GARET: improving throughput using gas consumption-aware relocation in Ethereum sharding environments

Sangyeon Woo, Jeho Song, Sanghyeok Kim, Youngjae Kim & Sungyong Park

ONLINE FIRST

Cluster Computing
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-020-03087-1

Springer

Cluster Computing
https://doi.org/10.1007/s10586-020-03087-1

GARET: improving throughput using gas consumption-aware relocation in Ethereum sharding environments

Sangyeon Woo[1] · Jeho Song[1] · Sanghyeok Kim[1] · Youngjae Kim[1] · Sungyong Park[1]

Received: 1 December 2019 / Revised: 31 January 2020 / Accepted: 9 March 2020

Abstract
Advances in blockchain technology have made a significant impact on a wide range of research areas due to the features such as transparency, decentralization and traceability. With the explosive growth of blockchain transactions, there has been a growing interest in improving the scalability of blockchain network. Sharding is one of the methods to solve this scalability problem by partitioning the network into several shards so that each shard can process the transactions in parallel. Ethereum places each transaction statically on a shard based on its account address without considering the complexity of the transaction or the load generated by the transaction. This causes the transaction utilization on each shard to be uneven, which makes the transaction throughput of the network decrease. This paper formulates this problem as a multi-dimensional knapsack problem (MKP) and proposes a heuristic algorithm called GARET. The GARET dynamically relocates the transaction load of each shard based on *gas* consumption to maximize the transaction throughput. Ethereum *gas* is a unit that represents the amount of computational effort needed to execute operations in a transaction. Benchmarking results show that GARET outperforms existing techniques by up to 12% in transaction throughput and decreases the makespan of transaction latency by about 74% under various conditions. It is also shown that the relocation overhead is minimal and does not affect the overall performance.

Keywords Blockchain · Ethereum · Sharding · Scalability · Relocation

1 Introduction

Blockchain is a peer-to-peer (P2P) based distributed ledger technology that ensures integrity and reliability without an authorized third party's involvement. Although the blockchain was originally developed as part of Bitcoin [22], it has recently been drawing much attention as an innovative technology that can support a variety of fields such as health-care [21], internet of things (IOT) [15] or medical data management [6].

Despite the worldwide interest in the blockchain, applying this technology to various areas is sometimes limited due to the scalability problem [16]. When the number of transactions increases, the transaction per second (TPS) of the blockchain decreases severely because the time for sharing a block or reaching a consensus is delayed. For example, when one of the most well-known Ethereum-based games called CryptoKitties [12] was released, the amount of pending transactions was sharply increased [3] and finally broke down the Ethereum network. While simply increasing the block size or shortening the interval

A preliminary version of this article [17] was presented at the 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W), Umea, Sweden, June, 2019.

✉ Sungyong Park
parksy@sogang.ac.kr

Sangyeon Woo
tkddus121@sogang.ac.kr

Jeho Song
oidwin@sogang.ac.kr

Sanghyeok Kim
sangh228@sogang.ac.kr

Youngjae Kim
youkim@sogang.ac.kr

[1] Department of Computer Science and Engineering, Sogang University, 35, Baekbeom-ro, Mapo-gu, Seoul, Republic of Korea

Published online: 18 March 2020

to create a block can improve the transaction throughput, it may also weaken the network security as the newly created blocks cannot possibly be delivered to all the nodes in the blockchain network [29].

To solve this scalability problem effectively, several approaches have been proposed. Those include the mechanisms using off-chain payment [23], using byzantine fault tolerance (BFT) consensus instead of existing proof-of-work (PoW) algorithm [26], using a permissioned (or private) blockchain that requires only authorized clients to participate in the consensus process [11], and using sharding [18, 28].

Sharding is a method to increase the transaction throughput of the network by partitioning blockchain into several pieces called shards and allowing each shard to process the transactions in parallel. This mitigates the amount of data transferred among the nodes as the size of data within the shard is smaller than that in the entire blockchain. Consequently, sharding can increase the transaction throughput. Whereas sharding seems to be a viable solution for scaling blockchain, it also creates other challenging issues that needs to be solved such as how to allocate each transaction to a specific shard, how to reach consensus between shards, how to access or synchronize the states in a shard with other shards, etc.

Among those important challenging issues, this paper attempts to address the problem of transaction allocation to a shard in Ethereum sharding environments. Ethereum [9] is a distributed, permissionless (or public) blockchain platform that can run smart contracts. Due to its decentralized, secure, and flexible nature, Ethereum has been widely used as a platform for initial coin offering (ICO). In Ethereum sharding, the client's accounts are statically partitioned based on the account address [1] and distributed to each shard. This scheme is referred as static address-based placement method (S-ACC) throughout this paper.

The S-ACC causes transaction load imbalance between shards as it does not consider the complexity of transactions and the load generated from the transactions. When such imbalance occurs in Ethereum sharding environments, the number of pending transactions may increase, which in turn lowers transaction throughput and increases the makespan of transaction latency.

This paper formulates this problem as a multi-dimensional knapsack problem (MKP) and proposes a *gas* consumption-aware account relocation mechanism in the Ethereum called GARET. The GARET is dynamic in the sense that the accounts between shards can be relocated to improve the transaction throughput and minimize the makespan of transaction latency by periodically checking the *gas* consumption in each shard. To summarize, this paper makes the following specific contributions:

- The method proposed in this paper uses the *gas* consumption as an indicator for the complexity of transaction load that changes over time. The *gas* in the Ethereum is a unit of fee that is paid for the computation resources consumed to execute operations in a transaction. Therefore, the amount of *gas* consumption in a transaction represents the transaction load more accurately than the number of transactions since each transaction may have different complexity.
- The GARET consists of two sub-components: transaction load prediction algorithm and account relocation algorithm. Instead of using transient load in the Ethereum, the amount of *gas* consumption requesting to the account group is predicted. Based on the prediction, the account relocation algorithm dynamically relocates the account groups of each shard. This balances the transaction load of each shard and minimizes the makespan of the transaction latency.
- To evaluate the performance of the proposed technique, we have conducted various experiments with the OMNeT++ 5.4.1 simulator [25]. Because of the difficulty of using real workloads, we generated a variety of synthetic workloads that mimic the real workload as much as possible by analyzing the real trace log from the Etherscan [2]. The performance results showed that the transaction throughput is improved by up to 12%, while the makespan of the transaction latency is decreased by up to 74%.
- To analyze the overhead incurred by account relocation, we developed another version of account relocation algorithm called GARET-PS (GARET with partial shuffling). We also showed that the GARET-FS (GARET with full shuffling) still outperforms the GARET-PS although the GARET-PS reduces the relocation count under relatively little traffic conditions.

The rest of the paper is organized as follows. Section 2 presents an overview of Ethereum sharding and the motivation behind the proposed mechanism. Section 3 defines the problem. Section 4 discusses the design issues of GARET and Sect. 5 evaluates the performance of the proposed method. Chapter 6 introduces previous research approaches to scale blockchain network and Chapter 7 concludes the paper.

2 Background and motivation

This section briefly introduces the Ethereum and its sharding mechanism, and discusses the motivation for the proposed approach.

2.1 Overview of Ethereum

Ethereum is a permissionless blockchain platform for creating and executing *Dapps* through the smart contract [9]. Smart contract is an application executed on all participating blockchain nodes, which ensures integrity and reliability of its execution results.

Ethereum provides users with the *Turing-complete* programming language and Ethereum virtual machine (EVM) to enable them to create various smart contracts. Ethereum users create a smart contract using the Turing-complete programming language. The smart contracts created by users are compiled into the bytecode to be deployed in the blockchain network. As a deployed smart contract is considered as an account, the contract can be executed in the similar way that users send a transaction to the account. The EVM is a 256-bit virtual machine (VM) that can execute the deployed smart contract. All nodes can execute the deployed smart contract by using the EVM. Therefore, based on the information in the smart contracts that are deployed through blockchain, all nodes can execute all smart contracts and validate the results executed by other nodes.

Due to the inherent scalability limitation on the permissionless blockchain networks, Ethereum proposed a mechanism called sharding. The Ethereum sharding partitions the Ethereum network into several shards so that each shard executes the transactions in parallel as shown in Fig. 1. Each shard contains a *collation* chain (collation is a block in a shard), which is a data structure to process and store transactions. The validator in each shard is responsible for validating all transactions within the shard and generating a collation at appropriate intervals. Therefore, one of the challenging issues in Ethereum sharding is how to allocate transactions to each shard. Ethereum assigns each account to a shard statically according to its address prefix (we call this as S-ACC). This leads to a load imbalance problem as the complexity of each transaction and the load condition in each shard are not properly considered.

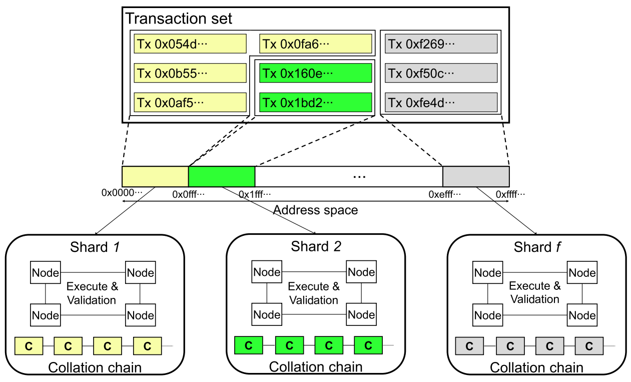


Fig. 1 Overview of Ethereum Sharding

2.2 Motivation

2.2.1 Transaction complexity

Ethereum uses a unit called *gas* that represents the amount of computational effort needed to execute operations in a transaction. For example, if we need to run transactions that require more execution cycles, more *gas* consumption is expected. As shown in Table 1, the amount of *gas* required to execute a transaction is calculated based on the *gas* consumption defined for the operation code executed in the transaction.

In order to confirm that transactions have different *gas* consumption, we analyzed the *gas* consumption trace of about 1400 transactions collected from the real Ethereum network. As shown in Fig. 2, the *gas* consumption of transactions varies from a minimum of 13,678 to a maximum of 2,315,546. Considering that a *gas* limit in a single block is set in advance, the number of transactions that can be included in a single block largely depends on the amount of total *gas* consumption from all transactions. Therefore, if we allocate transactions to each shard statically based on the account address prefix or simply based on the number of transactions, the total number of transactions generated at each interval can possibly be decreased, which lowers the transaction throughput (i.e.,

Table 1 Gas consumption by operation [27]

Name	Value	Description
G_{base}	2 Gas	Gas for {*ADDRESS, ORIGIN, CALLER...*}
$G_{verylow}$	3 Gas	Gas for {*ADD, SUB, NOT...*}
G_{low}	5 Gas	Gas for {*MUL, DIV, SDIV...*}
G_{mid}	8 Gas	Gas for {*ADDMOD, MULMOD, JUMP...*}
G_{high}	10 Gas	Gas for {*JUMPI*}
$G_{extcode}$	700 Gas	Gas for {*EXTCODESIZE*}
$G_{balance}$	400 Gas	Gas for {*BALANCE*}

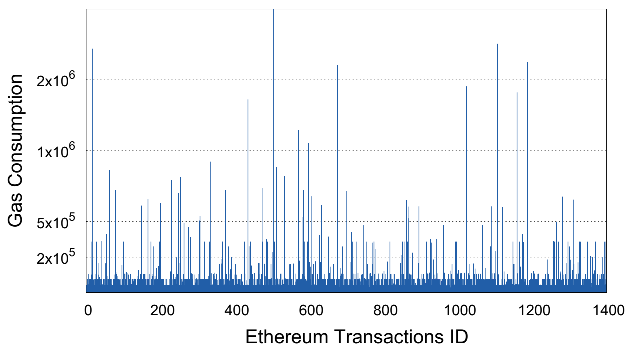


Fig. 2 Variance of gas consumption

low TPS). For this, allocating transactions to each shard based on the *gas* consumption can lead to performance improvement.

2.2.2 Effects of gas consumption imbalance

We conducted a preliminary experiment with the OMNeT++ 5.4.1 simulator to identify the load imbalance problem between shards based on the S-ACC. We measured the average collation utilization of each shard during the 50 collation cycle, assuming an environment where there are 20 shards. The collation utilization of a shard denotes the actual *gas* consumption versus the maximum *gas* consumption that can be included in the collation (i.e., *gas* consumption/*gas* limit).

The transaction load used in this experiment was generated using the workloads summarized in Table 2. Figure 3 shows the average collation utilization of each shard. As shown in Fig. 3, only 25% of all 20 shards show the maximum utilization, while other 25% shards were under 60% utilization. This indicates that placing accounts on each shard regardless of the complexity of transaction load may cause imbalance in the transaction load. When the imbalance in the transaction loads between shards increases, it is possible that the number of transactions pending at a particular shard is abnormally larger than those at other shards. This means that the pending transactions are more likely to be included at the next collation cycle since the gas limit that can be included in a collation is defined. This leads to performance degradation.

Table 2 Summary of Ethereum workloads (1400 transactions)

	Type	Value (*gas*)	Ratio (%)
Gas	G-Low	20742	30
	G-Medium	51857	40
	G-High	105153	30
	Type	Volume ratio (%)	Ratio (%)
Transaction	T-Small	20	80
	T-Large	80	20

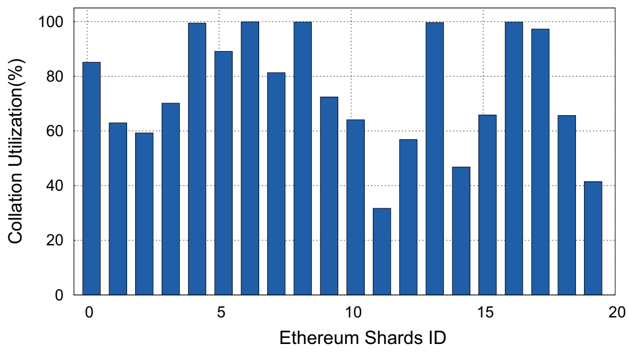


Fig. 3 Collation utilization of 20 shards

3 Problem definition

The basic system model assumed in GARET conforms to the Ethereum sharding architecture presented in Fig. 1, where the number of shards is fixed. As Ethereum sharding uses the proof-of-stake (PoS) consensus algorithm, every shard has the same collation cycle. Therefore, we can apply GARET to all shards simultaneously. We also assume that the size of collation is limited according to the amount of *gas* consumption and no malicious attack occurs on the sharding network. All accounts belong to one account group that each account group includes the same number of accounts.

Let us assume that A_i refer to the *i*-th account group and S_i refer to the *i*-th shard, respectively. Also, assume that N_{ag} refers to the number of account group and the number of shards composing the sharding network is defined as N_{sh}. Unlike traditional systems, which create and manage user accounts in the central server, the GARET manages the accounts by dividing their addresses into N_{ag} groups. Moreover, $Gas^{used}_{S_i}$ and U_{S_i} are the amount of *gas* consumption per shard and the collation utilization per shard, respectively.

Figure 4 shows a pictorial representation of the problem in this study. As shown in Fig. 4, the problem is to assign account groups to N_{sh} shards so that the sum of U_{S_i} is maximized, while the sum of $Gas^{used}_{S_i}$ does not exceed the Gas^{limit} assigned to each shard. This is a multi-dimensional knapsack problem (MKP) [13] and can be formulated with the following Eqs. from 1 to 4.

$$Gas^{used}_{S_i} = \sum Gas^{used}_{A_j}, \quad for\, all\, A_j \in S_i \tag{1}$$

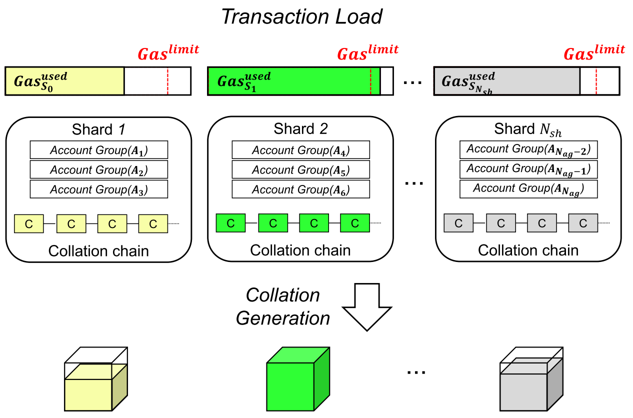


Fig. 4 Problem definition

$$U_{S_i} = \frac{Gas_{S_i}^{used}}{Gas^{limit}}, \quad i = 1, \ldots, N_{sh} \tag{2}$$

$$\text{Maximize} \quad \sum_{i=1}^{N_{sh}} U_{S_i} \tag{3}$$

$$\text{subject to} \quad Gas_{S_i}^{used} \leq Gas^{limit}, \quad i = 1, \ldots, N_{sh} \tag{4}$$

Since this problem is considered as NP-complete [13], we propose a heuristic algorithm to solve this problem in the following section.

4 Design of GARET

This section presents an overview of the GARET architecture and discusses its components in detail.

4.1 Overview

Figure 5 depicts the overall architecture of GARET. In the proposed architecture, we assume that each shard has a validator connected both to the main blockchain and the collation chain, and the GARET is installed within the validator. The GARET consists of two algorithms: transaction load prediction algorithm and account relocation algorithm. The GARET initially places N_{ag}/N_{sh} account groups in each shard so that the same number of accounts can be located in each shard. When the number of generated collations reaches to the relocation cycle, the GARET runs the transaction load prediction algorithm for each account group based on the *gas* consumption. With the predicted load, the GARET invokes the account relocation algorithm to determine whether the accounts should be relocated or not. If relocation is required, the consensus protocol is used to synchronize the outcome from validators and the account state table is updated accordingly.

For example, assume that the time interval between two consecutive collations is C^T and the number of collations generated in one shard at every relocation cycle is N_{col}. Then, P^T, the relocation cycle of GARET, is defined as Eq. 5. Therefore, if N_{col} is 5, the GARET is executed when 5 collations are generated in one shard.

$$P^T = N_{col} \times C^T \tag{5}$$

4.2 Transaction load prediction algorithm

The transaction load prediction algorithm predicts the transaction load that will occur in the future, based on the *gas* consumption of the previous transactions. This algorithm has the following features. First, it predicts the transaction load with the *gas* consumption instead of the number of transactions. This is because the transactions in the Ethereum are much more complex and the number of transactions cannot properly reflect the complexity of transactions. Second, it predicts the future transaction load using the *gas* consumption that has been processed in one cycle before the last cycle. This allows the Ethereum to have enough time to prepare and validate the collations that were newly relocated in each shard. When the collations generated in the previous cycle are invalidated, the validation may not be finalized. Therefore, if the transaction load is predicted with the gas consumption obtained from the previous cycle, it is possible that incorrect gas consumption can be used for the prediction.

Let us assume that $Gas_{i,j}^{used}$ is the *gas* consumption of the i-th account group in the j-th collation and W_j is the weight value for the j-th collation. Then, $Gas_{A_i}^{pred}$, the predicted *gas* consumption of the i-th account group defined in Eq. 6 is the sum of product of $Gas_{i,j}^{used}$ and W_j , where j is increased from 1 to N_{col}. Moreover, the W_j in Eq. 7 is defined such

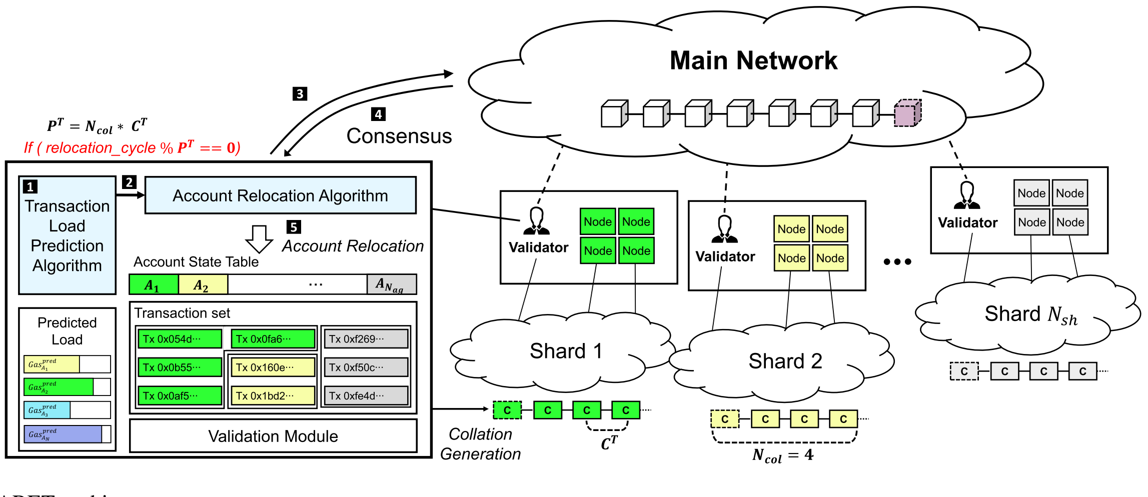
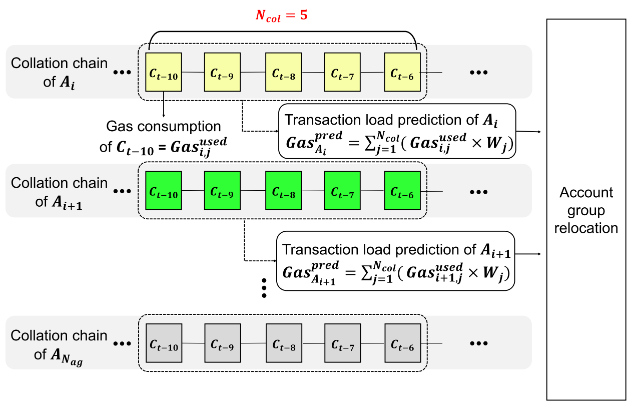


Fig. 5 GARET architecture



$N_{col} = 5$
Collation chain of A_i
C_{t-10} C_{t-9} C_{t-8} C_{t-7} C_{t-6}
Gas consumption of C_{t-10} = $Gas^{used}_{i,j}$
Transaction load prediction of A_i
$Gas^{pred}_{A_i} = \sum_{j=1}^{N_{col}}(Gas^{used}_{i,j} \times W_j)$
Collation chain of A_{i+1}
Transaction load prediction of A_{i+1}
$Gas^{pred}_{A_{i+1}} = \sum_{j=1}^{N_{col}}(Gas^{used}_{i+1,j} \times W_j)$
Collation chain of $A_{N_{ag}}$
Account group relocation

Fig. 6 Transaction load prediction algorithm

that the nearest past gas consumption value has more weight. For example, if N_{col} is 5, each W_j would be 1/15, 2/15, 3/15, 4/15, 5/15, respectively when j is increased from 1 to 5. Figure 6 shows the detailed steps of $Gas^{pred}_{A_i}$ calculation in each account group when the N_{col} is 5.

$$Gas^{pred}_{A_i} = \sum\nolimits_{j=1}^{N_{col}} (Gas^{used}_{i,j} \times W_j) \tag{6}$$

$$W_j = \frac{j}{\sum_{k=1}^{N_{col}}(k)} = \frac{2 \times j}{N_{col}(N_{col}+1)} \tag{7}$$

4.3 Account relocation algorithm

The account relocation algorithm uses the predicted transaction load obtained from the transaction load prediction algorithm described in Sect. 4.2. Figure 7 shows an overall flow of the account group relocation algorithm.

The account relocation algorithm is composed of two steps as shown in Algorithm 1. First, it creates a priority queue based on the information from the N_{ag} account groups, and puts an account group with the biggest transaction load to come first. Then, it selects an account group from the queue and relocates it to a shard with the minimum *gas* consumption. That is, the destination shard is the one with the smallest $\sum_{A_j \in S_i} Gas^{pred}_{A_j}$ value, where the set of account groups in the i-th shard is defined as S_i. Finally, the previous steps are repeated until every account group is relocated. In order to minimize the time complexity of ordering the account groups in the descending order, we used a max heap data structure for the priority queue. On the other hand, we used a min heap data structure for finding the shard with the smallest $\sum_{A_j \in S_i} Gas^{pred}_{A_j}$ value to minimize the time complexity.

Algorithm 1 Account Relocation Algorithm

```
Require:
    $N_{ag}$ : Number of account groups
    $N_{sh}$ : Number of shards
    $Gas^{pred}_{A_i}$: Predicted gas consumption of $i$-th account
    $Gas^{pred}_{S_i}$: Predicted gas consumption of $i$-th shard

1:  Priority queue of A: $Q_A \leftarrow \emptyset$
2:  $i \leftarrow 1$
3:  while $i < N_{ag}$ do
4:      $Q_A.push(\{A_i, Gas^{pred}_{A_i}\})$
5:      $i = i + 1$
6:  end while
7:
8:  $S_1 \leftarrow \emptyset, S_2 \leftarrow \emptyset, ..., S_{N_{sh}} \leftarrow \emptyset$
9:
10: while $Q_A$ is not empty do
11:     $\{A_k, Gas^{pred}_k\} = Q_A.pop()$
12:     $S_j = \min_{j=1}^{N_{sh}}(Gas^{pred}_{S_j})$
13:     $S_j.push(A_k)$
14: end while
```

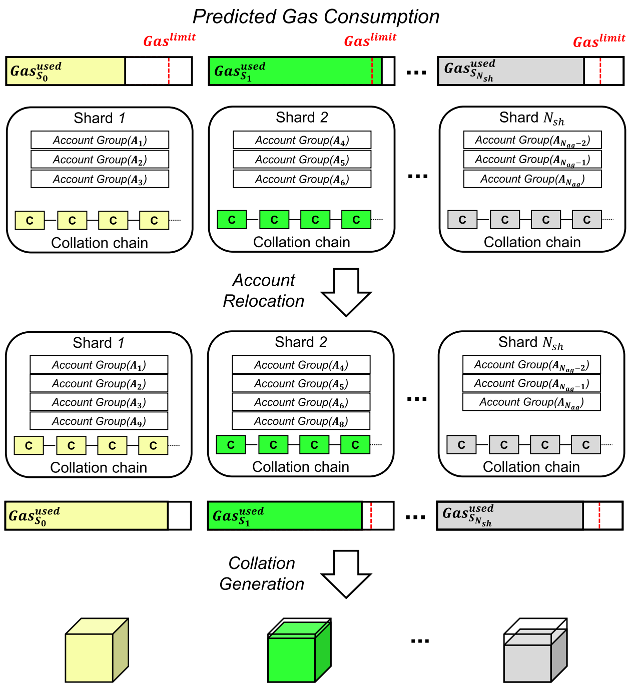


Fig. 7 Account relocation algorithm

5 Performance evaluation

In this section, we evaluate the performance of GARET and present the comparison with existing approaches using OMNeT++ 5.4.1 simulator [25]. In order to evaluate the performance of GARET, we first analyzed the real Ethereum workloads and generated similar synthetic patterns as

Table 3 Simulation workloads

	Type	G-Low (%)	G-Med (%)	G-High (%)
Gas	L_1^{Gas}	10	80	10
	L_2^{Gas}	20	60	20
	L_3^{Gas}	30	40	30
	Type	T-Small (%)		T-Large (%)
Transaction	L_1^{Vol}	90		10
	L_2^{Vol}	80		20
	L_3^{Vol}	70		30

much as possible. In addition, we report various performance results by varying the amount of *gas* consumption and the number of transactions. The overhead incurred by account relocation is also anaylzed.

5.1 Experiment setup

5.1.1 Real workload analysis

In order to properly generate workloads for the simulation, we analyzed the amount of *gas* consumption and the number of transactions in the Ethereum by using about 1,400 most recent Ethereum transactions from Etherscan [2].

From the analysis, we found that the number of transactions (i.e., transaction volume) generated by the heaviest top 20% accounts constitutes almost 80% of the total transactions. Furthermore, the average *gas* consumptions of bottom 30%, medium 40%, and top 30% are 20742, 51,857, and 105,153, respectively. Therefore, we classified the simulation workloads into two different traffic volumes (T-Small and T-Large) with three types of *gas* consumption (G-Low, G-Medium, G-High). Table 2 summarizes the types and distributions for the amount of *gas* consumption and the number of transactions. For example, G-Small includes transactions with the *gas* consumption of bottom 30%, while G-Medium and G-High include transactions with the *gas* consumption of medium 40% and high 30%, respectively. The *gas* value represents the average *gas* consumption of each category.

5.1.2 Workloads and comparison targets

Based on the analysis given above, we generated three types of *gas* consumption rate (L_1^{Gas}, L_2^{Gas}, L_3^{Gas}) and two types of traffic volume (L_1^{Vol}, L_2^{Vol}, L_3^{Vol}) as shown in Table 3. For example, L_3^{Gas} and L_2^{Vol} represent a workload type that matches with the real Ethereum workload.

We have implemented two different account allocation schemes for Ethereum sharding environments: S-ACC and D-TX. The S-ACC is a static address-based placement method currently proposed for the Ethereum sharding. And the D-TX is a relocation mechanism based on the number of transactions such that the number of transactions per account is evenly distributed between shards without considering the amount of gas consumption. For the comparison, we measured transaction throughput and makespan of transaction latency. Another version of GARET called GARET-PS (GARET with partial shuffling) has also been developed. The throughput and makespan of GARET-PS and GARET-FS[1] (GARET with full shuffling) were measured to analyze the effects of account relocation.

Table 4 Simulation parameters

Parameter	Value
C^T	12
Collation generation cycle	50
Gas^{limit}	8,000,000
Number of account group	100
Number of shard	20
Total number of transactions	150,000

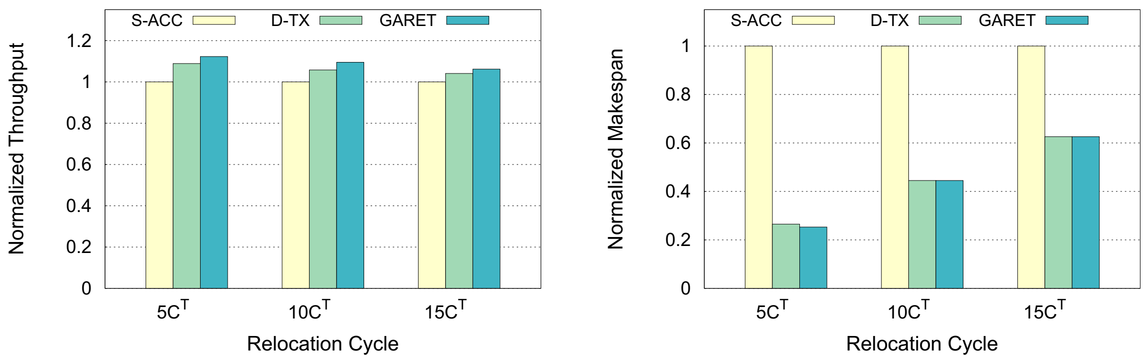
5.1.3 Simulation parameters

For the simulation, we assumed that an Ethereum sharding environment consists of 20 shards and there exist 100 account groups. The *gas* limit of each collation and C^T are set to 8,000,000 and 12, as is the case in current Ethereum. Each experiment was executed over 50 collation generation cycle, averaging out the values measured from experiments repeated three times. Table 4 summarizes the paramters used for the simulation.

5.2 Performance analysis

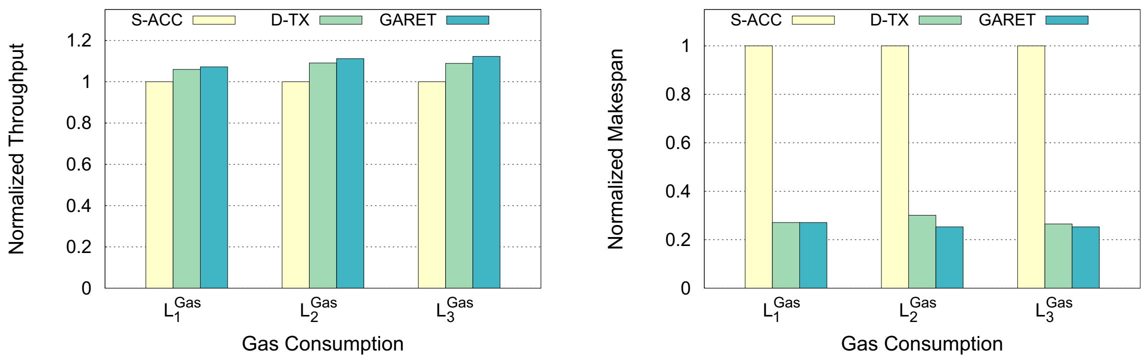
Figures 8, 9 and 10 compare the performance of S-ACC, D-TX, and GARET in terms of transaction throughput and makespan. For the comparison, we varied relocation cycle, *gas* consumption rate and traffic volume.

[1] GARET is also called as GARET-FS.



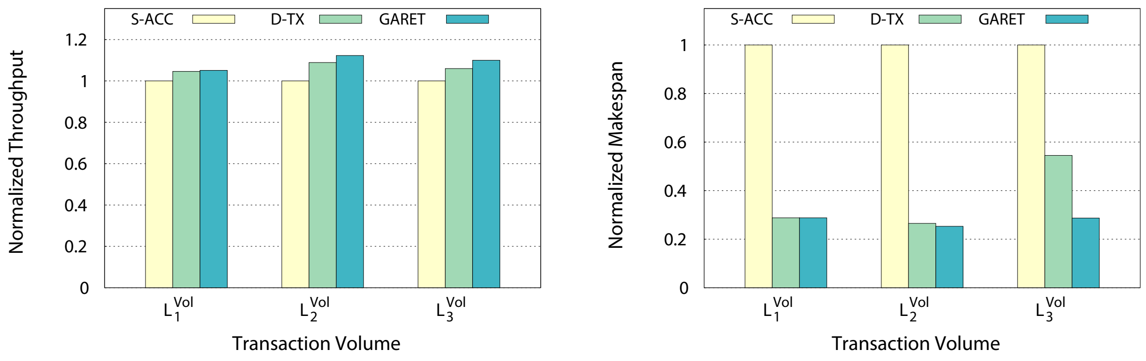
(a) Normalized Throughput **(b)** Normalized Makespan

Fig. 8 Performance by varying relocation cycle



(a) Normalized Throughput **(b)** Normalized Makespan

Fig. 9 Performance by varying gas consumption



(a) Normalized Throughput **(b)** Normalized Makespan

Fig. 10 Performance by varying traffic volume

5.2.1 Performance by varying relocation cycle

Figure 8 shows the throughput and makespan of three approaches by varying relocation cycle P^T ($5C^T$, $10C^T$, $15C^T$). As discussed in Sect. 4.1 (Eq. 5), P^T is a value obtained by multiplying N_{col} and C^T, where N_{col} is the number of collations generated in one shard at every relocation cycle and C^T is a time interval between two consecutive collations.

As shown in Fig. 8a, the GARET outperforms other schemes by about 9% on average and up to 12% to the maximum. The GARET also shows shorter makespan value than other schemes by about 55% on average and up to 74% to the maximum in all relocation cycle P^T as shown in Fig. 8b. Moreover, as we use shorter relocation cycle, the performance improvement gets bigger. This is because frequent relocation can balance the workloads better at the cost of relocation overhead.

5.2.2 Performance by varying gas consumption

Figure 9 shows the comparison of throughput and makespan by varying *gas* consumption type (L_1^{Gas}, L_2^{Gas}, L_3^{Gas}) given in Table 3. In this case, the relocation cycle and traffic volume are fixed to $5C^T$ and L_2^{Vol}, respectively.

As shown in Fig. 9a, the throughput of GARET is higher than those of other methods by about 8% on average and up to 12.3% to the maximum. The makespan of GARET is also a lot shorter than that of S-ACC as shown in Fig. 9b. It is worthy to note that as the complexity of transactions increases from L_1^{Gas} to L_3^{Gas}, the performance improvement of GARET compared to S-ACC also increases. Since the *gas* consumption is used for balancing the workloads among shards, the performance of GARET should be better in the case where the complexity of transactions increases.

5.2.3 Performance by varying traffic volume

Figure 10 shows the comparison of throughput and makespan by varying traffic volume (L_1^{Vol}, L_2^{Vol}, L_3^{Vol}) with the relocation cycle and traffic volume set to $5C^T$ and L_2^{Vol}, respectively.

Similarly, as shown in Fig. 10a, b, the GARET outperforms other methods in terms of throughput and makespan. The performance gap widens as we increase the traffic volume. Especially, the performance improvement in makespan is significant compared to those of S-ACC and D-TX when traffic condition is relatively heavy (L_3^{Vol}). This explains that the relocation mechanism proposed in this paper is more effective under heavy traffic conditions.

5.3 Relocation overhead analysis

It has been shown from previous results that the account relocation based on *gas* consumption can help improve the throughput and reduce the latency. However, it is also important to check how much overhead the account relocation incurs.

This subsection analyzes the relocation overhead of GARET and discusses the relationship between relocation count and performance. For this, we have modified the account relocation algorithm of GARET such that it only shuffles part of the shards instead of shuffling all shards. We call this as GARET-PS (GARET with partial shuffling).

Algorithm 2 `GARET with Partial Shuffling`

```
Require:
    $N_{ag}$ : Number of account groups
    $N_{sh}$ : Number of shards
    $Gas_{A_i}^{pred}$: Predicted gas consumption of $i$-th account
    $Gas_{S_i}^{pred}$: Predicted gas consumption of $i$-th shard
    $Gas^{limit}$ : Gas limit

 1: Priority queue of A: $Q_A \leftarrow \emptyset$
 2: $i,\ j \leftarrow 1$
 3: while $i < N_{sh}$ do
 4:     if $Gas^{limit} < Gas_{S_i}^{pred}$ then
 5:         while $S_i$ is not empty do
 6:             $A_j = S_i.pop()$
 7:             $Q_A.push(\{A_j, Gas_{A_j}^{pred}\})$
 8:             $j = j + 1$
 9:         end while
10:     end if
11: end while
12:
13: while $Q_A$ is not empty do
14:     $\{A_k, Gas_{A_k}^{pred}\} = Q_A.pop()$
15:     $S_j = \min_{j=1}^{N_{sh}}(Gas_{S_j}^{pred})$
16:     $S_j.push(A_k)$
17: end while
```

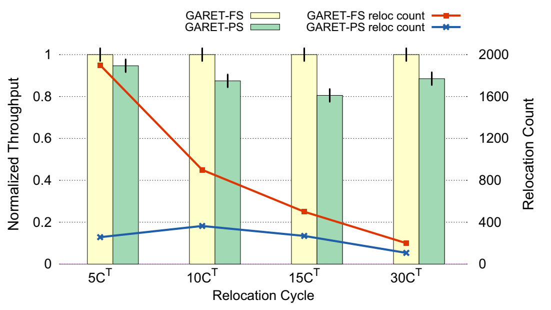
5.3.1 GARET with partial shuffling

Unlike GARET-FS (or GARET) that shuffles all shards at every relocation cycle, the GARET-PS relocates only account groups that belong to the shards with *gas* consumption exceeding the pre-defined *gas* limit as described in Algorithm 2. It also uses the predicted transaction load obtained from the load prediction algorithm. Initially, the GARET-PS creates a priority queue based on N_{ag} account groups, and finds the shards in which the *gas* consumption exceeds the *gas* limit. Then, it puts the account groups, which belong to the exceeded shards, into the priority queue. Next, it selects an account group from the priority queue and relocates it to a shard with the minimum *gas* consumption, repeatedly. This heuristic algorithm improves collation utilization while decreasing the relocation count as much as possible.

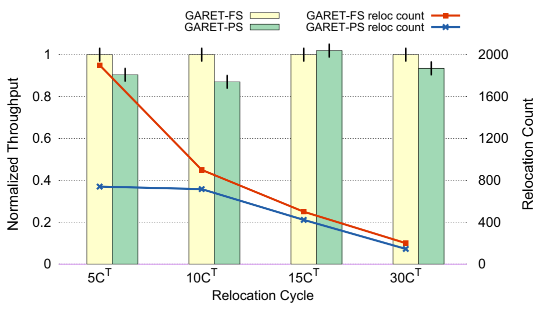
5.3.2 Overhead analysis

Figure 11 compares the throughput and relocation count of GARET-FS and GARET-PS by varying transaction traffic. The transaction traffic is defined as the percentage of total transaction load to total *gas* limit (i.e., total transaction load/total *gas* limit).

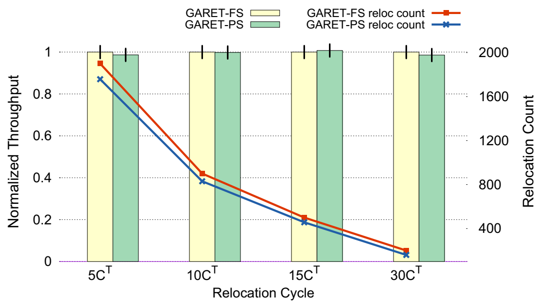
As shown in Fig. 11a, b, when the total transaction load per *gas* limit is under 70% and the relocation cycle is relatively short (e.g., $5C^T$), the difference in relocation count between GARET-FS and GARET-PS is huge as



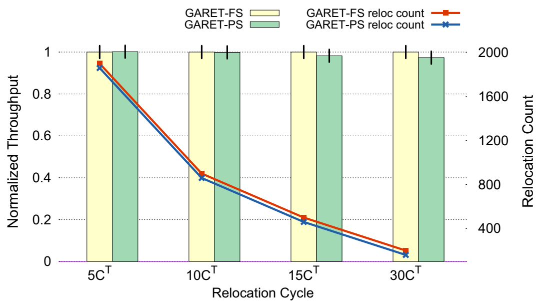
(a) Total Transaction Load per Gas Limit = 60%



(b) Total Transaction Load per Gas Limit = 70%



(c) Total Transaction Load per Gas Limit = 80%



(d) Total Transaction Load per Gas Limit = 100%

Fig. 11 Relocation overhead analysis

expected. As we increase the relocation cycle or the total transaction load per *gas* limit exceeds beyond 80%, both GARET-FS and GARET-PS show similar number of relocation counts.

Surprisingly, however, the GARET-FS always outperforms the GARET-PS in terms of throughput regardless of the relocation count. For example, in the case where the total transaction load per *gas* limit is 60%, the GARET-FS performs better than the GARET-PS by about 13% on average and up to 20% to the maximum, although the GARET-PS reduces the relocation count by up to 87%. As the transaction traffic increases, the gap in the number of relocation count and throughput between GARET-FS and GARET-PS gradually decreases as shown in Fig. 11c, d.

The aforementioned results indicate that the benefits obtained by relocation seem to overwhelm the overhead incurred by frequent account relocation. That is, the relocation cost does not significantly affect the throughput. The main reason for this is that relocating accounts requires only a change in the data structure managed by the validator in each shard. For example, when an account relocation is requested by the account relocation algorithm, the changes are broadcasted to other validators to reach a consensus as shown in Fig. 5. Although a small amount of network traffic is generated for the consensus, this would not make a large impact on the Ethereum throughput.

6 Related work

There have been numerous solutions to scaling blockchains by replacing consensus protocol [7, 10, 19], limiting network membership [11], and modifying network topology [8, 23]. Among them, this section introduces various research efforts related to blockchain sharding.

RSCoin [14] proposed a blockchain sharding protocol using the central bank authorized monetary supply. A set of validators, called Mintette, delegated by the central bank are responsible for the maintenance of disjoint ledger. Since the whole transaction validation processes are executed by entrusting Mintettes, and each Mintette validates disjoint transactions, RSCoin prototype shows 2,000 TPS, which is about 300 times higher than Bitcoin with strong auditability guarantees. However, RSCoin is not an ideal solution for sharding protocol, since its architecture is centralized.

Luu et al. [20] proposed Elastico, the first sharding protocol for public blockchains. Elastico statically splits the blockchain network into multiple smaller groups called the committee using the identity. Each established committee runs Byzantine consensus protocols independently and reconfigures in every epoch using generated randomness, which is modified epochrandomness for better fault tolerance and scalability. However, Elastico can only tolerate up to a 1/4 fraction faulty with high failure probabilities.

Kokoris-Kogias et al. [18] proposed OmniLedger, unspent transaction output (UTXO) [22] based sharding protocol. To resolve the problems of Elastico. the OmniLedger considers an atomic cross-shard transaction that is not addressed by previous protocols. Furthermore, the OmniLedger applied verifiable random function (VRF) for leader node election against adversary nodes. Despite several methods for improvement, the OmniLedger still tolerates up to a 1/4 fraction faulty and is vulnerable to denial-of-service (DoS) attacks.

Mahdi Zamani et al. [28] proposed RapidChain that solves the problems of the aforementioned protocols with additional contributions. RapidChain shows better resiliency and throughput than other protocols. Furthermore, it suggests decentralized bootstrapping, a routing protocol for sharding that is not mentioned in any other protocols, and other optimization methods [5, 24].

Chainspace [4] is a state-based smart contract platform with sharding protocol. The main challenging issue is to synchronize the state of shards with cross-shard transactions. Chainspace tries to resolve this issue using sharding-byzantine agreement and atomic commit (S-BAC) based on state lock-unlock protocol. However, the locking protocol on smart contract requires methods for the attacks that lock entire state.

Ethereum [9], one of the most popular state-based blockchain platform, also suggested its own sharding protocol. Although the detailed protocol is still under development up to now, the overall architecture has initially been proposed. In Ethereum sharding, a special node called validator in each shard is responsible for the validation and execution of transactions within the shard. In order for the validator to generate a sequence of collations, all account groups are initially assigned to each shard statically by using address prefix. This causes each shard to have uneven transaction loads, which may degrade the overall performance.

Although several proposals have been made for blockchain sharding, the dynamic management of account groups has not been addressed in any research efforts.

7 Conclusion

This paper proposed a *gas* consumption-aware dynamic account relocation mechanism called GARET in the Ethereum sharding environments. The GARET selects *gas* consumption as a metric to measure the complexity of a transaction and uses this metric to balance the workload of each shard by relocating account groups dynamically. We formulated this problem as a multi-dimensional knapsack problem (MKP) and proposed a heuristic algorithm to maximize the throughput and minimize the makespan of transaction latency. Through the simulation, it was shown that the GARET outperforms other allocation mechanisms such as S-ACC and D-TX by 12% and 74% in terms of throughput and makespan, respectively. We also analyzed the relocation overhead and showed that the relocation cost does not significantly affect the throughput.

Although the GARET is currently targeted only at Ethereum, its design principle can also be applied to the sharding mechanism for other blockchains as well. Since sharding is a mechanism to partition a network into several pieces and maintaining balanced transaction load among shards is crucial to improve throughput, the formulation of the problem and its application to real systems should not be much different.

Acknowledgements This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2020-2017-0-01628) supervised by the IITP (Institute for Information & communications Technology Promotion)

References

1. Ethereum sharding faq. https://github.com/Ethereum/wiki/wiki/Sharding-FAQs. Accessed 15 Oct 2018
2. Etherscan: The ethereum block explorer. https://etherscan.io/. Accessed 24 May 2018
3. Pending ethereum transactions after cryptokitties' realse. https://www.theatlas.com/charts/rkt8jKMZz
4. Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: Chainspace: A sharded smart contracts platform. arXiv preprint arXiv:1708.03778 (2017)
5. Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., Stern, J.: Scalable secure storage when half the system is faulty. In: International Colloquium on Automata, Languages, and Programming, pp. 576–587. Springer (2000)
6. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: Medrec: Using blockchain for medical data access and permission management. In: 2016 2nd International Conference on Open and Big Data (OBD), pp. 25–30. IEEE (2016)
7. Bentov, I., Lee, C., Mizrahi, A., Rosenfeld, M.: Proof of activity: Extending bitcoin's proof of work via proof of stake. IACR Cryptol. ePrint Arch. **42**, 34–37 (2014)
8. Brown, R.G., Carlyle, J., Grigg, I., Hearn, M.: Corda: an introduction. R3 CEV **1**, 15 (2016)
9. Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014). https://github.com/ethereum/wiki/wiki/White-Paper. Accessed: 33 Aug 2016
10. Buterin, V., Griffith, V.: Casper the friendly finality gadget. arXiv preprint arXiv:1710.09437 (2017)
11. Cachin, C.: Architecture of the hyperledger blockchain fabric. In: Workshop on distributed cryptocurrencies and consensus ledgers, vol. 310, p. 4 (2016)
12. Cai, W., Wang, Z., Ernst, J.B., Hong, Z., Feng, C., Leung, V.C.: Decentralized applications: The blockchain-empowered software system. IEEE Access **6**, 53019–53033 (2018)
13. Chu, P.C., Beasley, J.E.: A genetic algorithm for the multidimensional knapsack problem. J. Heuristics **4**(1), 63–86 (1998)
14. Danezis, G., Meiklejohn, S.: Centrally banked cryptocurrencies. arXiv preprint arXiv:1505.06895 (2015)

15. Dorri, A., Kanhere, S.S., Jurdak, R., Gauravaram, P.: Blockchain for iot security and privacy: the case study of a smart home. In: 2017 IEEE international conference on pervasive computing and communications workshops (PerCom workshops), pp. 618–623. IEEE (2017)
16. Eyal, I., Gencer, A.E., Sirer, E.G., Van Renesse, R.: Bitcoin-ng: A scalable blockchain protocol. In: 13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16), pp. 45–59 (2016)
17. Kim, S., Song, J., Woo, S., Kim, Y., Park, S.: Gas consumption-aware dynamic load balancing in Ethereum sharding environments. In: 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W), pp. 188–193. IEEE (2019)
18. Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: Omniledger: a secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 583–598. IEEE (2018)
19. Larimer, D.: Delegated proof-of-stake (dpos). Bitshare whitepaper (2014)
20. Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 17–30. ACM (2016)
21. Mettler, M.: Blockchain technology in healthcare: the revolution starts here. In: 2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom), pp. 1–3. IEEE (2016)
22. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2009). http://www.bitcoin.org/bitcoin.pdf
23. Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016)
24. Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM (JACM) **36**(2), 335–348 (1989)
25. Varga, A.: The omnet++ discrete event simulation system. In: Proceedings of the European Simulation Multiconference (ESM'01) (2001)
26. Vukolić, M.: The quest for scalable blockchain fabric: proof-of-work vs. BFT replication. In: International workshop on open problems in network security, pp. 112–125. Springer (2015)
27. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07) (2017). https://ethereum.github.io/yellowpaper/paper.pdf. Accessed: 03 Jan 2018
28. Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 931–948. ACM (2018)
29. Zheng, Z., Xie, S., Dai, H.N., Chen, X., Wang, H.: Blockchain challenges and opportunities: a survey. Int. J. Web Grid Serv. **14**(4), 352–375 (2018)

Sangyeon Woo is a Master candidate of Computer Science and Engineering Department a sogang University, Korea. He has received B.S degrees from the Computer Science and Engineering, Sogang University Korea, in 2018. His research interests include Blockchain, Distributed System.

Jeho Song is a Master candidate of Computer Science and Engineering Department a sogang University, Korea. He has received B.S degrees from the Computer Science and Engineering, Sogang University Korea, in 2018. His research interests include Blockchain, Distributed System.

Sanghyeok Kim is an employee of TmaxSoft, Korea. He received her B.S. and M.S. degrees in computer science from Sogang University. Now he works for TmaxSoft as a cloud programmer.

Youngjae Kim received his Ph.D. degree in Computer Science and Engineering from Pennsylvania State University, University Park, PA, USA in 2009. He is currently an associate professor in the department of computer science and engineering at Sogang University, Seoul, Republic of Korea. Before joining Sogang University, Dr. Kim was a staff scientist in the U.S. Department of Energy's Oak Ridge National Laboratory (2009-2015) and an assistant professor in Ajou University, Suwon, Republic of Korea (2015-2016). Dr. Kim received the B.S. degree in computer science from Sogang University, Republic of Korea in 2001, and the M.S.

degree from KAIST in 2003. His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.

Sungyong Park is a professor in the Department of Computer Science and Engineering at Sogang University, Seoul, Korea. He received his B.S. degree in computer science from Sogang University, and both the M.S. and Ph.D. degrees in computer science from Syracuse University. From 1987 to 1992, he worked for LG Electronics, Korea, as a research engineer. From 1998 to 1999, he was a research scientist at Telcordia Technologies (formerly Bellcore), where he developed network management software for optical switches. His research interests include cloud computing and systems, virtualization technologies, high performance I/O and storage systems, and embedded system software.