Shard Scheduler: object placement and migration in sharded account-based blockchains

Michał Król
City, University of London
michal.krol@city.ac.uk

Onur Ascigil
University College London
o.ascigil@ucl.ac.uk

Sergi Rene
University College London
s.rene@ucl.ac.uk

Alberto Sonnino
Facebook Novi
asonnino@fb.com

Mustafa Al-Bassam
LazyLedger
mustafa@lazyledger.io

Etienne Rivière
UCLouvain
etienne.riviere@uclouvain.be

ABSTRACT

We propose Shard Scheduler, a system for object placement and migration in account-based sharded blockchains. Our system calculates optimal placement and decides of object migrations across shards and supports complex multi-account transactions caused by smart contracts. Placement and migration decisions made by Shard Scheduler are fully deterministic, verifiable, and can be made part of the consensus protocol. Shard Scheduler reduces the number of costly cross-shard transactions, ensures balanced load distribution and maximizes the number of processed transactions for the blockchain as a whole. It leverages a novel incentive model motivating miners to maximize the global throughput of the entire blockchain rather than the throughput of a specific shard. Shard Scheduler reduces the number of costly cross-shard transactions by half in our simulations, ensuring equal load and increasing the throughput 3 fold when using 60 shards. We also implement and evaluate Shard Scheduler on Chainspace, more than doubling its throughput and reducing user-perceived latency by 70% when using 10 shards.

1 INTRODUCTION

Sharding emerged as one of the most promising layer-1 solutions to the scalability problems of blockchains [1, 13, 22, 25, 39, 41]. A sharded system divides the blockchain infrastructure into groups called shards. Each shard has its own miners, holds a subset of the state, and processes a subset of transactions. This technique has the potential to increase the number of processed transactions per second, as they can be verified and agreed on in parallel by independent groups of miners. In theory, by increasing the number of shards, we can increase the global throughput of the blockchain.

A sharded blockchain [38] can be seen as a distributed database where each transaction performs write operations, creating, destroying or modifying objects in one or multiple partitions (shards). We can distinguish between transactions writing to only one shard (intra-shard transactions) or to multiple shards (cross-shard transactions). Intra-shard transactions are relatively cheap and can be agreed on using the consensus protocol within their shard. In contrast, cross-shard transactions are more costly as they require local consensus in all involved shards as well as a cross-shard agreement between these shards. This is achieved using expensive techniques such as 2-phase commit [1, 22, 31] or mutex-based protocols [13, 41]. Finally, cross-shard transactions must be included in the chains of all shards holding involved accounts resulting in state inflation. The placement of objects in shards plays a crucial role in determining

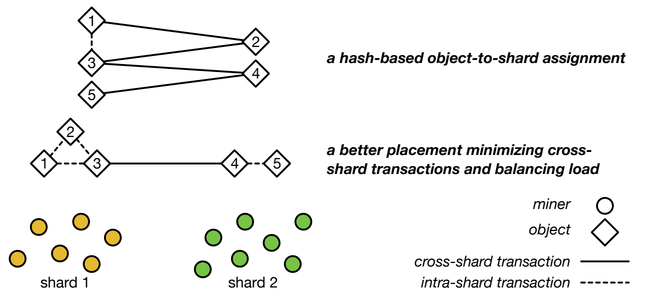
a hash-based object-to-shard assignment
a better placement minimizing cross-shard transactions and balancing load
miner
object
cross-shard transaction
intra-shard transaction
shard 1
shard 2
1
2
3
4
5

Figure 1: Object-to-shard assignment: a static placement (*e.g.* hash-based) results in a high number of cross-shard transactions. A better placement could place (or migrate) objects 1, 2 and 3 in (to) shard 1 and objects 4 and 5 in (to) shard 2.

the overall performance (*i.e.* the Transaction per Second–TPS–rate and the user-perceived confirmation latency).

In this paper, we focus on the account-based data model. Account-based objects are persistent. They represent user accounts (*i.e.* user balance) or smart contracts and can be modified multiple times. Placing an object in a shard in the account-based model influences all future transactions for this object (in contrast to single-use transaction outputs in the UTXO model). Ethereum, the largest blockchain system supporting smart contracts, is an example of an account-based blockchain transitioning into a sharded mode of operation [13].

Existing sharded blockchain designs generally use a static hash-based object-to-shard assignment [1, 13, 22, 25, 39, 41]. The hash space of object identifiers is divided equally between shards, and hashing the identifier of an object allows clients and miners to deterministically determine its location without using additional indexing services. In the long run, hash-based allocation equally spreads the load across shards but causes loss of data locality. Frequently interacting accounts may be spread across multiple shards causing costly cross-shard interactions [3]. Furthermore, a fixed assignment cannot always react to activity bursts of accounts located in a single shard, causing short-term load imbalance. Both problems become more pronounced with an increasing number of shards and with an increasing number of accounts involved in each transaction, *e.g.* as the result of the smart contracts executions.

Figure 1 presents a simplified view of a blockchain with two shards and five accounts. Edges represent interactions (transactions) between accounts. The upper hash-based placement results in a high number of cross-shard transactions. A better placement is a compromise between load-balancing and the number of cross-shard transactions. We note that achieving such a placement through initial placement decisions only is not necessarily possible, and may require *migrating* objects between shards (*e.g.* accounts 2 and 5 in our example). Migration operations [14, 15, 29] require additional transactions. The individual cost of these transaction executions, as well as the overhead they impose on the blockchain as a whole, must be worth paying, *i.e.* result in higher throughput and lower confirmation latency for future transactions.

Contributions. We present Shard Scheduler, a novel approach for deciding and enforcing object placement and migration decisions in sharded, account-based blockchains. Our scheduler balances the load between shards and improves data locality. It leverages the possibility to initiate account migrations when necessary and seeks to maximize the global throughput of the blockchain. At the same time, Shard Scheduler remains simple, deterministic, and verifiable for all the miners in the network to prevent abuse. Shard Scheduler is executed by the miners and does not require modifications of the clients, who are nonetheless able to verify the legitimacy of migration decisions taken as part of their transaction execution. Finally, Shard Scheduler makes scheduling decisions worth enacting for rationale miners through economic incentives. We do not seek to propose novel mechanisms for handling cross-shard transactions and account migrations, but rather build upon the different proposals by other authors [1, 13, 22, 25, 39, 41]. We only make minimal and common assumptions on the capabilities of the underlying sharded blockchain, allowing Shard Scheduler to be implemented on top of a vast range of account-based blockchains, from the upcoming evolution of Ethereum [13], to current systems such as Zilliqa [37].

Outline. We present a background on account-based blockchains and sharding mechanisms in Section 2. We outline the design and perimeter of use of Shard Scheduler in Section 3, present our assumptions on the underlying sharded blockchain together with our design goals in Section 4, and present our system model in Section 5. We then present our contributions as follows.

Our first contribution, presented in Section 6, is an analysis of the transaction history from Ethereum from a perspective of a sharded execution. We use the Ethereum Virtual Machine (EVM) to extract all accounts that were modified by every transaction. We then investigate the activity of the accounts, their data locality, and the load balancing when using a static hash-based assignment.

In Section 7 we present the design of Shard Scheduler, a transaction scheduler for sharded, account-based blockchains. Shard Scheduler observes system load and interactions between accounts to place and migrate accounts across shards to maximize the throughput.

In Section 8, we develop and discuss an incentive scheme for sharded blockchains that motivates miners to maximize the TPS of the blockchain as a whole. By deploying this scheme, we free blockchain end-users from costly, manual migrations of the state and avoid associated security problems. Furthermore, we incentivize miners to perform migrations providing the highest global TPS instead of focusing on the fees collected on their own shard.

In Section 9, we quantify the performance gain over a hash-based approach using a simulator.

In Section 10, we present the integration of Shard Scheduler with the Chainspace [1] sharded blockchain system and the results of its deployment on a large-scale testbed. Our evaluation shows that Shard Scheduler can adapt to many potential configurations of a sharded environment, more than doubles the throughput of the system, and lowers the latency by 65% for 60 shards.

Finally, Section 12 presents an analysis of Shard Scheduler properties, discusses future work, and concludes the paper.

2 BACKGROUND

In this section, we present background on account-based blockchains. We then discuss their transition into a sharded mode of operation, cross-shard transactions and migrations.

2.1 Accounts, state and transactions

A blockchain is an append-only ledger maintained by a number of nodes called miners. A blockchain is expanded by the addition of blocks by designated miners, who receive incentives for extending the chain with correct blocks and behave according to the protocol. A block consists of a block header together with a list of transactions. Transactions modify the state of the ledger ranging from simple coin transfers to invocation of sophisticated smart contracts. The block header contains a hash of the block, the hash of the previous block, the hash of the state snapshot at a given time, and additional information related to the consensus protocol. Each block has a fixed capacity limiting the number of transactions it can contain. Including a transaction in a block requires some of the available total capacity of the blockchain system. We refer to the capacity required by a transaction as the *cost* of that transaction. The cost usually depends on the size of the transaction (as done in Bitcoin [28]) or its complexity (as done in Ethereum [40]).

Miners that store all the blocks (including all the transactions) are called full nodes. In contrast, light nodes store only block headers and reactively pull required state elements or transactions from full nodes when needed. Light nodes can verify the integrity of the received data by comparing its hash against the value in the corresponding block header (*i.e.* using Merkle proofs [26]).

In the account-based data model, the state of a blockchain consists of a list of objects representing accounts and their respective states. An account is accessed by its identifier (*e.g.* a hash of its owner's public key) and represents an externally owned account (EOA), or a contract account (CA). For EOAs, the state consists of their balance. For CAs, the state may include more complicated data structures related to the logic of a smart contract. Importantly, while the state of EOAs is small and does not grow in time, the state of CAs can inflate as more data is put in the storage.

The state of an account can be modified by two types of transactions: external and internal. A transaction is external if sent from an EOA. For instance, a coin transfer, a contract creation, and a contract invocation are the 3 main external operation types happening in Ethereum [6]. Alternatively, a transaction is internal if it results from executing a smart contract invoked by an external transaction. A single external transaction may lead to multiple internal transactions depending on the smart contract logic.

A regular account-based transaction (*i.e.* a simple coin transfer) modifies the state of up to 2 EOAs (the balance of the sender and that of the receiver). With the addition of Smart Contracts, transactions can lead to the modification of multiple accounts. Algorithm 1 presents a Smart Contract implementing a *payAll()* function. Calling

Algorithm 1 Example of a smart contract function modifying the state of multiple accounts.

```
1: procedure PAYALL
2:     users ← a list of users to be paid
3:     amount ← amount to pay each account
4:     for user in users do
5:         if user.balance < 10 then
6:             user.transfer(amount)
```

this function modifies the state of the caller (to pay the transaction fees), the smart contract (to decrease its balance), and all the accounts stored in the *users* map (to increase their balance), provided they currently have less than 10 coins. Smart contracts can also interact with and modify the state of other contracts by invoking their functions. Processing smart contract transactions require the write and read sets to be known to the consensus protocol layer based on the current state of the blockchain.

2.2 Sharding

In fully sharded environments[1], the blockchain is split into multiple groups with their own chains of blocks and miners. Each shard maintains and modifies the state of only a subset of the accounts existing in the system. Objects to shards assignments are usually static unless changed in explicit migrations caused by miners or users. A migration locks (or destroys) an object in the source shard and recreates it in the destination shard using an atomic transaction. The object identifier may or may not change during the migration depending on the underlying objects-to-shards mapping system. Shards are expanded by running local consensus protocol between shard-specific miners. Some designs [13, 41] use a main chain that is used for coordination. The main chain periodically assigns miners to shards to prevent malicious miners from freely migrating and taking over a specific shard. As a result, only miners assigned by the main chain have the right to participate in the intra-shard consensus [38]. Furthermore, the main chain stores block headers of all the shards, which facilitates cross-shard communication [13, 41].

Cross-shard communication and migrations. Transactions modifying the state of accounts placed in a single shard can be processed using intra-shard consensus similarly as in a non-sharded scenario. If the involved accounts are spread across multiple shards, however, executing the transaction requires cross-shard consensus to ensure the atomicity of transactions. There are two main types of cross-shard consensus protocols, (*i*) protocols based on a two-phase commit protocol [16] such as S-BAC [1] and Atomix [22], and (*ii*) mutex-based protocols such as RapidChain [41] and the upcoming version of Ethereum [13]. In all cases, a cross-shard transaction requires an intra-shard consensus run in each shard holding at least one of the involved accounts together with the run of cross-shard coordination. The latter always causes additional overhead in all the involved shards. If any of the shards involved rejects a transaction, all other shards should likewise reject it to guarantee atomicity; that is, an atomic commit protocol typically runs across all the concerned shards to ensure the transaction is accepted by all or none of those shards. It also means that the processing time of a cross-shard transaction is determined by the slowest shard.

Objects can be migrated across shards by users (in explicit cross-shard transactions [29]) or by miners (as a part of the consensus protocol [41]). Performing migrations cause processing overhead for the miners and transaction fees for the end-users. The cost of migrations can be reduced when combined with cross-shard transactions. If account A in shard 1 sends a transaction to account B in shard 2, both accounts may remain in their respective shards (causing a costly cross-shard consensus round) or one of the accounts can be migrated to the shard of the other one[2]. In the latter case, the migration cost still needs to be paid, but further processing requires cheaper intra-shard consensus in the destination shard.

The use of migration can have a significant impact on the performance of the account-based blockchains. This impact can be positive or negative depending on the migration decisions made. Splitting frequently interacting communities may negatively impact the throughput of the entire system for many future blocks. On the other hand, migrations can equally spread the load across shards on a per-block basis improving resource utilization. Migrations increase the cost of individual transactions but, if done correctly, can also bring long-term performance gains. Correctly incentivizing decisions that are good for the blockchain as a whole can significantly improve the throughput of the entire system. We further discuss the topic in Section 8.

3 OVERVIEW

The goal of Shard Scheduler is to integrate smart, automatic account placement and migrations decisions to improve the throughput of the sharded blockchain *as a whole*. Our system strikes a balance between balanced load distribution, data locality, and the number and costs of performed migrations. Shard Scheduler performs migrations that are supported by the underlying consensus protocol, introduce relatively low short-term overhead, and reduces the cost of future transactions in the long run.

A fundamental design principle of Shard Scheduler is the implementation of our system on miners as a part of the consensus protocol. While client-based migrations have been proposed for throughput improvements in the UTXO model [29], such an approach is not effective for account-based blockchains. A transaction in the account-based model modifies the state of multiple accounts (*e.g.* sender, receiver, smart contract) but is authorized only by its sender. It thus restricts potential migrations to moving the sender only. In contrast, migration decisions taken by miners as a part of transaction processing can achieve optimal placement by moving any account involved in a transaction.

In Shard Scheduler, all migration decisions are taken based on a state snapshot of the blockchain, are deterministic, and can be verified by other miners. With decision verifiability, Shard Scheduler protects against malicious miners who might attempt a denial of service attack by forcing sub-optimal migrations. Our system requires only simple arithmetic operations to take optimal migration

[1]Fully sharded environments split both the state and the transaction processing. Some sharded blockchains such as Monoxide [39] or Elastico [25] only split the latter and do not fall into this category.

[2]Both accounts can be also migrated to a third or different shards. However, such migration would cause significant overhead to the system.

	Ethereum+	RapidChain	Chainspace	Omniledger
Smart Contracts	✓	✗	✓	✗
Beacon Chain	✓	✓	✗	✓
Miners reshuffling	✓	✓	-	✓
Write set specified by transactions	?	✓	✓	✓

Table 1: Shard Scheduler assumptions in existing systems.

decisions and introduces only negligible overhead to the transaction processing.

Shard Scheduler decouples the mining process from the collection of fees and aligns rewards collected by the miners with the throughput of the entire blockchain, rather than with the performance of a single shard. Rational miners are thus incentivized to pay the overhead cost related to automatic migrations. Finally, Shard Scheduler is completely transparent for the clients submitting transactions to the blockchain and does not require any client-side modifications.

The performance of a distributed system is tightly coupled to its submitted workload. Before outlining our design, we analyze the transaction history of Ethereum together with state-dependent smart contracts calls and extract new insights that allow understanding expected cross-shard interaction dynamics and shape the design of Shard Scheduler.

4 ASSUMPTIONS AND DESIGN GOALS

We base our assumptions on Ethereum [13, 40], the main account-based blockchain transitioning into a sharded environment with support for smart contracts. Where Ethereum does not yet specify all the design details of its transition to a sharded operation, we assume functionalities provided by academic sharded blockchains (Omniledger [23], Chainspace [1], and RapidChain [41]). The characteristics of these systems are shown in Table 1.

4.1 Security Assumptions

We distinguish two types of actors:

- *users* are owners of EOAs that use the blockchain;
- *miners* are maintainers of the blockchain.

We assume the presence of arbitrary malicious actors that can play the role of users or miners and try to disturb the system. No single user or miner is trusted by its peers. However, as for many sharded blockchain designs [1, 23, 32, 41], we assume that all shards have an honest consensus majority. With the current single-chain economic models applied to a sharded environment, miners may be incentivized to deviate from the protocol when taking migration decisions. In Section 8, we develop an economic model for sharded blockchain that makes the honest majority assumption more probable in a real-world deployment.

We assume a partially synchronous network for 2PC-based protocols[3] [10], and a synchronous network for mutex-based protocols (in light of recent replay attacks against sharded blockchains [32]).

We assume a sharded blockchain environment as envisioned by Omniledger [23]. A measure of time is determined from the chain length of an arbitrary shard and is divided into *epochs* of equal length. In every epoch, nodes can manifest their intention to become miners for the next epoch by registering their public key to a dedicated smart contract [1] (or hardcoded logic on a beacon chain [13, 23]). The system runs a black box Sybil detection algorithm (typically proof-of-work [28, 40] or proof-of-stake [20]) that outputs the list of registered public keys of the nodes that will become miners during the next epoch. At the start of a new epoch, miners are shuffled and assigned to shards at random.

We assume the presence of a main chain (as in Omniledger [23] and RapidChain [41], and as proposed for Ethereum [13]) that stores the block headers of all the shards. Each miner is a full client for its respective shard and acts as a light client for the beacon chain and all the other shards. We assume the presence of a mapping service holding current accounts-to-shards assignments (*e.g.* implemented as a Distributed Hash Table).

Processing a cross-shard transaction requires modifying a set of objects. For a simple transfer transaction (*i.e.* not a call to a smart contract) the set contains the sender and the receiver and can be read directly from the transaction data. For blockchains supporting smart contracts, the list of involved accounts for a specific execution may depend on the current state across multiple shards. In Algorithm 1 for instance, the caller, the contract, and accounts from *users* may be spread across multiple shards. The required state and a list of involved objects can be either proactively locked and provided to the miners by the user as part of the transaction data (as done in Chainspace [1]) or reactively pulled by miners executing the transactions (as discussed for Ethereum [13]).

Our system is orthogonal to the actual implementation of cross-shard transactions with or without smart contracts. For each transaction, Shard Scheduler relies only on a write set (*i.e.* accounts whose state will be modified by this transaction). Such a set is already required to process smart contract transactions (Section 2). Finally, we assume that each cross-shard transaction is forwarded to a shard responsible for its execution. We refer to this shard and more specifically to a miner including the transaction in its block, as the *transaction coordinator*. The transaction coordinator obtains a list of accounts to be modified and coordinates other shards involved in the transaction.

4.2 Design Goals

The design of Shard Scheduler targets the following properties.

Migration and placement recommendations. Shard Scheduler analyzes interactions between accounts and issues recommendations specifying how an incoming transaction should be handled and, in particular, what (if) migrations should happen. These recommendations have the goal of keeping frequently interacting accounts within one shard while providing a balanced load across shards. By reducing the number of cross-shard transactions and their associated overheads, and avoiding performance degradation due to overloaded shards, two goals participate in unison to an increased throughput (total number of transactions per second for a given capacity).

Recommendation verifiability. Each recommendation is deterministic and can be reliably verified by all other miners. Shard Scheduler recommendations are part of the consensus and block validation protocols. This property is required to ensure the availability of the blockchain. Without verifiability, malicious miners may attempt to

[3]This assumption is not required by the cross-shard consensus protocol *per se*, but by the BFT protocol running within each shard.

Parameters			
$s_i \in S$	states	$t_i \in T$	transactions
$o_i \in O$	objects	$acc_i \in ACC$	accounts
b_i	balance of acc_i	ϕ	mapping service
$c(t_i)$	cost of t_i	C_i	capacity of s_i
$m_{j \to k}(acc_i)$	migrations		

Table 2: Notations.

move objects towards an overloaded shard or split frequently interacting communities, thus increasing the cost of transactions and lowering the number of transactions per second [27]. Such a denial of service attack, even when targeting a single shard, influences the throughput of the entire blockchain due to the impact on cross-shard transactions.

Lightweight recommendations. Shard Scheduler recommendations are generated on a per-transaction basis. The system ensures that the amount of required computation is low and can be easily performed by all miners without introducing significant space and time overhead. Shard Scheduler operations remain computationally tractable also when the number of accounts present in the blockchain grows. Shard Scheduler does not introduce any significant network overhead (*i.e.* fetching large, additional state from other shards).

No changes for the clients. Shard Scheduler is transparent for EOA owner and, in contrast to related work [29], does not require additional operation or maintenance of state by users.

Incentive model. Shard Scheduler provides an incentive model for the miners to motivate them to follow the recommendations. The reward of each miner is proportional to the amount of performed work (*i.e.* the number of mined blocks) and the total amount of rewards acquired by the blockchain as a whole. Miners are still incentivized to compete for producing new blocks include a maximum amount of transactions. However, miners do not benefit from keeping excessive numbers of accounts in their shards and ignoring ingoing or ongoing migrations recommended by Shard Scheduler.

5 SYSTEM MODEL AND NOTATION

We present the notations used throughout the rest of the paper, and the model in which Shard Scheduler operates. Notation are summarized in Table 2.

5.1 Blockchain Model

The blockchain is maintained by a number of miners $m \in M$ validating and processing transactions. We adopt a similar blockchain model as Al-Bassam *et al.* [2]. We model the blockchain as a set of state variables that encode its state $s \in S$ and transactions $t \in T$; at any time $s \in S$ represents a snapshot of the state of every object (*i.e.* accounts, smart contracts). The blockchain maintains an append-only log of ordered transactions $\{t_0...t_n\} \in T$. The blockchain starts in an initial state $s_0 \in T$ and transitions from one valid state to the valid next state with each valid transaction $t_i(s_i) \longrightarrow s_{i+1}$.

Sharded blockchains. In sharded blockchains, nodes are divided into groups called shards $z \in Z$, and each shard maintains a subset of the objects. Shard z_j at step i maintains $s_{ij} : acc_k \in ACC_j$. We assume a service mapping objects to their respective shard $\phi(acc_i) \longrightarrow z_j$ as defined by Chainspace [1].

Transactions lifecycle. Each node holds all incoming transactions in a fixed-sized *transaction pool* (also called *mempool*). At every time step, the transaction pool of every node is completely filled with transactions from clients. Executed transactions are removed from the transaction pool. Only valid transactions are considered (*e.g.*, for coin transfers, both the sender and receiver exist and the sender has sufficient funds to make the transfer). Invalid transactions are discarded.

5.2 Processing Capacity

The concept of *processing capacity* is key to our model. Every time period, each shard z_i has a processing capacity C_i indicating how many transactions it can process during that time period while maintaining a constant transaction latency. In practice, this capacity can be limited by a number of factors such a network conditions, the size of the shard, and specific implementations. We assume that each shard has the same capacity ($\forall i, j\ C_i = C_j$), and that the capacity of the whole blockchain is the sum of the capacity of all its shards $C = \sum C_i$.

The cost of cross-shard transactions is higher than the cost of intra-shard transaction; we denote $c(t_i)$ the cost of transaction t_i. The exact cost depend on the consensus protocol as well as the cross-shard agreement protocol. The cost of each cross- and intra-shard transaction depends also on its size $c(t_i) \propto size(t_i)$. The larger the transaction, the longer it takes to propagate the information to all concerned nodes[4].

To process a transaction, a shard needs to spend some of its capacity equal to the cost of the transaction. For an intra-shard transaction t_i, shard i spends $c(t_i)$ and is left with a capacity $C_i = C_i - c(t_i)$. For an cross-shard transaction each concerned shard spends the cost of an cross-shard transaction; so the transaction can only be processed if all shards have enough capacity to process it during this time period.

State migration. Shard Scheduler migrates objects between shards. When object o_i is migrated from shard z_j to z_k, $m_{j \to k}(o_i)$ the shard assignment service ϕ is updated accordingly. Similarly to transactions, state migrations also have a cost for all involved shard that depends on the size of the migrated object $c(m_{j \to k}(o_i)) \propto size(o_i)$.

6 OBSERVATIONS

We start by investigating the transactions in the Ethereum blockchain from the perspective of a sharded operation. Our observations motivate the design of Shard Scheduler. For each transaction, we extract all the accounts whose state was modified. Details on data extraction are presented in Section 9.1.

O1. Write-oriented. In a blockchain, one can securely read the state from any honest participant. In contrast, writing to the blockchain is complex, because the data must be propagated to every single miner and agreed on using consensus protocol. In this work, we focus on writing state to the blockchain.

[4]We determine the exact cost of transactions for Chainspace [1] in later sections.

O2. Hot Spots. The activity of accounts can vary significantly (Figure 2). The top 20% accounts (*e.g.* popular exchanges) are responsible for over 92% of overall transactions. In the context of sharding, the most active accounts should not all be placed in the same few (or unique) shard(s).

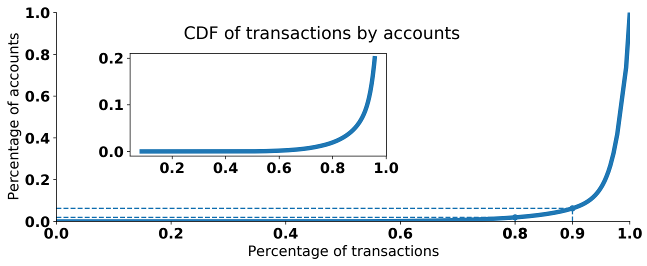

Figure 2: CDF of the number of transactions all the observed accounts were involved in.

O3. Communities. Multiple works reported accounts forming communities, *i.e.* groups of entities that interact frequently with each other [6, 33]. While the communities change over time, preserving them can significantly increase performance of a sharded blockchain due to the reduced number of cross-shard transactions [15].

O4. Load spikes. To maximize the throughput of the system, each shard should utilize its full capacity. Accounts in Ethereum experience bursts of activities caused by the market (*e.g.* Initial Coin Offerings, new tokens being added to exchanges) and "follow the sun" cyclical workload. We investigate the load shard of hash-based shard-to-account allocation (Figure 3). We observe significant differences in shard load, especially for shorter periods of observation. Without account migrations, a sharded blockchain is not able to fully utilize its capacity. The problem becomes more pronounced with increasing number of shards.

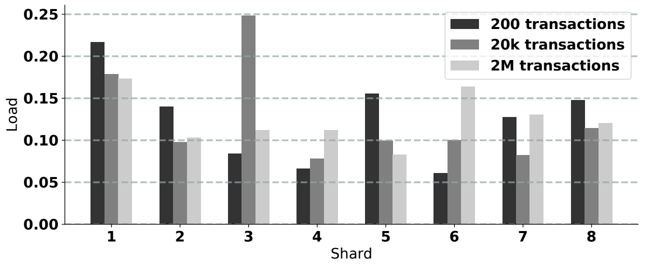

Figure 3: Shard load for hash-based account allocation for different periods of observation, when using 8 shards.

O5. Migrating state during inter-shard transaction is cheap. Under the model presented in Section 5, the cost of EOA migration is equivalent to the cost of an inter-shard transaction[5]. When two accounts spread across two shards are involved in a cross-shard transaction, one of the accounts can be migrated towards the other one replacing a cross-shard transaction by a migration and an intra-shard transaction. The intra-shard transaction will be processed only by the shard that hosts the accounts after the migration and does not generate additional overhead to the other shard.

O6. Inactive accounts. Accounts in blockchain are easy to create and are not constantly active. As of April 2020, the number of accounts exceeds 85 Millions, growing at a rate of about 50 to 150 thousands new accounts per day [12]. However, only 3% and 5% accounts are active within one-week and one-month observation periods, respectively. A newly created addresses is used, on average, for 35.45 days before going inactive [8]. At the same time, active accounts are likely to be updated soon after they are updated. An account is updated in a day from its previous activity with 62% probability [21]. We can say that only a fraction of accounts are active at any point of time, but once they are activated, they are likely to be accessed again soon (temporal locality). Inactive objects do not take part in new transactions and should not be migrated between shards even if they are highly connected with active objects. A migration of an inactive object involves a costly inter-shard agreement, does not decrease the state held by the input shard and increases the state held by the output shard without bringing any benefits.

O7. Smart Contracts. Smart contract migration is a complex process [14]. A migration of a Smart Contract requires creating a snapshot of its current state, locking it in the input shard and its recreation in the output shard [14]. However, the process of creating the snapshot is complex and there are currently no efficient mechanisms to perform it. At the same time, the migration cost depends heavily on the size of the snapshot. In contrast to EOAs, the state size of smart contracts can be significant.

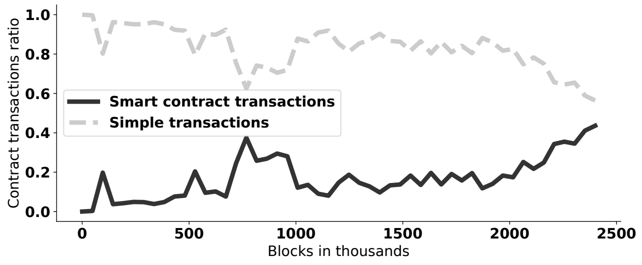

Figure 4: Simple and smart contract transactions over time.

O8. Internal Transactions. Internal transactions constitute a growing part of all the transactions in Ethereum. Figure 4) presents the percentage of transactions per type. Ordinary user-to-user transfers are on a solid downward trend. Contract transactions take up to 45% of the recent blocks in our sample. While majority of transactions modify the state of 2 accounts (EOA balances or internal state of smart contracts), some transactions modify up to 50 accounts at a time. Finally, the average number of accounts modified by an average transaction is on the rise over time caused by increased usage of smart contracts.

7 SHARD SCHEDULER DESIGN

Shard Scheduler is implemented as a part of the consensus protocol on all the miners of the blockchain. For each external cross-shard transaction, our scheduler operates in two steps:

(1) It determines the main shard and decide of placement of new accounts;

[5]A cost of of smart contract migration is proportional to its size.

(2) It decides on the migration(s) of existing account(s) towards the main shard.

We describe both steps in subsections below. Shard Scheduler does not migrate any accounts not involved in pending transactions (**O5. Migrating state during inter-shard transaction is cheap**) thus avoiding costly migrations that will not bring benefits in the future (**O6. Inactive accounts**). The main shard selected during the first step is then used during the second step. Only the main shard will be considered as a potential migration destination.

7.1 Data structures

Shard Scheduler miners associate an alignment vector

$$v_i = [a_{i1}, a_{i2}, ..., a_{in}]$$

with each account (including EOA and CA) in the blockchain where a_{ij} represents the *alignment* of account i towards shard j. The *alignment* is a positive integer and represents the total cost of transactions the account performed with the specific shard. When an account is created, the alignment vector values are set to 0. When account acc_i in shard z_i is involved in a transaction t_k with account acc_j in shard z_j, the respective values of both alignments vectors will be increased by the cost of t_k, so that $a_{ij} += c(t_i)$ and $a_{ji} += c(t_i)$. Importantly, a_i will not be updated when acc_j migrates between shards (and vice verse) simplifying the operation.

The alignment vector implements a sliding window approach and takes into account transactions from the last 100 blocks. It allows Shard Scheduler to better react to sudden burst of account activity (**O4. Load spikes**) and reduce memory overhead, as empty vectors can be dropped from the memory. Due to the large number of inactive accounts (**O6. Inactive accounts.**), Shard Scheduler maintains alignments vectors for a small fraction of the accounts at a time[6].

The alignment vector is held locally by each miner allocated to the shard where the account resides. It does not introduce any memory overhead to miners outside the shard and does not require storing any additional information on chain. The alignment vector is dropped (zeroed) when an account is being migrated between shards.

The second Shard Scheduler data structure is maintained on the beacon chain and represents the load of each shard in the system. The load for shard z_i is a positive integer that holds total cost of transactions processed by the shard during last 100 blocks. Similarly to the alignment vector, implementing a sliding window approach improves Shard Scheduler reactivity to sudden load changes. The load is reported by shards when submitting their block headers to the beacon chain and is certified by the shard-specific miners. Placing the load information on the beacon chain, makes it available to all the miners in the system.

7.2 Determining the main shard

The first step is performed by the transaction coordinator. Shard Scheduler takes as input a new a list of accounts being modified by the transaction l_i, the shard assignment function ϕ and the last state of the blockchain s_i(as defined by the previous block on the beacon chain). The list is known to the coordinator and includes all the internal transaction caused by the external one (**O8. Internal Transactions**). Based on this information, Shard Scheduler outputs allocation recommendations for new accounts (that appear on the blockchain for the first time) and a *main shard* for the transaction.

Based on the list of accounts and ϕ, Shard Scheduler starts by creating a set of shards involved in the transaction. Consider the smart contract from Algorithm 1, and a transaction t_j invoking the *payAll* function. The list of accounts l_j includes EOA of the caller, CA of the contract and accounts that from the *users* mapping that have less than 10 coins[7].

If the set of shards involved in the transaction is not empty, Shard Scheduler then reads the load of each involved shard from the beacon chain and chooses the least loaded one as the *main shard* for this transaction. If the set of shards involved in the transaction is empty[8], our scheduler chooses the least loaded shard from all the shards.

Shard Scheduler assigns all the new accounts from l_i to the main shard. The main shard identifier is then passed to shards holding non-new accounts involved in the transaction. The whole procedure for selecting the *main shard* is illustrated by Algorithm 2.

Algorithm 2 Main shard selection

```
1: procedure SELECTMAINSHARD($l_i, \phi, s_i,$)
2:     $involvedShards \leftarrow$ set()
3:     $newAcc \leftarrow$ new accounts from $l_i$
4:     for acc in $l_i$ do
5:         $involvedShards.add(\phi(acc))$
6:     if $involvedShards.empty()$ then
7:         $mainShard \leftarrow lowestLoad(allShards)$
8:     else
9:         $mainShard \leftarrow lowestLoad(involvedShards)$
10:    for acc in $newAcc$ do
11:        $\phi(acc) \leftarrow mainShard$
12:    return $mainShard$
```

The main shard selection is based uniquely on the load shards. It allows Shard Scheduler to migrate accounts to the least loaded shard performing load balancing (**O2. Hot Spots**).

7.3 Deciding migrations of existing accounts

The second step takes as input an account acc_i involved in a cross-shard transaction, the shard assignment function ϕ and transaction-specific *main shard* determined in the first step. The procedure is invoked only by miners associated with shard z_i, where acc_j resides so that $\phi(acc_j) = z_j$. Importantly, the procedure does not require any external (from other shards) data and can be performed within the shard in question.

From the local state, Shard Scheduler consults account alignment vector and extract account's alignment towards its current shard and the main shard. If the alignment towards the current shards multiplied by the cost of the cross-shard transaction is smaller than the sum of alignments towards the other shards, the account will be migrated to the *main shard*. Otherwise, the account remains in its current shard.

Taking into account the alignment vector stops the load balance-based migration if the account has strong connection with its current

[6]We further show the memory overhead in Section 9.

[7]Assuming that the contract has enough money to pay all the accounts

[8]This may happen if the transaction modified the state of new accounts - e.g., a coinbase operation.

shard. Such an approach preserves existing cluster of frequently interacting accounts (**O3. Communities**). The condition is more likely to stop the migration with increasing cost of cross-shard transaction. The whole procedure deciding on migrations is illustrated by Algorithm 3. For consensus protocols requiring all the accounts to reside in a single shard before processing (*i.e.* mutex-based), Shard Scheduler always migrates all the involved accounts to the main shard.

Algorithm 3 Migration decision algorithm.

```
1: procedure SHOULDMIGRATE($acc_i, \phi, mainShard$)
2:     $V \leftarrow$ the alignment vector for $acc_i$
3:     if $c(crossShard)V[acc]) < (sum(V) - V[acc])$ then
4:         $migrate(acc_i, mainShard)$
```

8 ECONOMICS

Maintaining a blockchain requires resources to store (disk space), exchange (network bandwidth) and verify (CPU cycles) transactions. In open systems, miners are incentivized to perform this useful work in exchange for a financial reward. Incentive mechanisms for open sharded blockchains is currently a gap in the blockchain literature [3]. We argue that naively applying incentives mechanisms from traditional (single-committee) blockchains to sharded systems has shortcomings, and then propose a novel design to fix them.

Purpose of the incentive mechanism. The purpose of the incentive mechanism is to motivate rational miners to follow the protocol. In the absence of externalities (*e.g.* secondary markets), it ensures that miners following the protocol collect a higher financial reward than if they were deviating from it. The main purpose of Shard Scheduler is to increase the performance of the blockchain. We require, therefore, an incentive mechanism that also goes in that direction: miners should be incentivized to follow the recommendations of Shard Scheduler.

Traditional incentive mechanism. Starting from Bitcoin, incentive schemes [13, 28, 40] typically involve collecting transaction fees from the end-user. The leader of the consensus protocol collects all the fees associated with the transactions it proposes; this leader is thus often rotated following system-specific strategies. Users are free to offer any fee for processing their transaction. Rational miners prioritize high fee transactions when constructing their blocks to maximize their financial reward.

A naive extension of the incentive mechanism described above could work as follows. User associate transactions fees as in single-committee systems. These fees are shared amongst the leaders of the intra-shard consensus protocol of every shard involved in the transaction. A similar incentive mechanism is adopted by Zilliqa [36].

We argue that directly applying this mechanism to a sharded environment does not incentivize rational miners to maximize the system's performance. We show that, if given the right to perform account migrations, miners financially benefit from taking actions that harm the total system performance by creating imbalances between shards.

LEMMA 8.1. *In the sharded environment described in Section 4, rational miners financially benefit from concentrating as many accounts as possible into their shard.*

PROOF. Miners are periodically elected as leaders according to the intra-shard consensus protocol and propose new blocks. When acting as the leader, rational miners choose the clients' transactions to include in their next proposal by selecting those with the highest fees. They can however only include transactions involving accounts in their own shards: these transactions are by definition a subset of the total transactions submitted to the system (for any epoch). As a result, miners have less options to select high-fees transactions than if they could choose amongst all transactions. To increase the number of transactions that involve their shard, and thus increase their choice of transactions, miners are motivated to concentrate a large portion of accounts in their shard. □

Lemma 8.1 indicates that rational miners may financially benefit from actively resisting optimal placement recommendations, which may worsen the system performance.

Adapting the model for sharded blockchains. To overcome the shortcoming presented above, we propose an alternative solution that decouples the process of collecting transactions fees from cashing them in. We leverage the fact that miners are randomly assigned to shards, and thus cannot predict which shard they will integrate next (Section 4). The incentive mechanism operates across every two consecutive epochs:

- During epoch n, miners collect the fees of transactions that involve their shard and lock them into a shard-special deposit (as opposed to adding them to their private accounts). This deposit keeps a fine-grained accounting of the fees that each miner of the shard collected during the epoch. We follow classic incentive mechanisms and attribute the transactions fees to the current leader of the consensus protocol.
- Upon epoch change, miners are randomly shuffled and reassigned to other shards. Upon entering the next epoch $(n+1)$, miners cash in the transaction fees deposited into their new shard's deposit, as a pro-rato of their contribution during the previous epoch.

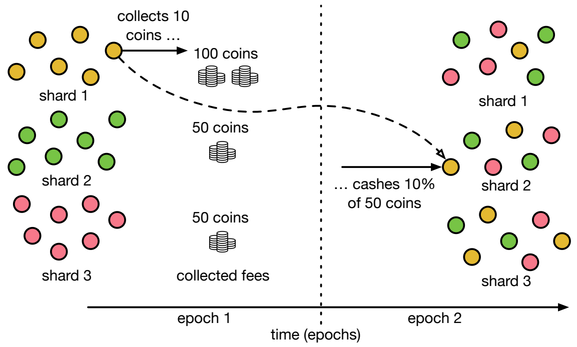

Figure 5: Incentive model for Shard Scheduler.

Consider a scenario with 3 shards: z_1, z_2 and z_3 and miner $m_1 \in z_i$ for epoch e_n (Figure 5). During epoch n, shard z_1 collects a total of

100 coins in transaction fees, z_2 collects 50 coins, and z_3 collects 50 coins as well. These fees are locked in their respective shard's deposit; that is, the deposit of z_1 holds 100 coins, and the deposits of z_2 and z_3 hold 50 coins each. No miners have access to these deposits for the time being. Let's say that miner m_1, when acting as leader, proposed transactions containing a total of 10 coins of fees during epoch n. That is, we attribute 10% of the total fees collected by z_1 during e_n to miner m_1. During epoch $n + 1$, m_1 is assigned to shard z_2. Upon entering the epoch, it cashes in 10% of the deposit accumulated by z_2 during epoch n. That is, m_1 cashes in 5 coins.

Effectiveness analysis. We argue that our proposed incentive scheme incentivizes rational miners to increase the total system's capacity.

LEMMA 8.2. *Each epoch n, the expected reward of miners is proportional to the total transaction fees collected in the system x_{tot}^{n-1} during the previous epoch.*

PROOF. The expected reward of a miner during epoch n is $E^n(x) = \sum_{i=1}^{k} x_i^{n-1} p_i$, where x_i^{n-1} is the total reward collected by shard i during epoch $n - 1$, p_i is the probability that the miner ends up in shard i in epoch n, and k is the total number of shards in the system. Since miners are randomly assigned to shards, $\forall i, j, \quad p_i = p_j = \frac{1}{k}$. Thus $E^n(x) = \frac{1}{k} \sum_{i=1}^{k} x_i^{n-1} = \frac{1}{k} x_{tot}^{n-1}$. □

LEMMA 8.3. *The total fees collected in the system x_{tot} increases with the total capacity of the system C.*

PROOF. As described in Section 5, we assume that the shards' processing capacity C_i is a scarce resource and that clients transactions are abundant. As a result, if the shards' capacity increases, they can process more transactions per epoch and thus collect more fees. This implies that the fees x_i collected by shard i increases with the shards' capacity C_i. We can thus express x_i in terms of C_i as a monotonically increasing function: $x_i(C_i)$.

The total fee collected in the system is defined as $x_{tot} = \sum_{i=1}^{k} x_i$. We can thus write $x_{tot} = \sum_{i=1}^{k} x_i(C_i)$ to show that the total fees x_{tot} increases with the shards' capacity C_i.

Section 5.2 defines the total capacity of the system as the sum of the capacity of every shard: $C = \sum_{i=1}^{k} C_i$, which means C increases with $\{C_i\}_{i=1}^{k}$. Combining those observations, we have that both x_{tot} and C increase with the shards' capacity $\{C_i\}_{i=1}^{k}$. It follows that the total collected fees x_{tot} increases with the total system's capacity C: x_{tot} and C are positively correlated. □

LEMMA 8.4. *The expected reward of miners increases with the total system's capacity.*

PROOF. Lemma 8.2 implies that the expected reward of miners increases with the total fees collected in the system. Lemma 8.3 shows that the total fees collected in the system increases with the total system's capacity. Therefore, the expected reward of miners increases with the total system's capacity. □

9 EVALUATION

We provide details on our data set as well as the setup and results of our simulations.

9.1 Data Extraction

We download first 2M blocks Ethereum transaction history (1 year). We extract 8M non-coinbase transactions and all the accounts that were modified during each transaction. We use openethereum v3.2.3[9] operating in archive mode that allows to recompute all the intermediary states of the blockchain. To extract the transactions and state modifications, we create a Python tool based on web3.py[10] that queries the client with *trace_replayTransaction* calls in *stateDiff* mode. We made the code and the dataset publicly available to the scientific community [11].

9.2 Setup

We implement a Python-based simulator to evaluate the effectiveness of our approach. The simulator closely follows the model presented in Section 5, operates in rounds and takes transactions (extracted in Section 9.1) as the input workload. Before the first round, the simulator fills up the mempool with transactions from the input workload, and in the beginning of each subsequent round, the simulator tops up the mempool from the input workload.

The size of the mempool is fixed and set up using simulation parameters. The transactions are processed in the order of arrival by the blockchain. The policy being evaluated indicates placement and in the case of Shard Scheduler, migration of objects. Each transaction increases the load of one or multiple shards. A transaction can be processed in the current round, only if there is enough processing capacity left in all the involved shards. The unprocessed transactions will remain in the mempool and be processed during subsequent rounds. The simulator reports the following performance metrics:

- **Throughput** - the global throughput of the entire blockchain in terms of the number of transactions per block.
- **Latency** - the average elapsed time to complete the processing of the transactions in the workload. We measure the elapsed time to complete a transaction in terms of number of rounds (blocks), *i.e.* from the round when a transaction is initially read until the round when a transaction is added to the blockchain.
- **Wasted Capacity** - the load-balancing performance of the system in terms of *residual capacities* of the shards summed over all the rounds. Residual capacity of the shards is sum of unused capacity of the shards at the end of a round.
- **Cross-shard transaction ratio** - the percentage of transactions that involve accounts from multiple shards. For Shard Scheduler, each migration is accounted as a separate cross-shard transaction.

We compare Shard Scheduler against hash-based and metis policy. The hash-based policy represents an approach used in existing sharded blockchains [1, 13, 22, 41], assigning accounts to shards based on their identifier and does not perform migrations. The Metis policy is a *hypothetical* one that reads all the transactions in the beginning of the first round, at once and proactively performs sharding using the well-known metis graph partitioning (a.k.a. community detection) algorithm [19] on the transaction graph, whose nodes

[9]https://openethereum.org/

[10]https://github.com/ethereum/web3.py

[11]Omitted for submission.

are individual accounts and edge weights indicate the number of transactions between the accounts [9].

The metis algorithm computes a desired number of "balanced" partitions, each corresponding to a shard, on an input transaction graph—the objectives of partitioning are to minimise the total weight of inter-partition edges (*i.e.* minimising inter-shard transactions) and to minimise variance across partitions in terms of their total of intra-partition edge weights (*i.e.* achieving similar number of intra-shard transactions in each shard). We do not compare against UTXO solutions such as OptChain [29] due to data model incompatibility.

We verify the impact of the following parameters on the performance:

- **Number of shards** - we vary this parameter from 1 to 60 shards, and set its default value to 16 shards. Higher number of shards means increased processing capacity, but also more cross-shard interactions and load balancing challenges.
- **Cross-shard transaction costs** - we assume a fixed cost of all the cross-shard transactions and measure the impact of changing this cost from one (as costly as intra-shard transaction) to ten. The actual cost depends on the consensus protocol and its implementation. We set the default value to 2 observed for Chainspace in Section 10. This parameter also impacts the cost of migrations performed by Shard Scheduler. We do not migrate smart contract accounts due to difficult to determine migration cost.
- **Shard processing capacity** - we investigate the impact of modifying the processing capacity of a single shard. We set the default value to 200 (*i.e.* 200 intra shard transactions per block) as observed for Ethereum [12].
- **Mempool-to-capacity ratio** - we express the mempool size in terms of the ratio of processing capacity of the entire blockchain per block. The mempool size of the system in practice depends on the rate of transaction submissions (*i.e.* rate of arrival to mempool buffer) and processing speed (*i.e.* rate of departures) of the blockchain.

In each experiment, we only vary one system parameter while the rest of them take their default values. We start by measuring the performance of all the policies in terms of throughput and latency and later explain the results by observing *wasted capacity* and *cross-shard transaction ratio*.

9.3 Results

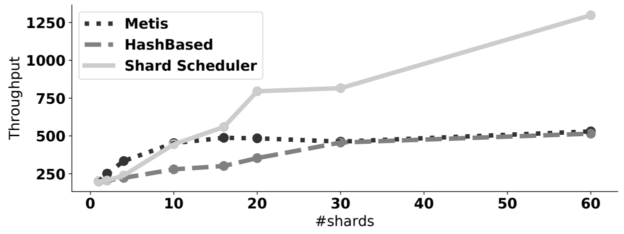

Figure 6: Throughput vs number of shards.

Throughput. Figure 6 shows the impact of the *number of shards* on the throughput. We observe that Shard Scheduler achieves increasingly better throughput as the number of shards increases. On the other hand, the throughput of both metis and hash-based policies flatten out with increasing number of shards. Shard Scheduler improves the throughput by 100% for 16 shards and by 250% for 60 shard over the hash-based approach. Shard Scheduler also outperforms the theoretical metis policy, which uses future transaction information, for more than 10 shard and achieves similar performance for lower values.

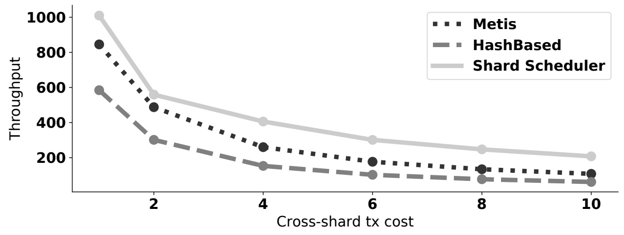

Figure 7: Throughput vs cross-shard transaction cost.

Figure 7 shows the impact of varying *cross-shard transaction costs* on the throughput. Higher processing cost results in lower throughput. For all the costs, Shard Scheduler achieves the highest throughput. The Shard Scheduler performance gain remains steady over both hash-based (80-95% throughput increase) and metis (10-40% throughput increase) policies.

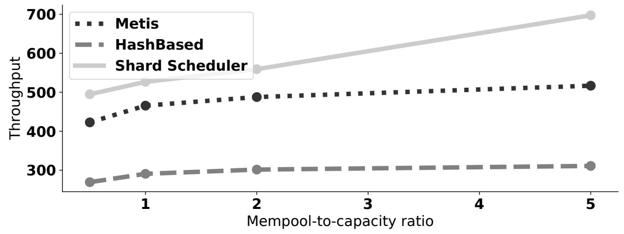

Figure 8: Throughput vs mempool size.

In Figure 8, we vary the *mempool size* as multiples of shard processing capacity. Larger mempool size allows to achieve better load balancing and improve the throughput of all the policies. However, the impact of an increased mempoll size on hash-based and metis policies is limited. Shard Scheduler achieves 80% throughput inceease for 0.5 mempool-to-capacity ratio and 130% throughput increase for the ratio equal to 5.

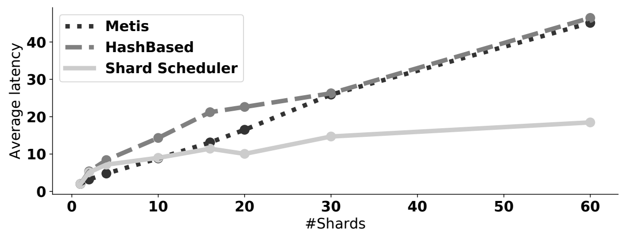

Figure 9: Average latency vs number of shards.

Latency. In Figure 9, we observe average processing latencies with increasing number of shards. Shard Scheduler higher throughput result in significantly lower latency (3.5 times lower than other policies for 60 shards). Surprisingly high latency of the metis policy is caused by unequal load allocation (as shown below).

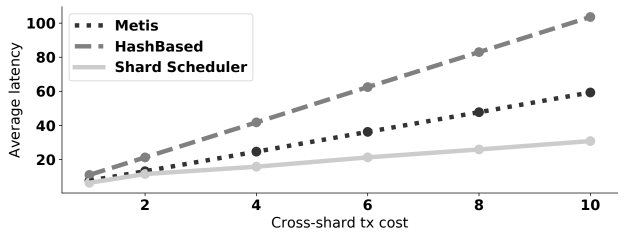

Figure 10: Average latency vs cross-shard transaction cost

Increasing the cross-shard transaction cost (Figure 10) increases the latency for all the policies. The metis policy preserves the account communities and performs better than the hash-base policy with the increasing cost of cross-shard interactions. However, Shard Scheduler achieves 2 times lower latency than metis and 3 times lower than hash-based policy for 10 cross-shard transaction cost.

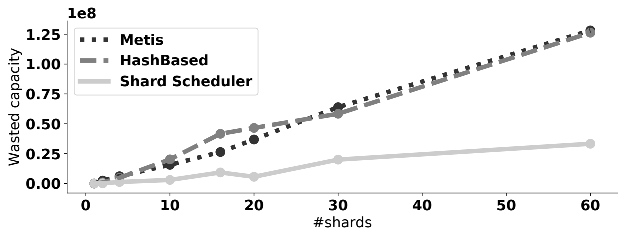

Figure 11: Wasted capacity vs number of shards.

Wasted Capacity. Both metis and hash-based policies achieve equal load spread across the shards in the long run. However, they fail to adapt to fine-grained activity changes due to the lack of migrations. Shard Scheduler takes per-transaction migration decision based on the previous load of all the shards and better utilizes the capacity of the blockchain. This effect is more pronounced as the number of shards (Figure 11) or the cost of cross-shard transactions (Figure 12) increases. More cross-shard interactions or their increased cost translates into more transactions waiting for one of the involved shards to become available.

Cross-shard Transaction Ratio. Finally, we observe in Figure 13 that Shard Scheduler is able to adapt gracefully to increasing inter-shard costs and reduce its number of cross-shard transaction ratio. This reduction is caused by the migration stopping condition (Algorithm 3) which takes the cross-shard transaction cost into account.

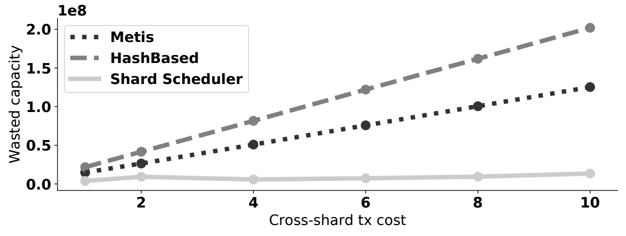

Figure 12: Wasted capacity vs cross-shard transaction cost.

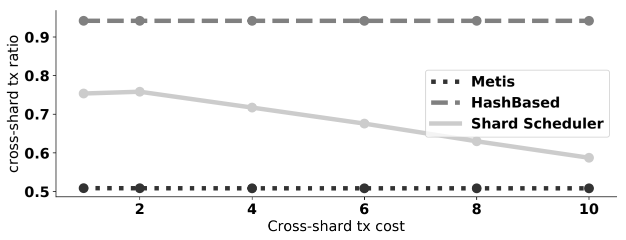

Figure 13: Cross-shard transaction ratio vs cross-shard transaction cost.

On the other hand, both metis and hash policy are both oblivious to cross-shard transaction costs and their cross-shard ratios remain roughly constant failing to adapt to the changing environment.

Overall, we observe that Shard Scheduler achieves significantly better performance despite scheduling additional cross-shard transactions linked to account migrations. The short-term migration overhead is largely compensated by long-term advantages of better load-balancing and preserving account communities.

10 PROTOTYPE

In this section we confirm the simulation results with real-world experiments.

Setup. We implement Shard Scheduler, metis and hash-based policied on top of Chainspace [1] with security improvements proposed by Byzcuit [32]. Other shard environments are not yet finished [13], do not provide the source code [39, 41] or does not fully partition the state [25]. By default, Chainspace implements a UTXO data-model and does not implement blocks (*i.e.* transactions are serialized as a continuous flow). We thus add blocks implementation and a data-model translation module that allows us to replay the history of Ethereum (Section 9.1) with an equivalent number of intra-shard and cross-shard transactions. We make the block implementation coherent with our model presented in Section 5 and publish the code[12]. We deploy 3 miners per shard on Amazon AWS within a single data centre and run tests for 5 and 10 shards. Due to high result variation within a single run, we repeat the tests 5 times and report the average values.

We create 2 synthetic workloads of 1M transactions containing uniquely: *(i)* intra-shard transactions and *(ii)* cross-shard transactions. Both workload create perfectly balanced load across all shards. For the second workload, we observe 2 times lower throughput than for the first workload. We thus assume the cost of cross-shard transactions to be 2 for Chainspace and use it as a parameter to Shard Scheduler (Section 7).

Results. We start by measuring the throughput of the system reported by Chainspace as the transaction per second (TPS) rate (Figure 14). Surprisingly, we observe almost no throughput improvement for the hash-based policy when increasing the number of shards from 5 (55TPS) to 10 (56TPS). This is caused by highly unequal load across shards. For 10 shards, we observe multiple blocks filled to less than 50% of their capacity. The metis policy provides much higher throughput (123TPS), but also suffer from unequal per-block load. The performance of the metis policy is expected to further drop down

[12]Ommited for submission.

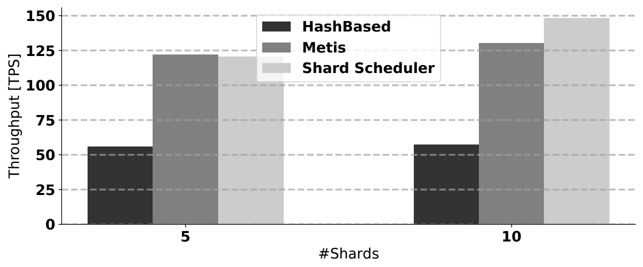

Figure 14: Chainspace throughput.

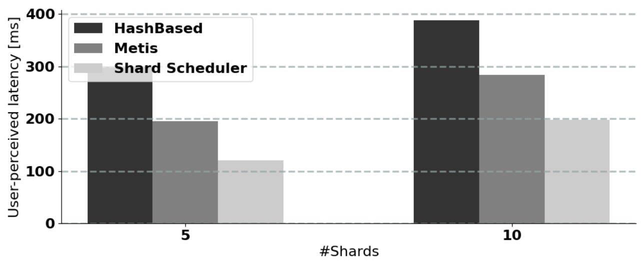

Figure 15: Chainspace latency.

when increasing the size of the input file. Shard Scheduler is the only policy experiencing a significant throughput improvement and, for 10 shards, it triples the TPS rate of the hash based policy. However, the TPS rate when doubling the number of shards increases by only 23%(from 120TPS to 148TPS). This is caused by migrations and cross-shard transactions that cannot be fully eliminated. Especially with complex smart contract transactions involving large numbers of accounts.

We continue by investigating the transaction latency as perceived by the end-users (Figure 15). Similarly to the simulations, the number of transactions submitted per block (*i.e.* the mempoll) is proportional to the per-block capacity of the entire blockchain. Without the linear increase of the throughput, this approach causes increase of the user-perceived latency (as more blocks are need to fully process the mempoll). However, we observe the average latency achieved by Shard Scheduler to be significantly lower than for both the hash-based policy (49% reduction for 10 shards) and the metis policy (31% reduction for 10 shards).

11 RELATED WORK

We review related work on object migration and placement for sharded blockchain. We then briefly discuss object management techniques from the area of distributed systems.

Object migrations and allocation. Optchain [29] proposes an oracle for optimal transaction placement in sharded blockchains. The system uses graph clustering techniques and is implemented as an external service for the clients. However, Optchain approach works only for UTXO blockchains and cannot be easily adapted to the account-based data model. Han *et al.* [17] study existing shard allocation protocols and propose WORMHOLE, a shard allocation protocol taking into account both self-balance and operability. However, the study focuses on allocating miners to shards, rather than objects residing on the blockchain. Fynn *et al.* [15] analyze the history of Ethereum transactions and investigate multiple graph clustering protocols in the context of account placement in sharding. Similarly to our observations, they show that proactive placement without periodic migration does not achieve optimal performance. Fynn *et al.* [14] develop techniques for moving smart contracts between shards and blockchains *de facto* enabling contract migrations. The authors implements their protocol on Ethereum [40] and Burrow [18].

Distributed systems. In the area of the distributed systems multiple works investigated optimal object assignment and migrations. The proposed systems focus on two main aspects: *(i)* developing a partitioning/migration plan (*i.e.* object-to-partition allocation) and *(ii)* efficient plan execution guaranteeing safety without causing significant downtime.

E-store [35] provides an efficient solution based on tuples monitoring and bin backing problem to compute an optimal assignment of object to partition. However, the system does not take into account data locality. Clay [30] balances the number of inter-partition transactions, load balancing and limiting the number of migrations in order to maximize the throughput of the system. P-store [34] creates a partition plan taking into account load only. It contains a traffic prediction module [5] that can proactively scale up or down the entire platform.

Squall [11] and Mgcrab [24] implement systems for efficient object partition and migration once given a partition plan. The platforms proposed for distributed systems provide important insights useful in our designs. However, they cannot be directly applied to sharded blockchains due to a different governance model. The majority of the platforms contain a non-deterministic element or cannot be verified by third parties [34], introduce significant computational overhead [5, 30] or migrate large clusters of the objects at once [30].

12 DISCUSSION AND CONCLUSION

Shard Scheduler provides objects migration and placement recommendations for account-based sharded blockchains. It provides a number of desirable properties and achieves the design goals specified in Section 4.2. First of all, Shard Scheduler improves the overall throughput of sharded blockchains. This is achieved through the mechanism explained in Section 7 and its effectiveness is demonstrated by experiments (see Sections 9 and 10). Furthermore, Shard Scheduler recommendations are publicly verifiable as they are deterministic. Any third party can verify the correctness of objects migration and miners can apply the recommendations without needing an extra round of consensus. Shard Scheduler is lightweight in the sense that it does not require extra protocol messages, and does not introduce significant computation or memory overhead. It integrates seamlessly into existing protocols requiring only minimal changes to the miners' software, and does not impact the way clients use the system. Section 8 provides a novel incentive mechanism for sharded blockchain to financially motivate miners to maximize the total throughput of the system—miners collect higher fees by improving the overall performance of the system rather than by concentrating accounts in their own shard.

We leave a number of open questions that are deferred to future works. First of all, the objects placement recommendations of Shard Scheduler are efficient based on current and past typical usages of blockchains. There are no guarantees that this would be the case if blockchains are used in significantly different ways in the future. A learning agent may solve this issue by predicting future interactions between accounts, but it is not clear how to ensure that such agent remains both deterministic and lightweight. Secondly, handling transactions fees could become costly operations as they are associated with each transaction and may involve multiple shards. It would thus be desirable to remove fees handling from the critical path of the transaction's processing, or even offload them to a side-infrastructure. Recent works [4] [7] demonstrate that distributed payment systems can efficiently be implemented without consensus, and by quorum-based systems that can be natively integrated into one or more shards of a sharded blockchain.

ACKNOWLEDGEMENTS

Alberto Sonnino is supported by Novi, a subsidiary of Facebook. This work is partially supported by The Brussels Institute for Research and Innovation (Innoviris) under project FairBCaaS, and by the Belgian *Fonds de la Recherche Scientifique* (FNRS) under Grant #F452819F.

REFERENCES

[1] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[2] Mustafa Al-Bassam, Alberto Sonnino, and Vitalik Buterin. Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044*, 160, 2018.

[3] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936*, 2017.

[4] Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

[5] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI*, volume 8, pages 337–350, 2008.

[6] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. Understanding ethereum via graph analysis. *ACM Transactions on Internet Technology (TOIT)*, 20(2):1–32, 2020.

[7] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

[8] ConsenSys. Ethereum by the numbers. https://media.consensys.net/ethereum-by-the-numbers-3520f44565a9, 2018.

[9] Inderjit Dhillon, Yuqiang Guan, and Brian Kulis. A fast kernel-based multilevel algorithm for graph clustering. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 629–634, 2005.

[10] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[11] Aaron J Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 299–313, 2015.

[12] EtherScan. Ethereum charts and statistics. https://etherscan.io/charts, 2021.

[13] Ethereum Foundation. Ethereum 2.0 phases. https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/eth-2.0-phases/, 2019.

[14] Enrique Fynn, Alysson Bessani, and Fernando Pedone. Smart contracts on the move. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 233–244. IEEE, 2020.

[15] Enrique Fynn and Fernando Pedone. Challenges and pitfalls of partitioning blockchains. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 128–133. IEEE, 2018.

[16] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, 1978.

[17] Runchao Han, Jiangshan Yu, and Ren Zhang. Analysing and improving shard allocation protocols for sharded blockchains.

[18] HyperLedger. Burrow. https://github.com/hyperledger/burrow, 2021.

[19] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[20] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[21] Jae-Yun Kim, Junmo Lee, Yeonjae Koo, Sanghyeon Park, and Soo-Mook Moon. Ethanos: efficient bootstrapping for full nodes on account-based blockchain. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 99–113, 2021.

[22] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[23] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[24] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron Elmore, and Shan-Hung Wu. Mgcrab: transaction crabbing for live migration in deterministic database systems. *Proceedings of the VLDB Endowment*, 12(5):597–610, 2019.

[25] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[26] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[27] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. Bdos: Blockchain denial-of-service. In *Proceedings of the 2020 ACM SIGSAC conference on Computer and Communications Security*, pages 601–619, 2020.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, 2008.

[29] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. Optchain: optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535. IEEE, 2019.

[30] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[31] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. *arXiv preprint arXiv:1901.11218*, 2019.

[32] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 294–308. IEEE, 2020.

[33] Hanyi Sun, Na Ruan, and Hanqing Liu. Ethereum analysis via node clustering. In *International Conference on Network and System Security*, pages 114–129. Springer, 2019.

[34] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 205–219, 2018.

[35] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[36] The Zilliqa Team. Zilliqa whitepaper. https://docs.zilliqa.com/whitepaper.pdf, 2017.

[37] Zilliqa Team et al. The zilliqa technical whitepaper. *Retrieved Sept*, 16:2019, 2017.

[38] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. Sok: Sharding on blockchain. In *1st ACM Conference on Advances in Financial Technologies*, AFT, 2019.

[39] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 95–112, 2019.

[40] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[41] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.