: Context, state-of-the-art, challenges,” *ACM Computing Surveys*, vol. 54, no. 5, pp. 107:1–107:34, 2021.

[8] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, “A secure sharding protocol for open blockchains,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 17–30.

[9] T. Nguyen and M. T. Thai, “Denial-of-service vulnerability of hash-based transaction sharding: attack and countermeasure,” *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 641–652, 2023.

[10] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, “Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers,” in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020, pp. 294–308.

[11] J. Hellings and M. Sadoghi, “Byshard: Sharding in a byzantine environment,” *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2230–2243, 2021.

[12] M. Zamani, M. Movahedi, and M. Raykova, “Rapidchain: Scaling blockchain via full sharding,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 931–948.

[13] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding,” in *IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 583–598.

[14] Z. Hong, S. Guo, and P. Li, “Scaling blockchain via layered sharding,” *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 12, pp. 3575–3588, 2022.

[15] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai, “Optchain: optimal transactions placement for scalable blockchain sharding,” in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 525–535.

[16] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, “Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding,” in *IEEE Conference on Computer Communications (INFOCOM)*, 2022, pp. 1968–1977.

[17] B. Wang, J. Jiao, S. Wu, R. Lu, and Q. Zhang, “Age-critical and secure blockchain sharding scheme for satellite-based internet of things,” *IEEE Transactions on Wireless Communications*, vol. 21, no. 11, pp. 9432–9446, 2022.

[18] M. Zhang, J. Li, Z. Chen, H. Chen, and X. Deng, “Cycledger: A scalable and secure parallel protocol for distributed ledger via sharding,” in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 358–367.

[19] Z. Hong, S. Guo, E. Zhou, J. Zhang, W. Chen, J. Liang, J. Zhang, and A. Zomaya, “Prophet: Conflict-free sharding blockchain via byzantine-tolerant deterministic ordering,” in *IEEE Conference on Computer Communications (INFOCOM)*, 2023, pp. 1–10.

[20] E. Wang, J. Cai, Y. Yang, W. Liu, H. Wang, B. Yang, and J. Wu, “Trustworthy and efficient crowdsensed data trading on sharding blockchain,” *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 12, pp. 3547–3561, 2022.

[21] A. Mizrahi and O. Rottenstreich, “Blockchain state sharding with space-aware representations,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 1571–1583, 2021.

[22] J. Wang and H. Wang, “Monoxide: Scale out blockchains with asynchronous consensus zones,” in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 2019, 2019, pp. 95–112.

[23] M. Li, W. Wang, and J. Zhang, “Lb-chain: Load-balanced and low-latency blockchain sharding via account migration,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 10, pp. 2797–2810, 2023.

[24] C. Li, H. Huang, Y. Zhao, X. Peng, R. Yang, Z. Zheng, and S. Guo, “Achieving scalability and load balance across blockchain shards for state sharding,” in *IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2022, pp. 284–294.

[25] K. Mu and X. Wei, “Efshard: Towards efficient state sharding blockchain via flexible and timely state allocation,” *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 2817–2829, 2023.

[26] M. J. Amiri, D. Agrawal, and A. El Abbadi, “Sharper: Sharding permissioned blockchains over network clusters,” in *ACM International Conference on Management of Data (SIGMOD)*, 2021, pp. 76–88.

[27] P. Zheng, Q. Xu, X. Luo, Z. Zheng, W. Zheng, X. Chen, Z. Zhou, Y. Yan, and H. Zhang, “Aeolus: Distributed execution of permissioned blockchain transactions via state sharding,” *IEEE Transactions on Industrial Informatics*, vol. 18, no. 12, pp. 9227–9238, 2022.

[28] X. Qi, “S-store: A scalable data store towards permissioned blockchain sharding,” in *IEEE Conference on Computer Communications (INFOCOM)*, 2022, pp. 1978–1987.

[29] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, “Towards scaling blockchain systems via sharding,” in *ACM International Conference on Management of Data (SIGMOD)*, 2019, pp. 123–140.

[30] Z. Cai, J. Liang, W. Chen, Z. Hong, H.-N. Dai, J. Zhang, and Z. Zheng, “Benzene: Scaling blockchain with cooperation-based sharding,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 639–654, 2023.

[31] R. Pass and E. Shi, “Hybrid consensus: Efficient consensus in the permissionless model,” in *International Symposium on Distributed Computing (DISC)*, vol. 91, 2017, pp. 39:1–39:16.

[32] M. Castro and B. Liskov, “Practical byzantine fault tolerance and proactive recovery,” *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[33] “Ethereum c++ client,” https://github.com/ethereumproject/cpp-ethereum, 2016.

[34] D. Boneh, B. Lynn, and H. Shacham, “Short signatures from the weil pairing,” *Journal of Cryptology*, vol. 17, pp. 297–319, 2004.

[35] “Solidity-compatible bls signatures in modern c++.” https://github.com/skalenetwork/libBLS, 2023.

[36] N. Alon, H. Kaplan, M. Krivelevich, D. Malkhi, and J. Stern, “Scalable secure storage when half the system is faulty,” in *Springer International Colloquium on Automata, Languages and Programming (ICALP)*, 2000, pp. 576–587.