INVITED PAPER *Special Section on Blockchain Systems and Applications*

Load Balancing with In-Protocol/Wallet-Level Account Assignment in Sharded Blockchains*

Naoya OKANAMI[†a)], **Ryuya NAKAMURA**[††b)], ***Nonmembers***, ***and*** **Takashi NISHIDE**[†††], ***Member***

SUMMARY Sharding is a solution to the blockchain scalability problem. A sharded blockchain divides consensus nodes (validators) into groups called shards and processes transactions separately to improve throughput and latency. In this paper, we analyze the rational behavior of users in account/balance model-based sharded blockchains and identify a phenomenon in which accounts (users' wallets and smart contracts) eventually get concentrated in a few shards, making shard loads unfair. This phenomenon leads to bad user experiences, such as delays in transaction inclusions and increased transaction fees. To solve this problem, we propose two load balancing methods in account/balance model-based sharded blockchains. Both methods perform load balancing by periodically reassigning accounts: in the first method, the blockchain protocol itself performs load balancing and in the second method, wallets perform load balancing. We discuss the pros and cons of the two protocols, and apply the protocols to the execution sharding in Ethereum 2.0, an existing sharding design. Further, we analyze by simulation how the protocols behave to confirm that we can observe smaller transaction delays and fees. As a result, we released the simulation program as "Shargri-La," a simulator designed for general-purpose user behavior analysis on the execution sharding in Ethereum 2.0.

key words: *sharding, blockchain, load balancing, game theory, heuristics, simulated annealing*

1. Introduction

Traditional distributed ledgers do not increase transaction processing capacity, no matter how many nodes exist in the network. To improve the scalability of such distributed ledgers, methods such as off-chain protocols and sharded blockchains have been proposed. One of them, sharding, implements parallelization by dividing validators that verify transactions into different groups and processing different transactions in each shard [2]–[5]. Sharding will be used in Ethereum [6] in the future.

There are two blockchain transaction models, the Un-spent Transaction-Output (UTXO) model and the account/balance model. The blockchain with the account/balance model is more compatible with implementing smart contracts. Ethereum, the most popular smart contract platform, uses the account/balance model.

Sharded blockchains with the account/balance model allow users to choose the shard to which their accounts belong freely. Users spend less fee and have less latency when their accounts belong to the same shard as the contracts that frequently communicate with them. Therefore, in reality, it is easier to collect accounts for shards to which popular contracts belong. As a result, the load on the shards is increasingly imbalanced. On the other hand, in a shard, the higher the load is, the more the fee increases. Users do not want to use shards with high fees, so no extreme imbalances will occur. In other words, when users act to improve their user experience (UX), there should be no extreme imbalance that all accounts are concentrated in one shard, and some load balancing will be performed. A user can actually have multiple accounts, but the same phenomenon still occurs.

We expected that, due to these two characteristics, if the user behaves selfishly, the account assignment state converges approximately to a state where all users have no incentive to go to another shard (ϵ-Nash equilibrium). That is, we expected that the sharding protocol already has a mechanism that performs load balancing when the user acts selfishly. In theoretical computer science and distributed systems, the fact that load balancing is performed by users acting selfishly as described above is called *selfish load balancing* [7]–[9]. However, we observed that the load imbalance still occurred among the shards in our simulation.

If the load on the shards is imbalanced, sharding protocols have the following issues.

- The hardware specs required for the validator will be higher than when the load is balanced. This prevents new validators from entering.
- The gas price differs across shards and the fact worsens the UX of cross-shard communications.
- Validators favor an environment, e.g., on Amazon Web Services (AWS), which can efficiently scale in/out.
- The incentive analysis around parameterization of rewards or gas costs might become complicated.

Monoxide is one of the sharded blockchains in the account/balance model [10]. The Monoxide paper mentions a solution to the load concentration in the upper layers, where application operators create one address for each shard and distribute the load.

However, as explained earlier, it is not that there is an imbalance just because there is a heavily loaded ac-

Manuscript received March 10, 2021.
Manuscript revised October 12, 2021.
Manuscript publicized November 29, 2021.

[†]The author is with University of Tsukuba, Tsukuba-shi, 305–8577 Japan.
[††]The author is with LayerX Inc., Tokyo, 103–0004 Japan.
[†††]The author is with the University of Tokyo, Tokyo, 113–8654 Japan.

*A preliminary version of this paper appeared in [1]. We newly showed the effectiveness of wallet-level load balancing through simulation and released the simulation program as "Shargri-La."

a) E-mail: minaminaoy@gmail.com
b) E-mail: ryuya.nakamura@layerx.co.jp

DOI: 10.1587/transinf.2021BCP0003

count. If many users are selfish, the imbalance will be more widespread, and the load will be concentrated in a few shards. If all the shards do not have the same load, the overall UX becomes worse than they do.

To solve the above problem, we propose *in-protocol load balancing*, in which the blockchain protocol periodically reassigns accounts, and *wallet-level load balancing*, in which each user's wallet selects a shard to achieve a global optimum, as a way to reduce shard load imbalance. There are trade-offs between the two methods.

In-protocol load balancing is a new method we propose in which the blockchain protocol optimally assigns accounts based on past transactions. To do in-protocol load balancing, we formulate load balancing as an optimization problem. As a result of the formulation, we show that this problem is NP-hard. Since it is NP-hard, there is no polynomial-time algorithm for finding an exact solution for the load balancing problem. Thus, it is necessary to use an approximation algorithm or heuristics, but it is very computationally expensive to obtain a good solution. Doing the calculation itself on-chain is not worth the cost. Therefore, in-protocol load balancing is done in a competition style where the problem is disclosed and delegated to the outside, and the best solution among the submitted ones is adopted. This provides a better solution than on-chain calculation. We apply this load balancing framework to the execution sharding in Ethereum 2.0 (Eth2) [11] and construct an algorithm that solves the load balancing problem using simulated annealing, which is one of metaheuristics. Also, comparing selfish load balancing with the proposed algorithm, we show that the total transaction fee and total transaction delay can be smaller.

Wallet-level load balancing is a new method we propose in which individual wallets cooperate in the assignments of globally optimal accounts based on past transactions. In the Eth2 execution sharding, we experiment with an environment where wallets are equipped with a *shard switching algorithm* that reduces user transaction fees by moving assets. We then show the effectiveness of the algorithm, which increases the benefit of all users. The source code of the program for the experiments are available on GitHub, and we have also designed a more generalized version of the program and released it as open-source software called "Shargri-La."[†] Shargri-La helps researchers to design and improve the Eth2 execution sharding protocol.

In summary, our contributions are:

- We propose in-protocol load balancing in which the sharded blockchain protocol periodically reassigns accounts (in Sect. 3).
- We apply the in-protocol load balancing to the Eth2 execution sharding, an existing sharding design, and demonstrate that transaction fees and latencies can be reduced (in Sect. 4).
- We propose wallet-level load balancing in which each user's wallet selects a shard to achieve a global optimum in a sharded blockchain (in Sect. 5).
- We also apply the wallet-level load balancing to the Eth2 execution sharding, and demonstrate that the load concentrates on a small number of shards and can be reduced (in Sect. 6).
- We released the simulation program as "Shargri-La," a simulator designed for general-purpose user behavior analysis on the Eth2 execution sharding.

[†]https://github.com/shargri-la/shargri-la

2. Preliminaries

2.1 Task Assignment Problem (TAP)

The following mathematical optimization problem, called *task assignment problem* exists:

> M resources (e.g., CPUs) and N tasks are given. It takes c_i to execute task i. Further, when task i and task j are assigned to different resources, the resources to which task i and task j are assigned cost d_{ij} and d_{ji}, respectively. Each task can be assigned to one resource. Then what is the smallest cost to complete all the tasks?

TAP is a well-known NP-hard problem, and various algorithms for solving it have been proposed [12], [13].

2.2 Cross Shard Transaction

A transaction sent from one shard to another is called a *cross-shard transaction*. A cross-shard transaction has to go through another shard or parent chain and has a higher fee and latency than a single-shard transaction. For example, the problem of how to handle hotel room reservations and train seat reservations atomically is called *train-and-hotel problem*. In sharding, it is a problem of handling contracts in one shard and contracts in another shard atomically.

2.3 Ethereum 2.0 (Eth2)

Ethereum 2.0 is a major upgrade to improve the security and scalability of Ethereum, which introduces proof-of-stake, sharding, etc. The Eth2 execution sharding (formerly known as Eth2 Phase 2) consists of one *beacon chain* and multiple *shard chains*. A shard chain is a sharded blockchain, and a beacon chain is a blockchain that manages the shard chain. The beacon chain mediates cross-shard communications.

Eth2 solves the train-and-hotel problem by introducing an operation called *yank* [14]. A yank is to delete a contract on one shard, issue a transaction receipt, and instantiate the contract on another shard. Then the yank performs some operation on the shard to which it is yanked. For example, to make an atomic reservation in the train-and-hotel problem, we yank a contract to reserve a room for a hotel to a shard that has a contract to reserve a train.

In Eth2, the unit commonly referred to as a block in a blockchain is a *slot*. One slot is 12 seconds. One *epoch* is consisting of 32 slots.

In the current Ethereum, there are two account types, Externally Owned Account (EOA) and Smart Contract, but in Ethereum 2.0, they will be integrated into Account, thus we use the term Account in this paper.

3. In-Protocol Load Balancing

In this section, we propose the blockchain protocol that performs load balancing (i.e., in-protocol load balancing).

The process flow of in-protocol load balancing is as follows.

1. Competition coordinators collect necessary transaction load information of accounts.
2. Coordinators formulate load balancing as an optimization problem.
3. Competition participants calculate good account assignment candidates.
4. Coordinators move accounts based on the new selected assignment.

3.1 Problem Definition

The formulation of the optimization problem varies depending on what metrics the community and users value.

We formulate minimizing the highest load among loads of shards as an optimization problem. Let S be a mapping from account to shard id. Let l_{ij} be the load of a shard that accounts i and j belong to when they belong to the same shard. Further, let l'_{ij} be a load for the shard to which the account i belongs when the accounts i and j belong to different shards. The total load $L_k(S)$ in shard k per unit time is

$$L_k(S) \coloneqq \sum_{i,j,S(i)=k \wedge S(j)=k} l_{ij} + \sum_{i,j,S(i)=k \wedge S(j)\neq k} l'_{ij} \tag{1}$$

There is a correlation between shard fees and shard load. Let the overall load of a shard be L, the fee for processing the load l be $C(L, l)$. In reality, the function C cannot be determined exactly because the fees are proposed by users, and the auction determines which transaction is incorporated into the block by validators.

There are several optimization problems that can be used to improve UX by equalizing the load on all users — for example, minimizing the load on the heaviest shard. Shards with heavy loads have higher transaction fees, and reducing them can significantly reduce overall fees. We formulate this as follows.

$$\text{minimize} \qquad \max_k L_k(S) \tag{2}$$

We name this optimization problem *maximum load minimization problem* (MLMP). TAP is polynomial-time reducible to MLMP with simple formula transformations. If MLMP could be solved in polynomial time, TAP can be solved in polynomial time using that algorithm. Therefore, MLMP is NP-hard.

Good results can also be obtained by minimizing the overall fee. In order to reduce the overall cost, it is necessary to reduce the load on the shard, which is the bottleneck and has the highest load. Thus, the load on all the shards is equalized, and the overall fee is reduced. In addition, the fee is reduced when the number of cross-shard transactions is reduced. Therefore, that optimization is performed so that the number of cross-shard transactions is reduced. This also reduces latency. We formulate this as follows.

$$\text{minimize} \qquad \sum_k C(L_k(S), L_k(S)) \tag{3}$$

This problem is as difficult as MLMP.

3.2 Competition

Since the above optimization problems are difficult, heuristics and approximate algorithms must be used to find a good solution. However, running such heavy processing algorithms on-chain is not worth the cost, so in our design, anyone can submit a solution, and we build a game that rewards the player who submitted the best solution.

For each epoch, the account assignment at the next epoch is determined using the information of the previous epoch. If too old information is used for the past epoch information, load balancing suitable for the transaction in the next epoch is not performed, so it is necessary to use appropriate information of the previous epoch.

If we use transaction load information of all accounts, the amount of information is $O(n^2)$, where n is the number of accounts. In actual operation, the transaction information of the account selected by some algorithm is used for each epoch because of the limited capacity of the beacon chain. For example, there is a method of randomly selecting half of the active accounts or 10% of contracts in descending order of load.

To host a competition, we have nodes that act as competition coordinators. The coordinators formulate and publicize the account assignment as an optimization problem using past epoch transaction load information. The competition players obtain the optimization problem, work on optimization, and submit candidate solutions before the deadline. After the epoch, the coordinators evaluate the candidate solutions and rewards the player who submits the best solution. Rewards are paid as a pool or newly issued coins. Since a malicious player may submit a poorly evaluated solution and put an unnecessary load on the coordinators, the player must pay a fee when submitting the solution. Also, if there are multiple players who have both submitted the best solution, the winner is the one with the fastest submission time.

Coordinators are elected for each epoch. In Ethereum 2.0, a coordinator is a validator who was elected as the first block proposer in an epoch.

3.2.1 Collecting Transaction Data

Every shard has transaction load information for accounts

belonging to that shard. To perform in-protocol load balancing, this information must be passed to the competition coordinators. The method differs depending on the sharding protocol.

In the Eth2 execution sharding, the state of each shard is committed as a Merkle root called *crosslink* [15], [16] that is stored in the beacon chain. The validity and data availability are checked by the shard's validator set.

Since the beacon chain cannot handle transaction load information of all accounts, all shards build data as follows:

1. Every epoch, a shard i randomly samples k contracts $A_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,k}\}$.
2. Accounts not selected by random sampling are merged as a single virtual account as $a_{i,\mathrm{rest}}$. Let R be the unselected set and C_{i_x,j_y} be the cross-shard transaction load from shard i account x to shard j account y.

$$C_{i_{\mathrm{rest}},j_y} = \sum_{x \in R} C_{i_x,j_y} \tag{4}$$

The shard chain sends the information constructed in this way to the beacon chain by crosslink.

3.2.2 Player Algorithms

The player selects themselves the algorithm that they will use. Examples of the algorithm include hill climbing, simulated annealing, and genetic algorithm. Alternatively, players can use a mathematical optimization solver or a combination of the solver and their algorithm. The longer the sharding protocol that introduced in-protocol load balancing operates, the more efficiently the player's algorithm will evolve, and the better the load balancing will be.

3.2.3 Commit-Reveal Scheme

If the solution is submitted, another player may copy the solution and submit an improved solution starting from that solution. If the commit-reveal scheme is adopted, this problem can be solved by releasing the solution and verifying the best solution after the competition is over. That is, the player submits the commitment of (**solution** || **signature**). However, there must be at least one honest player in order for the user to benefit from in-protocol load balancing.

3.3 Security Analysis

The above protocol only changes the state transition rules, so it does not affect the safety, liveness, and validity properties of the blockchain. Also, the consensus protocol and validator validation rules have not changed radically. On the other hand, there is room for validators to selfishly choose a solution to make a profit by external opportunity such as front-running. The analysis of such potential attacking vectors is left as future work.

4. Experiments with In-Protocol Load Balancing

In this section, we show that applying in-protocol load balancing to Ethereum 2.0, modeling users, and simulating them actually lead to reducing shard imbalance, fees and latency. The optimization problem used in the experiment is formulation (3), which minimizes the overall fee.

4.1 Simulation Settings

This subsection describes the user strategy, the algorithm used by the player, and the sharded blockchain model to be simulated.

4.1.1 User Strategy

We use Berenbrink's method [8] to model how a user behaves. Let m be the number of accounts, n be the number of shards and $m \gg n$. In one unit of time, a user moves an account with the following strategy.

Let i be a shard to which the user belongs, and j be a destination shard, and j is selected at random. Let C_i and C_j be the loads of i and j per unit time, respectively. If $C_j < C_i$, we assume that the user moves to shard j with probability $1 - \frac{C_j}{C_i}$. If not, we assume that the user does not move.

When performing in-protocol load balancing, the shard allocation is changed by the protocol, so the cost of moving the shard cannot be ignored and should be taken into consideration in defining user behavior. If C_t is the cost of moving the shard, and the time until the next allocation, that is, epoch time is T, if $C_j + C_t/T < C_i$, then we assume that the user moves to shard j with probability $1 - \frac{C_j + C_t/T}{C_i}$. If not, we assume that the user does not move. As T becomes shorter, C_t/T becomes so large that the user has no incentive to change the shard.

4.1.2 Simulated Annealing Approach

We use the simulated annealing approach for this simulation. Simulated annealing is a generalization of hill climbing and is a metaheuristic used for difficult problems such as NP-hard problems [17]. It is difficult to find the global optimal solution by using hill climbing, but simulated annealing can obtain a value close to the global optimal solution. The algorithm is such that a solution in the neighborhood of the provisional solution is selected at random, and the transition is always made when the score is improved.

The pseudo code is as follows (see Algorithm 1). Let T be the time to execute this algorithm. Neighbor is a function that randomly selects a nearby solution, Score is a function that evaluates the solution, and GetTime is a function that returns how much time has passed since this algorithm was executed. The evaluation value of the score function moves to the better one. Therefore, Score(assignment) is −(whole total fee). The Probability is a function that returns the probability of transition based on the current time

Algorithm 1 Simulated annealing approach

```
t ← 0
while t < T do
    next_assignment ← NEIGHBOR(current_assignment)
    s_c ← SCORE(current_assignment)
    s_n ← SCORE(next_assignment)
    if s_n > s_c then
        current_assignment ← next_assignment
    else
        p ← PROBABILITY(t, s_c, s_n)
        if p > RANDOM() then
            current_assignment ← next_assignment
        end if
    end if
    t ← GETTIME()
end while
```

t, the current_assignment score, and the next_assignment score. The RANDOM function returns a uniform random number between 0 and 1.

Also, no competition is held, i.e., one person submits one solution.

4.1.3 Sharded Blockchain Model

Eth2 execution sharding will generate one block every 12 seconds, with 64 shards planned to be introduced first. Ethereum currently trades 300,000 accounts a day. Simulating all of them requires a lot of computational resources, so this time we set $T = 0.1$ seconds and simulate with 8 shards and 1,000 accounts. Also, the load information of all active accounts is used.

We model how accounts trade with other accounts in a directed graph. The vertex in the graph represents an account, and the directed edge extending from account i to account j represents the average load on account i in all transactions between account i and account j in one unit time (block). This load includes not only the transaction from account i to account j, but also the load at the time of transaction from account j to account i. Since, in reality, transactions are concentrated on very popular accounts such as MakerDAO, we set a parameter called account popularity, so that the more popular the account is, the more easily transactions to that account are sent. The popularity of the account is simply a quadratic function. In other words, the popularity of account i is $\text{popularity}_i = i^2$. However, it is impossible in reality that one account is trading with all other accounts. Therefore, considering the total number of accounts 1000, an account accounts for 5% of all accounts.

We believe this setting is sufficient to show the effect of our in-protocol load balancing.

4.2 Results and Comparisons

As a result of the simulation, the sum of account fees and the number of cross-shard transactions have been reduced. Although this setting is small, the effect of in-protocol load balancing was confirmed.

Table 1 Simulation parameters

Parameter	Value
Number of shards	8
Number of accounts	1000
Load balancing interval	0.1 second
Number of accounts traded by one account	5 %
Number of epochs	1000

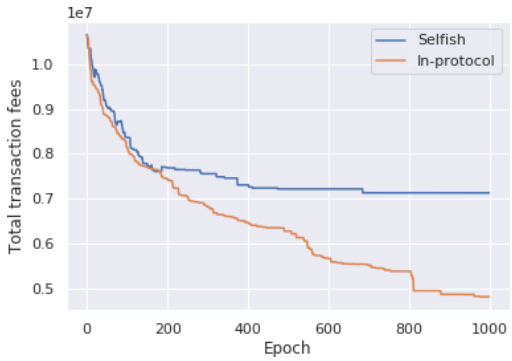

Fig. 1 Decrease of total transaction fees when all accounts selfishly move between shards at each epoch (blue: selfish load balancing, orange: in-protocol load balancing)

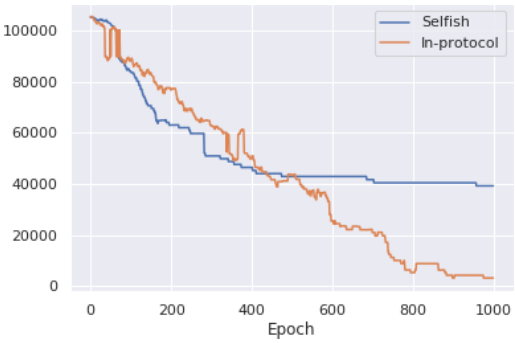

Fig. 2 Decrease of number of cross-shard transactions when all accounts selfishly move between shards at each epoch (blue: selfish load balancing, orange: in-protocol load balancing)

Figures 1 and 2 show selfish load balancing and in-protocol load balancing when all accounts selfishly move among shards at each epoch. Both have converged to specific values, but in-protocol load balancing has reached better values. This is a natural result because selfish load balancing converges to ϵ-Nash equilibrium, while in-protocol load balancing can obtain a Pareto optimal solution.

5. Wallet-Level Load Balancing

In this section, we propose the method in which a wallet selects a shard to be used for the global optimal accounts assignment.

In the real world, users delegate various processes to wallets. For example, Metamask[†], the most popular wallet on Ethereum, offers users the best transaction fee when

[†]https://metamask.io/

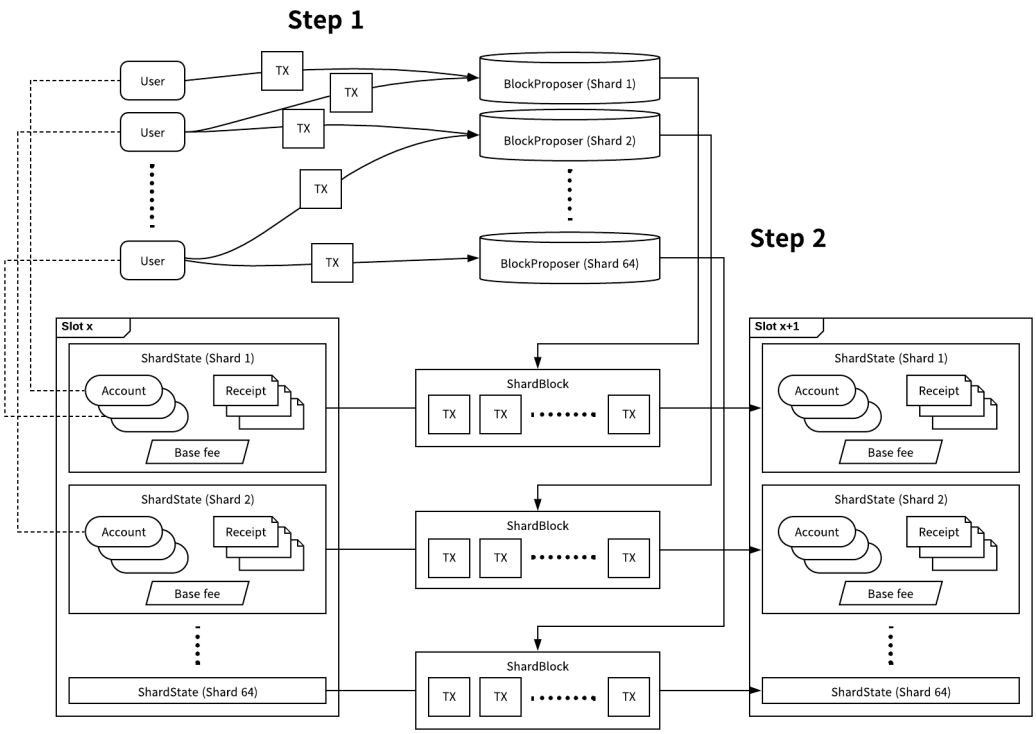

Fig. 3 The simulator's architecture. The simulator performs a discrete event simulation with two steps. The components include user, transaction, block proposer, shard block, shard state, account, receipt, and EIP-1559 base fee, etc.

they send a transaction. Metamask also offers the service called DEX Aggregator, which allows users to exchange crypto assets at the best rates among multiple decentralized exchanges, and users can use it by paying a fee to Metamask.

The fact that the latter aggregator service is viable is evidence that there are many users who value convenience at the expense of the cost of fees. This means that even if a wallet is introduced with an algorithm that does not completely optimize the user themself, but rather improves the whole with a moderate improvement of the individual, the wallet will still be used by the user.

Based on this premise, we implemented a simplified version of the Eth2 execution sharding that simulates an environment where wallets are equipped with shard switching algorithms that reduces transaction fees for users. Users can reduce the number of cross-shard transactions by following the wallet's shard switching algorithm and moving assets to the shards they use most often. We experimented not only with algorithms that simply optimize themselves but also with algorithms that are globally optimal. Those algorithms are not enforced at a protocol level, but they are supposed to be enforced at a wallet level. We then compare the results of the experiments and show that the overall UX is improved when the global optimal shard switching is implemented in the wallet.

In addition, the source code of the program for the experiments we conducted was released on GitHub as the open-source software "Shargri-La." Shargri-La is generic in design and can be reused for various other experiments. Shargri-La is a simulator that aims to analyze user behavior on the Eth2 execution sharding. Shargri-La can help researchers to design and improve the Eth2 execution sharding protocols. Also, Shargri-La is fast and robust because it is implemented in the Rust language. All the parameters used in the experiments in this paper are publicly available, and the experiments can be reproduced by running Shargri-La with those parameters. It is also possible to run Shargri-La with various parameters other than those.

5.1 Simulation Model

We introduce the model of our simulation. The simulator follows a model called discrete event simulation and runs in slots. The blockchain consists of 64 shards. There exist 10,000 users, and the user set is static throughout the simulation. Also, the transaction pricing mechanism of the simulation is EIP-1559 [18].

EIP-1559 introduces a *base fee*, which is a minimum transaction fee that must be paid. This base fee increases or decreases depending on the demand of the transaction. Unlike the traditional first-price auction transaction fee mechanism, users only need to pay this base fee (plus a small bribe), which will mitigate overpayment.

The simulation proceeds with slots. In a nutshell, as shown in Fig. 3, the simulator performs the following two steps for each shard at each slot:

- Step 1: Users create and broadcast transactions.
- Step 2: Block proposers create a shard block.

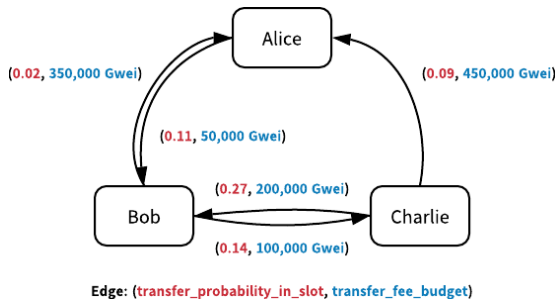

Fig. 4 A small example of UserGraph consisting of three users.

At each slot, users create transactions to perform either one of the following operations:

- Intra-shard transfer
- Cross-shard transfer
- Shard switching (Cross-shard transfer of all the Ether owned by the user)

We define five transaction types. First, we define three transaction types where users transfer Ether to another.

- Transfer: Execute intra-shard transfer of Ether (as we do in the current Ethereum). We define the gas cost as 21,000.
- CreateCrossTransfer: Initiate cross-shard transfer of Ether, producing a receipt. We define the gas cost as 31,785.
- ApplyCrossTransfer: Process an incoming cross-shard transfer by submitting the receipt of CreateCrossTransfer. We define the gas cost as 52,820.

For cross-shard transfers, we assume that there exist proxy contracts specified in the Eth1x64 proposal [19], which implement CreateCrossTransfer and ApplyCrossTransfer as functions.

Next, we define two transaction types where users transfer all their Ether for shard switching.

- CreateCrossTransferAll: Initiate cross-shard transfer of all the Ether controlled by the sender, producing a receipt. We define the gas cost as 31,785.
- ApplyCrossTransferAll: Process an incoming cross-shard transfer by submitting the receipt of CreateCrossTransferAll. We define the gas cost as 52,820.

These two transaction types can be implemented similarly to CreateCrossTransfer and ApplyCrossTransfer. We omit the details of their implementations and gas cost analysis, but these are available on the web.

We formalize users' demand for transfer transactions by a UserGraph, i.e., a directed graph where each node represents a user, each edge represents the demand for the transfer between users. Figure 4 shows a simple example of a UserGraph. The simulator creates transactions based on UserGraph every slot.

We instantiate UserGraph with 10,000 users such that each node has 0 to 15 outgoing edges (determined randomly). The meaning of the two parameters of each edge (from Alice to Bob, for example) are:

- `transfer_probability_in_slot`: The probability that Alice initiates an intra/cross-shard transfer of Ether to Bob in a slot. We set `transfer_probability_in_slot` randomly such that the transactions created in each slot are, on average, about as much as the total capacity (i.e., the sum of block gas limits of each shard).
- `transfer_fee_budget`: The maximum fee in total that Alice is willing to pay to complete the transfer. Alice calculates the fee cap of the transaction based on `transfer_fee_budget`.

5.2 Shard Switching Algorithms

We describe the concrete process of shard switching. We assume users perform the wallet's cost-reducing function once every 100 slots on average, which is modeled as a lottery in the simulation. Then, the wallet of a user u, who is currently active in shard i_{now}, executes the following.

First, for each shard i, calculate the expected transaction fee f_i for the next 100 slots in the case where u moves to shard i. The wallet has the estimation of the probabilities that u makes a transfer to other users in a slot. This is equivalent to the edges directed from u's node in UserGraph. Based on those probabilities, the wallet estimates the number of Transfer, CreateCrossTransfer, and ApplyCrossTransfer transactions on each shard they will create. The transfers to receivers active in shard i are considered as intra-shard, and the other transfers are considered as cross-shard. Based on those numbers of transactions of each type, the wallet calculates f_i. f_i is the sum of (number of expected transactions) × (transaction fee) of Transfer, CreateCrossTransfer, and ApplyCrossTransfer plus (transaction fee) of shard switching. We assume the wallet knows the current base fee of each shard. Although the base fee can change over time, the current base fee is used as the expected base fee in this calculation. I.e., (transaction fee) = (gas cost) × (current base fee).

Then, the wallet selects the shard i_{rec} to recommend the user to move the Ether. We define two recommendation algorithms:

- **The minimum selection algorithm.** Select the shard with the minimum expected transaction fee, i.e., $i_{\text{rec}} = \arg\min_{1 \le i \le 64} f_i$.
- **The weighted random selection algorithm.** Let r_i be the expected reduction of the transaction fee if u switches to shard i, i.e., $r_i = f_{i_{\text{now}}} - f_i$. Select the shard randomly using the expected reduction as weights, i.e., the probability of $i_{\text{rec}} = i$ is $r_i / \sum_{j \in \{j' | r_{j'} > 0\}} r_j$.

6. Experiments with Wallet-Level Load Balancing

In this section, we show that applying wallet-level load balancing to Ethereum 2.0, modeling users, and simulating them actually lead to reducing shard imbalance, fees and latency.

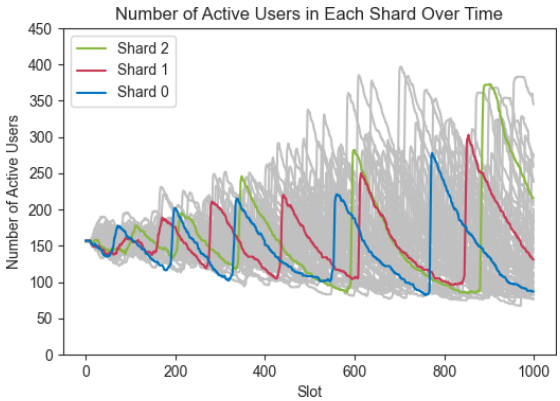

Fig. 5 The number of active users in each shard over time in Experiment 1. Note that we highlight the lines of only three shards (shard 0, 1, and 2) in the figures.

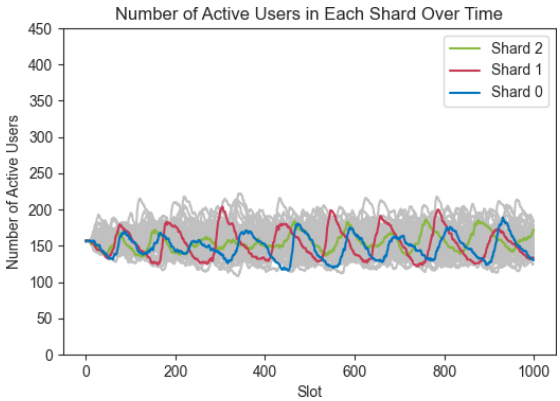

Fig. 6 The number of active users in each shard over time in Experiment 2. The scale of the axis is the same as in Fig. 5.

We conducted experiments in two different user set settings.

1. Majority (67%) of users execute the minimum selection algorithm to determine whether to perform shard switching, and the rest of the users do not execute the minimum selection algorithm (thus do not perform shard switching).
2. Majority (67%) of users execute the weighted random selection algorithm to determine whether to perform shard switching, and the rest of the users do not execute the minimum selection algorithm (thus do not perform shard switching).

6.1 Experiment 1: The Minimum Selection Algorithm

We simulated the case where a majority (67%) of users adopt the minimum selection algorithm to move to the shard with the minimum expected transaction fee.

Figure 5 shows that users are flooding the shard with the minimum fee. We also observed that the rapid increase in demand caused transactions to accumulate in the EIP-1559 transaction pool, which would not occur under normal situations.

6.2 Experiment 2: The Weighted Random Selection Algorithm

Since the minimum selection algorithm becomes ineffective if more and more users adopt it, we devise the recommendation algorithm that selects a shard randomly using the expected reduction as weights. We hypothesize that by dispersing the destination shards of the switchers based on the weights, we can avoid the congestions in shards while users still enjoy the benefit of reducing the overhead of cross-shard transactions.

We simulated the case where a majority (67%) of users adopt the weighted random selection.

Figure 6 shows that the congestions in shards are mitigated compared to the previous experiment (Fig. 5). This algorithm solved the sudden congestion and transaction clogging in a particular shard. We also observed a decrease in fees for users who performed shard switching compared to users who did not perform shard switching.

7. Comparison of Proposed Load Balancing Methods

In this section, we describe the pros and cons of in-protocol load balancing and wallet-level load balancing.

7.1 Protocol Complexity

In-protocol load balancing requires modifying the blockchain protocol (i.e., hard fork) to implement the load balancing function. On the other hand, Wallet-level load balancing does not require any changes to the blockchain protocol and only needs to be implemented by each wallet, resulting in lower infrastructure costs.

7.2 User Cooperation

In-protocol load balancing is enforced by the blockchain protocol, and thus users are forced to cooperate in load balancing. On the other hand, even if wallet-level load balancing is implemented in a wallet used by many users (e.g. Metamask), very selfish users such as high-frequency trading bots and front-running bots can avoid cooperating in the load balancing, and thus the load balancing is less effective than in-protocol load balancing.

8. Discussions

In this section, we discuss the cautions regarding the real-world implementations of the proposed methods and the room for improving the accuracy of the experiments of the proposed methods.

8.1 Other Algorithms

In this paper, simulated annealing is used for in-protocol

load balancing, but it may be possible to find a more efficient solution by using another heuristic algorithm or by using mixed-integer optimization with a mathematical optimization solver. The algorithm actually used for in-protocol sharding will be refined as players compete. What is important is not the efficiency of the algorithm used, but the use of our proposed in-protocol load balancing can improve total fees and latency better than selfish load balancing.

8.2 Simulation Settings

The simulation settings in this paper have room to be improved in terms of experimentation. A more strict simulation may show that in-protocol load balancing is more effective. It may also indicate cases where in-protocol load balancing is not effective, as well as cases where it is effective. It is desirable that we deal with even larger data to see whether the results obtained by in-protocol load balancing can be worth the cost.

In experiments of wallet-level load balancing, although users put their Ether in only one shard in this simulation, users can keep their Ether in multiple shards in reality. If a user regularly makes transfers to specific accounts in different shards, they will put some portion of their Ether in each of those shards.

9. Conclusion

We confirmed the imbalance phenomenon by modeling and simulating users although we expected that a few shard accounts would be concentrated by acting selfishly in sharded blockchains with the account/balance model. We also showed that the shard load imbalance worsens UX due to higher transaction fees and increased latency. To solve this problem, we proposed the load balancing framework for sharded blockchains. This framework achieves in-protocol load balancing by taking advantage of the incentive to change shards by changing account assignments frequently. We also proposed the method for efficiently obtaining good account assignments in the competition format. Although small, our simulations showed that transaction fees and latency are lower than the selfish load balancing that occurs when users act on their own with this in-protocol load balancing. Furthermore, we showed by simulation that wallet-level load balancing is effective for Eth2 execution sharding. We released the program used in the experiments "ShargriLa," an Eth2 execution sharding simulator designed for general-purpose user behavior analysis.

Acknowledgments

This work was supported in part by JSPS KAKENHI Grant Number 20K11807 and Exploratory IT Human Resources Project (MITOU Program) 2020 of Information-technology Promotion Agency (IPA).

References

[1] N. Okanami, R. Nakamura, and T. Nishide, "Load balancing for sharded blockchains," Financial Cryptography and Data Security, pp.512–524, Springer International Publishing, 2020.
[2] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, New York, New York, USA, pp.254–269, ACM Press, 2016.
[3] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding," Proceedings - IEEE Symposium on Security and Privacy, pp.583–598, Institute of Electrical and Electronics Engineers Inc., July 2018.
[4] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A Sharded Smart Contracts Platform," Internet Society, Feb. 2018.
[5] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling blockchain via full sharding," Proc. ACM Conference on Computer and Communications Security, pp.931–948, Association for Computing Machinery, Oct. 2018.
[6] V. Buterin, "A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM," Tech. Rep.
[7] S. Suri, C.D. Tóth, and Y. Zhou, "Selfish load balancing and atomic congestion games," Annual ACM Symposium on Parallel Algorithms and Architectures, vol.16, pp.188–195, 2004.
[8] P. Berenbrink, T. Friedetzky, L. Ann Goldberg, P.W. Goldberg, Z. Hu, and R. Martin, "Distributed selfish load balancing," SIAM Journal on Computing, vol.37, no.4, pp.1163–1181, 2007.
[9] C.P. Adolphs and P. Berenbrink, "Distributed selfish load balancing with weights and speeds," Proc. Annual ACM Symposium on Principles of Distributed Computing, pp.135–144, 2012.
[10] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pp.95–112, 2019.
[11] G. Wang, Z.J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," AFT 2019 - Proc. 1st ACM Conference on Advances in Financial Technologies, pp.41–61, 2019.
[12] A. Billionnet, M.C. Costa, and A. Sutter, "An Efficient Algorithm for a Task Allocation Problem," Journal of the AwocLatlon for Computing Machinery, VOI, vol.39, no.3, pp.502–518, 1992.
[13] V. Chaudhary and J.K. Aggarwal, "A Generalized Scheme for Mapping Parallel Algorithms," IEEE Trans. Parallel Distrib. Syst., vol.4, no.3, pp.328–346, 1993.
[14] V. Buterin, "Cross-shard contract yanking - Sharding - Ethereum Research."
[15] V. Buterin, "Cross-links between main chain and shards - Sharding - Ethereum Research."
[16] V. Buterin, "Serenity Design Rationale."
[17] K.A. Dowsland and J.M. Thompson, "Simulated annealing," Handbook of Natural Computing, vol.4-4, pp.1623–1655, 2012.
[18] "EIP-1559: Fee market change for ETH 1.0 chain." https://github.com/ethereum/EIPS/eip-1559.md. Accessed: 2021-3-10.
[19] "Eth1x64 variant 1 "apostille"." https://ethresear.ch/t/eth1x64-variant-1-apostille/7365, May 2020. Accessed: 2021-3-10.

Naoya Okanami received B.S. degree from University of Tsukuba in 2020. From 2019 to 2021, he had worked at LayerX Inc. He is currently a student in the Master's program, Risk and Resilience Engineering, University of Tsukuba. He was selected for the IPA Exploratory IT Human Resource Development Project in 2020. His research is in the areas of blockchain and information security.

Ryuya Nakamura is an executive officer at LayerX Inc. His research focuses on the development of technologies to solve the security and privacy issues of data in finance, government, and Internet voting. Also, he has been worked on the security of real-world blockchain consensus protocols, and discovered several vulnerabilities in the Ethereum 2.0 protocol, and contributed to the development of the specification. He was the first Japan-based team to win a grant from the Ethereum Foundation. He was selected for the IPA Exploratory IT Human Resource Development Project in 2020.

Takashi Nishide received B.S. degree from the University of Tokyo in 1997, M.S. degree from the University of Southern California in 2003, and Dr.E. degree from the University of Electro-Communications in 2008. From 1997 to 2009, he had worked at Hitachi Software Engineering Co., Ltd. developing security products. From 2009 to 2013, he had been an assistant professor at Kyushu University and from 2013 he is an associate professor at University of Tsukuba. His research is in the areas of cryptography and information security.