Toward Reducing Cross-Shard Transaction Overhead in Sharded Blockchains

Liuyang Ren
University of Waterloo
Waterloo, Canada
l27ren@uwaterloo.ca

Paul A. S. Ward
University of Waterloo
Waterloo, Canada
pasward@uwaterloo.ca

Bernard Wong
University of Waterloo
Waterloo, Canada
bernard@uwaterloo.ca

ABSTRACT

Sharding is a promising approach to high-performance blockchains and has been extensively explored in academia recently. However, sharding also introduces cross-shard transactions, which require expensive inter-shard coordination to ensure state consistency. Such transactions significantly limit the performance of sharded blockchains.

To reduce cross-shard transactions in UTXO-based sharded blockchains, we propose Rooted Graph Placement, which identifies the most appropriate shard for a transaction based on the interaction between the transaction and historical transactions. In conjunction with the placement algorithm, we also devise two techniques to lessen the system performance impact of the remaining cross-shard transactions. One technique parallelizes dependent transaction verification with the atomic commit protocol, and the other consolidates the atomic commit protocol. Combining all the three techniques, we can improve the maximum system throughput by 118% when compared with a state-of-the-art transaction placement algorithm.

CCS CONCEPTS

• **Information systems → Distributed database transactions**; • **Computing methodologies → Vector / streaming algorithms**; • **Mathematics of computing → Graph algorithms**.

KEYWORDS

blockchain, sharding, cross-shard transaction, transaction placement, dependency graph

ACM Reference Format:
Liuyang Ren, Paul A. S. Ward, and Bernard Wong. 2022. Toward Reducing Cross-Shard Transaction Overhead in Sharded Blockchains. In *The 16th ACM International Conference on Distributed and Event-based Systems (DEBS '22), June 27-July 1, 2022, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524860.3539641

1 INTRODUCTION

Conventional blockchains suffer from low performance due to their need for every node in the chain to verify and execute all transactions. To improve performance, various designs have been proposed—e.g., shortening block intervals [4], incorporating off-chain blocks [20] [21], allowing one miner to consecutively propose multiple blocks [10], sharding [22] [18] [34] [5], journaling aggregated transaction effects to blockchains [6] [28], etc. Among these techniques, sharding is a very promising approach that has been extensively explored in academia and adopted by Ethereum 2.0 [8]. The high-level idea of sharding is to partition a system into independent shards and distribute workloads among shards for parallel processing, so that the system performance scales with the number of nodes.

However, because each shard usually stores a disjoint subset of the system state [18] [34] [5] [1] [15], transactions modifying more than one subset inevitably incur cross-shard communication. Moreover, since blockchains operate in trustless environments, expensive digital signatures must be employed to prove cross-shard message authenticity. The overhead makes cross-shard transactions consume more network and CPU resources than single-shard transactions. Nonetheless, as the most common transaction placement algorithm [22][18][34][5], hashing placement creates a huge number of cross-shard transactions, e.g., 95% of transactions are cross-shard in a 16-shard system. With so many cross-shard transactions, sharded blockchains can hardly approach their full potential.

Previous work on reducing cross-shard transactions either relies on additional trust points or applies only to account-balance blockchains. Thus this work proposes a novel transaction placement algorithm, namely Rooted Graph Placement (RGP), that is fully decentralized and applies to UTXO-based blockchains. We observed that transactions with (cascading) dependencies between them are more likely to be referenced together by future transactions than unrelated transactions. This is because transaction dependencies reflect the connections between the users who initiate the transactions. Users with connections are more likely to collaborate in the future than those without connections, especially considering that a user may control multiple identifies/addresses in a blockchain system and transfer cryptocurrencies between them. Based on this observation, RGP tends to place a transaction to the shard that includes most of the ancestor transactions so that future transactions referencing this transaction will have a better chance of executing within a single shard.

RGP can reduce cross-shard transactions but not eliminate them, so we also devise two techniques for efficiently processing the remaining cross-shard transactions. The first technique is dependent transaction pre-verification, which parallelizes the atomic commit protocol of cross-shard transactions with the signature verification of their dependent transactions. This design shortens the execution latency of the dependent transactions. The second technique utilizes the fact that RGP places most cross-shard transactions to one of their input shards. For such shards, the request for lock input

DEBS '22, June 27-July 1, 2022, Copenhagen, Denmark

ACM ISBN 978-1-4503-9308-9/22/06…$15.00
https://doi.org/10.1145/3524860.3539641

UTXO(s) and the request for generating output UTXO(s) can be merged into one message, which reduces signatures and consensus rounds involved in cross-shard transaction processing.

2 BACKGROUND

2.1 Unspent Transaction Output (UTXO) Model

Unlike conventional banking systems, cryptocurrencies like Bitcoin use a UTXO model to express their system states instead of the account-balance model. Accordingly, a transaction spends input UTXOs and generates output UTXOs. Figure 1 demonstrates the transaction execution in Bitcoin. Bob sends 1.5 BTC to Alice by creating a transaction that spends the 2-BTC UTXO, which belongs to Bob, and generates a 1.5-BTC UTXO for Alice as well as a 0.5-BTC UTXO for Bob. Once the transaction is executed, the 2-BTC UTXO (i.e., *UTXO B* in Figure 1) does not exist anymore. Every transaction consumes some input UTXO(s), except for coinbase transactions, which spend nothing and credit output UTXOs to miners.

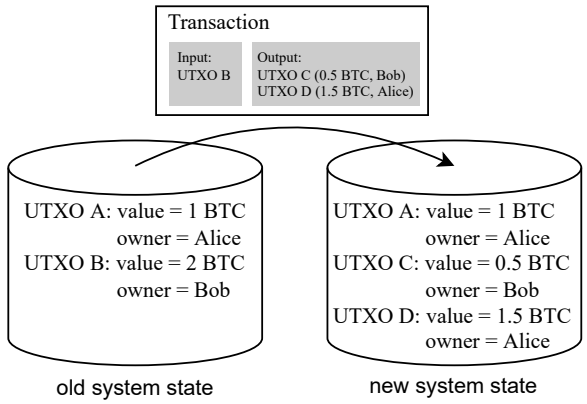


Figure 1: Transaction execution

Given a transaction, the transaction placement problem in a UTXO-based system means finding a shard to process the transaction and store the output UTXOs. This shard is called the *output shard* of this transaction. Similarly, a shard storing at least one input UTXO of this transaction is called an *input shard*. All output UTXOs of a transaction should be stored in the same shard because they are likely to be consumed together in the future. Therefore, every transaction has only one output shard.

2.2 Blockchain Sharding Protocols

In this section, we first describe the sharding protocol OmniLedger [18] and then briefly introduce other sharding protocols. OmniLedger is described in detail because it will be used for evaluating the techniques proposed in this paper. This is consistent with a related work called OptChainV2, which also employs OmniLedger in its evaluation.

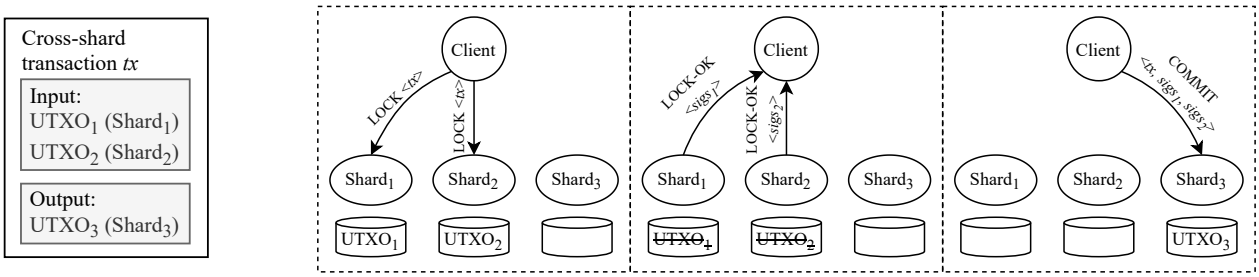
OmniLedger partitions peers in the blockchain network into concurrently operating shards that maintain their respective ledgers and states. Shard members are selected based on an unbiased random number and periodically reconfigured so that slow-adaptive attackers cannot corrupt the blockchain network by attacking a shard. Within a shard, peers run the Practical Byzantine Fault Tolerance (PBFT) protocol [3] to agree on the order of transactions. To ensure state consistency between shards, cross-shard transactions rely on an atomic commit protocol to be either unanimously committed or unanimously aborted. The problem of guaranteeing transaction atomicity dates back to the late 1970s [19][12]. Among various protocols, the two-phase commit (2PC) protocol [19] is most widely used [13][31]. OmniLedger adopts the "two-phase" concept and invents *Atomix* for sharded blockchains. In trustless environments, it is challenging to find an atomic commit protocol coordinator, whose misbehavior may lead to forever-locked UTXOs. OmniLedger utilizes clients as the coordinators of their own transactions so that coordinators are incentivized to conform to the protocol.

The basic idea of Atomix is shown in Figure 2. A client requests the input shards to lock the input UTXOs (i.e., mark the UTXOs as spent), and the input shards respond with signed lock results. If all input shards reply with positive lock results, the client then sends the output shard a `COMMIT` request along with signatures from the input shards as proof of successful locking. Upon receiving the `COMMIT` request, the output shard adds the output UTXO(s) to its system state, provided that all signatures of the input shards are valid. If any input shard fails to lock an input UTXO, the client requests the other input shards to unlock their respective input UTXOs with the signed response from the fail shard as proof of unsuccessful execution.

Other blockchain sharding protocols include Elastico [22], RapidChain [34], AHL [5], SharPer[1], etc. Elastico, AHL, and SharPer also employ PBFT as their intra-shard consensus protocol, whereas RapidChain uses a variant of a synchronous consensus protocol [29] to tolerate the same number of faulty peers with a smaller shard size. In terms of atomic commit protocols, RapidChain processes a cross-shard transaction by splitting it into multiple sub-transactions, each of which spends UTXOs that reside in one shard (i.e., every sub-transaction has only one input shard). AHL incorporates 2PC and leverages an entire Byzantine fault-tolerant committee as the coordinator. SharPer[1] claims that AHL cannot process cross-shard transactions in parallel due to the single coordinator committee, so it utilizes individual peers as coordinators instead. Specifically, every node serves as its own coordinator by exchanging messages with the other nodes in involved shards and deriving the commit decision locally. Such decentralized approaches have also been explored by Cerberus [14] (a series of cross-shard transaction processing protocols) and Byshard [15] (a framework for the study of sharded resilient systems). However, decentralized coordination usually incurs high message complexity. Elastico is fundamentally different from other sharding protocols in that it does not partition the system state. As a result, there are no cross-shard transactions in Elastico. Table 1 summarizes the above sharding protocols.

3 RELATED WORK

The difference between UTXO-based blockchains and account-balance blockchains necessitates different approaches to reducing cross-shard transactions, i.e., transaction placement and account placement. Previous works handling both types of blockchains are reviewed in this section.



(a) A cross-shard transaction **(b) Atomix (commit scenario)**

Figure 2: An example of OmniLedger's Atomix protocol

Table 1: Blockchain Sharding Protocols

Protocol	Intra-shard Consensus	Atomic commit protocol	Coordinator	Transaction Placement
Elastico	PBFT	N/A	N/A	Hashing placement
OmniLedger	PBFT	Atomix	Client	Hashing placement
RapidChain	Synchronous BFT	Transaction splitting	PBFT leader of output shard	Hashing placement
AHL	PBFT	2PC	Dedicated BFT committee	Hashing placement
SharPer[a]	PBFT	decentralized protocol	decentralized	An involved shard

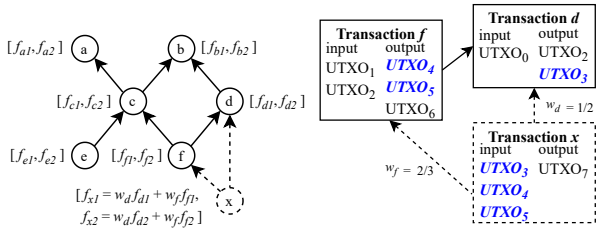
[a] SharPer supports networks consisting of either crash-only or Byzantine nodes. Here we consider SharPer only in Byzantine-faulty networks, since all other sharding protocols operate under such environments.

3.1 Transaction placement in UTXO-based blockchains

The hashing placement algorithm places transactions to shards based on the prefix matching of transaction IDs and shard IDs. Because transaction IDs are essentially hash values (e.g., SHA256 values in Bitcoin [24]), which are uniformly distributed over the output range of the corresponding hash function [33], hashing placement is equivalent to placing transactions randomly. In contrast, our placement algorithm RGP considers transaction dependencies.

OptChainV2[1] [26] is a client-side transaction placement algorithm for UTXO-based sharded blockchains. To reduce cross-shard transactions, OptChainV2 builds a graph with transactions as vertices and transaction dependencies as edges. OptChainV2 associates every transaction with a fitness-score array, each element of which reflects the fitness between the transaction and the corresponding shard. Based on PageRank [27], OptChainV2 computes a child transaction's fitness-score array as an element-wise weighted sum of its parents' fitness-score arrays, as shown in Figure 3. To account for load balance, OptChainV2 divides fitness scores by the corresponding transaction partition sizes and requires clients to frequently sample shards for communication latency and transaction queue length. There are two main differences between OptChainV2 and our RGP algorithm. First, OptChainV2 utilizes the information of all ancestor transactions as fitness scores are calculated in a top-down approach. By contrast, RGP only uses the recent ancestors but can achieve the same low cross-shard transaction numbers as we will see in Section 4.4. Second, RGP does not require shard sampling because peers are byzantine faulty and may not respond honestly.

[1]The difference between OptChainV2 and OptChain[25] has been given in our previous work [30].



(a) Transaction graph **(b) Weights of parents**

Figure 3: Principle of OptChainV2. f_{ij} is the fitness score between transaction i and the j-th shard. The fitness-score array of transaction x is an element-wise weighted sum of the fitness-score arrays of x's parents (i.e., transaction d and f). The weights (e.g., w_d) depend on what fraction of input UTXOs are from the parent transactions.

3.2 Account placement in account-balance blockchains

Hashing placement also applies to account partitioning and creates many cross-shard transactions [9][17]. Generally, cross-shard transactions are reduced by placing accounts that frequently transact with each other in the same shard [11][23]. To identify such accounts, Fynn et al. [11] model Ethereum transactions as a graph with accounts as vertices and transactions as edges. Using this graph, they evaluate multiple partitioning algorithms, such as METIS [16] (a well-regarded offline graph partitioning algorithm) and its variants. Since account behaviour may change over time, some of these algorithms re-partition the graph periodically. Fynn et al. concludes that METIS produces the fewest cross-shard transactions but the

worst load balance, whereas hashing placement is at the other extreme. Consequently, neither METIS nor hashing placement helps the system achieve the best performance. This conclusion agrees with the OptChainV2 paper, which employs METIS and hashing placement for comparison.

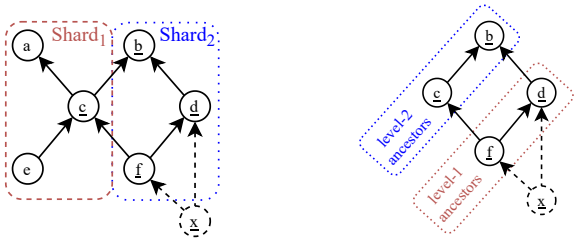
4 ROOTED GRAPH PLACEMENT

In this section, we describe our transaction placement algorithm—Rooted Graph Placement (RGP), which reduces cross-shard transactions in UTXO-based blockchains. As we observed that transactions with dependencies between each other are more likely to have their output UTXOs spent in the same future transaction, RGP attempts to reduce cross-shard transactions by placing a transaction to the shard with most of its ancestor transactions. Besides cross-shard transaction reduction, RGP also considers load balancing. Because sharding protocols usually partition peers based on unbiased random numbers and reconfigure shard membership periodically for security purposes[18][34][5], this work assumes computational resources are evenly distributed among shards. Thus, RGP attempts to assign an equal number of transactions to all shards.

4.1 Cross-shard Transaction Reduction

As RGP aims at placing a transaction to the shard with most ancestor transactions, it models these transactions and dependencies between them using a rooted directed acyclic graph (DAG). Given a new transaction, RGP builds a graph $G = (V, E)$ rooted at the new transaction. A transaction $u \in V$ only if u is an ancestor transaction of the new transaction (including the new transaction itself). A directed edge $(v, u) \in E$ if $u \in V$, $v \in V$, and v consumes the output UTXO(s) of u. The graph is built starting from the root because child transaction's input UTXO IDs carry information about parent transaction IDs. Finding all ancestor transactions is expensive, so RGP only considers ancestors within a certain distance from the root. We refer to RGP that traces back k levels of ancestor transactions as RGPk. Figure 4a illustrates an example of RGP2, where transactions that are part of the rooted graph are underlined. Transaction a is not part of the rooted graph because it is 3 hops away from the root x while RGP2 only considers ancestors within two hops. Therefore, the rooted graph is essentially a subgraph of the global transaction dependency graph. An ancestor transaction is called a *level-j ancestor* if the shortest path between the root vertex and the ancestor vertex consists of j edges, as shown in Figure 4b. With the rooted graph, RGP counts the number of ancestor transactions in each shard and calculates *cost scores* based on the counting results. A shard's cost score reflects the cost-effectiveness of placing the new transaction to the shard. Generally, RGP attempts to place a new transaction to the shard that has processed most of its ancestors, but we have two special considerations.

When calculating cost scores, RGP distinguishes between *totally spent ancestors* and *partially spent ancestors*. A partially spent ancestor will have at least one output UTXO that remains unspent after the new transaction is executed, whereas a totally spent ancestor has no output UTXOs left unspent. Figure 5 demonstrate why totally spent ancestors should be given less weight than partially spent ancestors. Suppose transaction x is a new transaction to be



(a) Ancestors in two shards (b) Ancestor levels

Figure 4: An example of RGP2. Underlined transactions are vertices of the rooted graph.

placed, and its two ancestors (i.e., transaction d and f) are in different shards. Transaction d is totally spent since $UTXO_2$ has been spent by transaction f and $UTXO_3$ will be spent by x. On the other hand, transaction f is partially spent since no transaction consumes $UTXO_5$. As a result, a future transaction y may consume the output UTXOs of both transaction f and transaction x. To prevent such future transactions from modifying two shards, transaction x should be placed to the same shard as f. Therefore, we use a coefficient $\alpha \in (0, 1)$ to give totally spent ancestors less weight than partially spent ancestors.

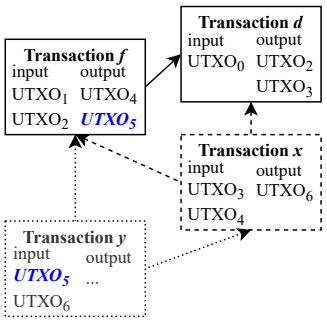


Figure 5: Transaction f is a partially spent ancestor of transaction x, and transaction d is a totally spent ancestor of x.

RGP also takes level breadths (i.e., the number of ancestors in each level) into consideration so that a counting result is not biased by the level with the most transactions. Suppose a new transaction have one level-1 ancestor (in $shard_1$) and four level-2 ancestors (one in $shard_1$ and three in $shard_2$). Without considering level breadths, RGP would tend to assign the new transaction to $shard_2$, since $shard_2$ holds one more ancestor than $shard_1$. In other words, the cost scores would be biased by level-2 ancestors because they outnumber the level-1 ancestor by 4x. However, the level-1 ancestor is important since the new transaction would be single-shard if placed to its shard. Thus, RGP divides ancestor counts by the corresponding level breadths to ensure the equal significance of each level, hence the cost score definition below:

$$S_{cost}(i) = \sum_{j=1}^{k} \frac{p_{ij} + \alpha t_{ij}}{\sum_{m=1}^{n_s}(p_{mj} + t_{mj})} \quad (1)$$

where $S_{cost}(i)$ is the cost score of shard i ($1 \le i \le n_s$); n_s is the number of shards; k is the number of ancestor levels in the graph; p_{ij} (or p_{mj}) is the number of partially spent level-j ancestors that have been placed to shard i (or shard m); t_{ij} (or t_{mj}) is the number of totally spent level-j ancestors that have been placed to shard i (or shard m); α is the totally spent ancestor weight ($0 < \alpha < 1$). The denominator $\sum_{m=1}^{n_s}(p_{mj} + t_{mj})$ is the sum of level-j ancestors across shards, which represents the breadth of the j-th level in the rooted graph. The cost score of a shard is in the range of $[0, k]$. A high cost score means placing the transaction to the shard is likely to reduce future cross-shard transactions (including the one currently being placed).

4.2 Load Balancing

When placing a transaction, RGP also calculates a *load score* for each shard to account for load balancing. A high load score means that the shard is experiencing a relatively light workload, so the transaction will experience a relatively low queuing delay if placed to the shard. To avoid ambiguity, we use partition i (denoted by P_i) to refer to the set of transactions that have been placed to shard i. Obviously, small partitions should receive high load scores. Also, we want to limit the maximum partition size difference so that load imbalance is bounded. Therefore, we model the load score of a shard as a piecewise function:

$$S_{load}(i) = \begin{cases} 0 & \text{if } |P_i| \ge |P_{min}| + \theta \\ 1 - \gamma \frac{|P_i| - |P_{min}|}{|P_{max}| - |P_{min}|} & \text{if } |P_{min}| + \theta > |P_i| > |P_{min}| \\ 1 & \text{if } |P_i| = |P_{min}| \end{cases} \quad (2)$$

where $S_{load}(i)$ is the load score of shard i ($1 \le i \le n_s$); $|P_i|$ is the size of partition i; $|P_{min}|$ and $|P_{max}|$ are sizes of the smallest partition and largest partition, respectively; $|P_{min}| + \theta$ is the boundary partition size that distinguishes *large partitions* from *medium partitions*; $\gamma \in (0, 1]$ is a coefficient that determines how heavily a medium partition is penalized. $S_{load}(i)$ gently penalizes medium partitions based on their sizes and aggressively penalizes large partitions. Note that in the second line of Equation 2, the denominator $|P_{max}| - |P_{min}|$ is implicitly guaranteed to be greater than zero because of the condition $|P_i| > |P_{min}|$. For medium partitions, $S_{load}(i)$ is in range $[1 - \gamma, 1)$.

To take into account both cross-shard transaction reduction and load balancing, the final decision should be based on both the cost score and the load score. While adding up the two scores seems to be a reasonable choice, it cannot limit load imbalance because large partitions with non-zero ancestors could have a higher score sum than small partitions without ancestors. Thus, to have bounded load imbalance, we design the overall score as the multiplication of the two scores:

$$S_i = S_{cost}(i) \cdot S_{load}(i) \quad (3)$$

where S_i is the overall score of shard i. As $S_{cost}(i)$ is in range $[0, k]$, and $S_{load}(i)$ is in range $[0, 1]$, S_i must be in range $[0, k]$. The overall scores of large partitions are always zero because of Equation 2. If $S_i = 0$ for all shards, which occurs when the new transaction only has ancestors in large partitions, RGP places the new transaction to the smallest partition. In this way, RGP never places transactions to large partitions, so the maximum partition size difference is bounded by θ. Algorithm 1 shows the complete RGP algorithm.

Algorithm 1: Rooted Graph Placement

Input: a new transaction x, the number of ancestor levels k, the number of shards n_s, transactions reachable from x within k hops, totally spent ancestor weight α, partition sizes $|P_1|, |P_2|, \ldots, |P_{n_s}|$, medium partition penalty coefficient γ, imbalance upper bound θ

Output: x's output shard ID $s_{out}(x)$

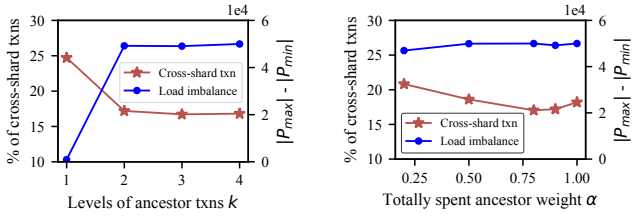
```
if x is a coinbase transaction then
    s_out(x) = hash(x) mod n_s
else
    /* Build the rooted graph */
    starting from transaction x, using BFS to build a rooted graph with k levels of ancestors.
    for i ∈ [1, n_s] do
        /* Compute the cost score of shard i */
        S_cost(i) = Σ_{j=1}^{k} (p_ij + α t_ij) / Σ_{m=1}^{n_s} (p_mj + t_mj)
        /* Compute the load score of shard i */
        if |P_i| ≥ |P_min| + θ then
            S_load(i) = 0
        else if |P_i| > |P_min| then
            S_load(i) = 1 − γ (|P_i| − |P_min|) / (|P_max| − |P_min|)
        else
            S_load(i) = 1
        /* Compute the overall score of shard i */
        S_i = S_cost(i) · S_load(i)
    /* Place x into the shard with the highest overall score */
    if max(S_i) > 0 then
        s_out(x) = argmax_i (S_i)
    else
        s_out(x) = the ID of the smallest partition
```

4.3 Impacts of Parameters

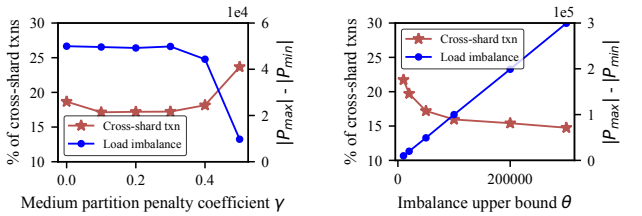
In this section, we demonstrate how the four parameters of RGP affect its transaction partitioning quality (i.e., the cross-shard transaction number and load balancing) and give the recommended parameter values. Generally, due to the intrinsic tradeoff between the two metrics, varying a parameter usually improves one metric at the cost of worsening the other. The recommended parameter values are derived using the first 200k Bitcoin blocks. However, we will see these parameters are pretty robust and transferable to different workloads in Section 4.4.

Figure 6a demonstrates that RGP1 (i.e., RGP that traces back one level of ancestor transactions) can lower the cross-shard transaction percentage to 25%, in contrast to 93% with hashing placement. RGP2 further reduces cross-shard transactions to 17%. RGP2 is able to produce fewer cross-shard transactions than RGP1 because it feeds the RGP algorithm with more information about the global dependency graph. This is consistent with what has been observed in graph partitioning [32]. Adding more levels to the rooted graph can yield even fewer cross-shard transactions, but the improvement is marginal. In Figure 6b, the lowest cross-shard transaction percentage occurs when $\alpha \in [0.8, 0.9]$. This is because when $\alpha = 1$ (i.e., totally spent ancestors are given the same weight as partially spent

ancestors), RGP ignores the fact that a partially spent ancestor and the new transaction may be referenced in the same future transaction. Nevertheless, low α values make RGP undervalue totally spent ancestors and thus increase cross-shard transactions as well.



(a) Vary k ($\alpha = 0.9, \gamma = 0.2, \theta = 50k$) **(b) Vary α ($k = 2, \gamma = 0.2, \theta = 50k$)**



(c) Vary γ ($k = 2, \alpha = 0.9, \theta = 50k$) **(d) Vary θ ($k = 2, \alpha = 0.9, \gamma = 0.2$)**

Figure 6: Influence of RGP parameters (16 shards)

Figure 6c illustrates the impact of γ. As expected, with high γ values, medium partitions are penalized heavily, hence better load balancing but more cross-shard transactions. Surprisingly, $\gamma = 0$ also results in a relatively high cross-shard transaction percentage. We believe this is because, when $\gamma = 0$, load scores equal either zero or one according to Equation 2. As a result, an overall score defined in Equation 3 equals either the cost score or zero instead of a comprehensive assessment based on both ancestor transaction distribution and shard loads. Finally, as θ controls the maximum partition size difference, it is natural that load imbalance grows linearly with θ and cross-shard transactions drop as θ grows. The elbow of the curve in Figure 6d suggests that $\theta = 50k$ is a reasonable choice since further increasing θ does not reduce cross-shard transactions much but cause high imbalance. Per the above analysis, the recommended parameter values are as follows: $k = 2, \alpha = 0.9, \gamma = 0.2, \theta = 50k$. These values are used in the rest of this paper.

4.4 Partitioning Quality Comparison

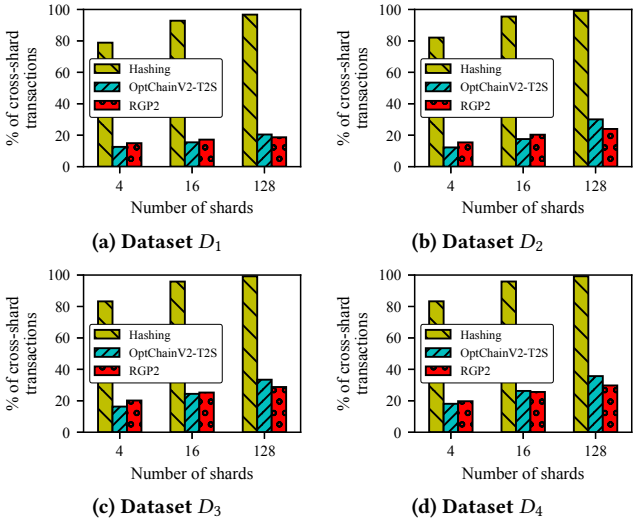
To show that the parameter value combination identified in the last section generalizes well with other workloads, we employ four transaction sets of similar sizes as detailed in Table 2. The transaction partitioning quality of RGP2 is compared with that of hashing placement and *OptChainV2-T2S*, which is OptchainV2 without shard communication latency and transaction queue length sampling[2]. OptChainV2-T2S is used instead of OptChainV2 in this section because it can be evaluated analytically without deploying shards, which is also true for hashing placement and RGP2. This is particularly useful when comparing the algorithms under a large number of shards, e.g., 128 shards. OptChainV2-T2S produces slightly fewer cross-shard transactions than OptChainV2 as it misses the sampling feature for fine-grained load balancing. If RGP2 can achieve a similar number of cross-shard transactions as OptChainV2-T2S, it will be at least as effective as OptChainV2 in terms of cross-shard transaction reduction. We will compare RGP2 with sampling-enabled OptChainV2 in Section 6.

[2]The name *OptChainV2-T2S* comes from the OptChainV2 paper, where the method is referred to as *T2S-based*.

Table 2: Four Transaction Datasets

Dataset	Bitcoin block heights	Transaction count[a]
D_1	[0, 200k)	7,316,308
D_2	[200k, 227k)	7,371,053
D_3	[227k, 252k)	7,316,337
D_4	[252k, 275k)	7,238,332

[a]Coinbase transactions are excluded.



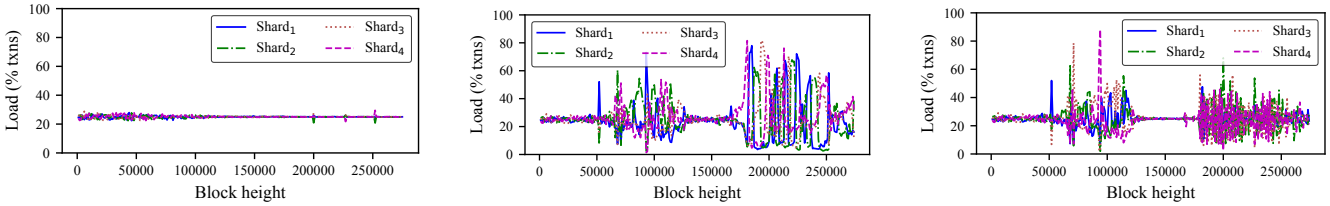
(a) Dataset D_1 **(b) Dataset D_2**

(c) Dataset D_3 **(d) Dataset D_4**

Figure 7: Cross-shard transactions

Figure 7 compares the cross-shard transaction percentage of the three placement algorithms. For all datasets, regardless of the number of shards, RGP2 and OptChainV2-T2S produce similar numbers of cross-shard transactions, and the number is significantly less than that of hashing placement. These results confirm that, despite considering only two levels of ancestors, RGP2 is able to reduce cross-shard transactions as effectively as OptChainV2-T2S. To learn how shard loads vary over time, we analyze the dynamic shard loads in a 4-shard environment, as illustrated in Figure 8. Unsurprisingly, hashing placement balances loads extremely well with every shard constantly receiving about 25% of the transactions. In Figure 8a, the three small spikes at block height 200k, 227k, and 251k correspond to the start heights of datasets $D2 \sim D4$, and the slight fluctuations at the beginning are due to small block sizes.

Another observation is that Figure 8b and Figure 8c exhibit a common pattern: all curves are quite flat in blocks [0, 50k) and [125k, 175k), but fluctuate a lot in the range of [70k, 120k) and



(a) Hashing placement (b) OptChainV2-T2S (c) RGP2

Figure 8: Dynamic shard loads (4 shards)

[180, 250k). This pattern relates to transactions consuming the output UTXOs of their immediate predecessors. For example, starting from the 326th transaction in Bitcoin block 177253, each of the 267 subsequent transactions spends the UTXOs produced by its preceding transaction[2]. When faced with such transactions, both OptChainV2-T2S and RGP2 tend to place them to the same shard as their predecessors. Consequently, a sequence of such transactions will cause a shard to temporarily receive more transactions than other shards. Table 3 shows that such transactions account for a relatively high percentage whenever the shard load curves fluctuate drastically. However, the load balancing mechanisms prevent a shard from being overloaded for a long time, so shards take turns to receive the most transactions.

Table 3: Transactions Depending on Their Predecessors

Bitcoin block height	Transactions consuming UTXOs produced by their predecessors
[0, 50k)	0.1%
[70k, 120k)	18.6%
[125k, 175k)	6.1%
[180k, 250k)	21.3%
[253k, 275k)	14.6%

5 EFFICIENT CROSS-SHARD TRANSACTION PROCESSING

Although RGP reduces the number of cross-shard transactions, it cannot eliminate them. In this section, we propose two techniques to lessen the impact of cross-shard transactions on system performance. The first technique expedites dependent transaction processing, while the second technique reduces the communication and computational overhead involved in cross-shard transaction processing. Both techniques require modifications to the atomic commit protocol. Since neither technique deals with the intra-shard consensus protocol, we abstract away the consensus process as if client requests are ordered as soon as they reach the destination shards. The techniques focus on how transactions are processed (i.e., verified and executed) once an order has been established.

5.1 Dependent Transaction Pre-verification

Cross-shard transactions usually experience long execution latency due to atomic commit protocols such as Atomix in Figure 2. This inevitably delays the processing of their dependent transactions since transactions must be executed in a dependency-respectful order. Figure 9a illustrates the timeline of processing one cross-shard transaction (i.e., tx_1) and two single-shard transactions that depend on it (i.e., tx_2 and tx_3). Suppose the client runs RGP2 and determines that tx_2 and tx_3 should be placed to the output shard of tx_1, which is $Shard_2$. Although the client sends tx_2 and tx_3 soon after sending tx_1, $Shard_2$ delays processing the two dependent transactions until tx_1 is executed to respect dependencies. In the above process, two steps are notably expensive—the locking phase and the verification of tx_2 and tx_3—because both steps involve the verification of signatures from input UTXO owners. Nonetheless, the two steps do not have to be carried out in a serialized manner.

In a UTXO-based blockchain, a peer mainly checks for three conditions when verifying a transaction: 1) the input coins exist and are unspent, 2) the total input value is not less than the total output value, and 3) the transaction includes the correct signatures of the input UTXO owners. If a transaction meets all three conditions, a peer executes it by updating its system state. Checking the first condition is stateful, whereas checking the second and third conditions is stateless. In fact, as long as the input UTXO properties (i.e., owner address, amount, etc.) are available, the second and third conditions can be checked at any time.

To reduce the execution latency of dependent transactions, we propose dependent transaction pre-verification (DPV), which performs dependent transaction signature verification in parallel with the locking phase. In order to accommodate this idea, the output shard must be informed about the cross-shard transactions' output UTXO properties early. Figure 9b illustrates how we achieve this by splitting the `COMMIT` request into two messages, namely `OUTPUT` and `LOCK-SIG`. The `OUTPUT` message carries only the cross-shard transaction and is sent to the output shard as soon as the `LOCK` request is sent to the input shard. $Shard_2$ can start verifying the signatures of tx_2 and tx_3 as soon as it receives the `OUTPUT` message. Meanwhile, the input shard is verifying tx_1's signature(s). On the other hand, the `LOCK-SIG` message carries the input shard signatures as well as a hash of the transaction, which is used to match the `LOCK-SIG` message with the corresponding `OUTPUT` message. The `LOCK-SIG` message serves as proof that the input shard has successfully locked the input UTXOs of tx_1. The `OUTPUT` and `LOCK-SIG` messages are later assembled into one `COMMIT` request so that the original `COMMIT` request processing routine can be reused. DPV also applies to cascading dependencies, e.g., $tx3$ may depend on $tx2$ instead of $tx1$, and cross-shard dependent transactions. In the

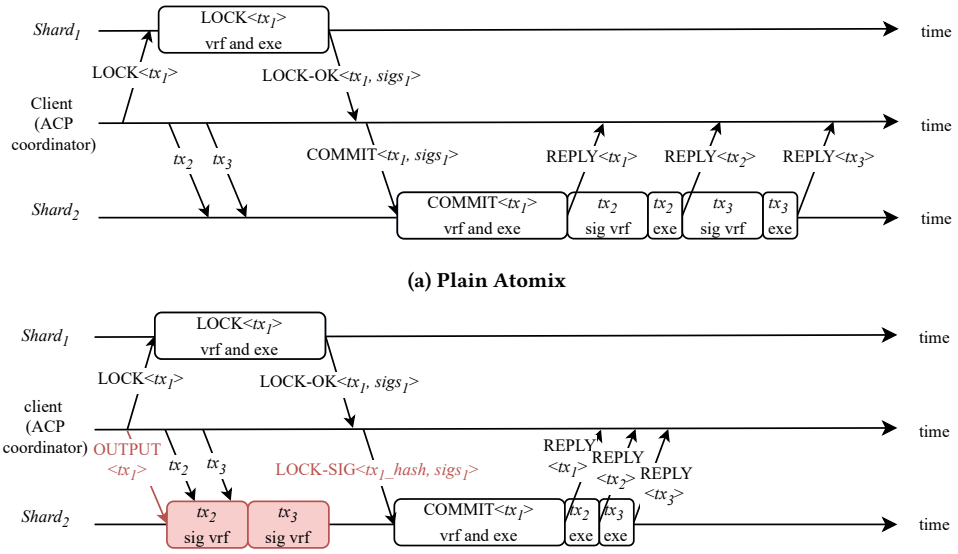


Figure 9: DPV parallelizes the Atomix lock phase with dependent transaction signature verification.

latter case, the output shard pre-verifies the LOCK requests of the cross-shard dependent transactions. DPV pre-verifies dependent transactions in their appearance order on shard ledgers.

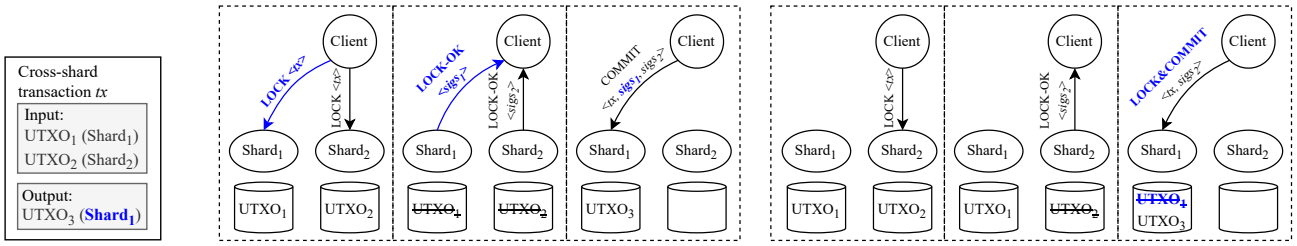
DPV is safe, i.e., invalid transactions will not be mistakenly treated as valid ones. For parent cross-shard transactions, although dishonest clients may send dummy OUTPUT messages, the transactions will not be executed without valid signatures from the input shards. In other words, LOCK-SIG messages from input shards protect output shard's system states from being tampered with. For dependent transactions, DPV may verify their signatures but does not execute them until their parent transactions are executed. As a result, a dummy OUTPUT message cannot induce the output shard to execute either the cross-shard transaction or the dependent transactions. Nevertheless, the computational work involved in pre-verifying dependent transactions is wasted, so peers should only pre-verify dependent transactions when CPUs are idle to avoid performance degradation and DoS attacks caused by dummy OUTPUT messages.

5.2 Atomic Commit Protocol Consolidation

As RGP2 takes transaction dependencies into account, the vast majority of cross-shard transactions are placed to one of their input shards. We refer to such output shards as *input-output shards*, i.e., $Shard_1$ in Figure 10a. This placement pattern is quite different than that of hashing placement, which only places a few transactions to their input shards, as illustrated in Figure 11a. This difference opens up opportunities for atomic commit protocol optimization. Specifically, the lock and commit requests can be combined into one request for the input-output shard. Figure 10 takes Atomix as an example in order to illustrate atomic commit protocol consolidation (ACPc). Instead of requesting the two input shards to lock the corresponding UTXOs as in Figure 10b, consolidated Atomix merges the LOCK request and the COMMIT request into a LOCK&COMMIT request for the input-output shard (Figure 10c). Upon receiving the LOCK&COMMIT request, the input-output shard checks for the following conditions: 1) the input UTXO(s) to lock exist, and the transaction is signed properly by the owners, 2) the signatures from other input shards are valid, and 3) total input value is not less than the total output value. If all the three conditions are met, the input-output shard deems the transaction valid and executes the transaction by removing the input UTXO(s) and adding output UTXO(s) to its UTXO database. Otherwise, the input-output shard informs the client about failed locking using a signed LOCK-NOT-OK message, which can be used as proof to restore input UTXOs in other input shards.

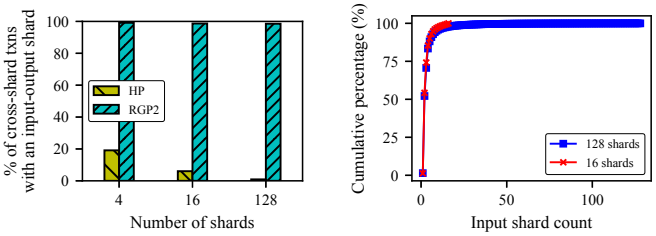
In the successful scenario, consolidated Atomix saves two messages, one shard signature, and one consensus round. For two-input-shard transactions, the saving is almost half of the processing cost, which includes five messages, two shard signatures, and three consensus rounds (two for ordering the LOCK requests and one for ordering the COMMIT request), as shown in Figure 10b. Considering that over 50% of cross-shard transactions touch only two input shards, and over 80% of cross-shard transactions span two to four input shards (Figure 11b), ACPc should produce obvious performance improvement. One may notice that there is a small portion (around 1.4%) of cross-shard transactions with one input shard. Such transactions are not placed to their input shards probably for the purpose of load balancing. They will not benefit from ACPc since they lack input-output shards. The input shard count distribution in Figure 11b results from the fact that parent transaction counts follow a power-law distribution [30]. Other transaction datasets in Table 2 show the same pattern as dataset $D1$ does in Figure 11.

ACPc is compatible with DPV. When the two techniques are deployed together, the LOCK&COMMIT message in Figure 10c splits into two parts: one message carrying the transaction, and the other



(a) Input-output shard **(b) Plain Atomix** **(c) Consolidated Atomix**

Figure 10: Consolidation of Atomix



(a) Cross-shard transactions with input-output shards **(b) CDF of input shard count**

Figure 11: The vast majority of cross-shard transactions are assigned to one of their input shards under RGP2, and over 80% of cross-shard transactions have 2~4 input shards (dataset $D1$).

carrying the signatures from other input shards as well as a hash of the transaction. The input-output shard utilizes the former to pre-verify dependent transactions, and the latter to learn the commit votes of other input shards.

6 EVALUATION

We implemented RGP2 as a client-side algorithm, which means a client runs the algorithm to compute the output shard ID before sending its transaction to blockchain peers. This architecture has also been employed by OptChainV2.

We also implemented OptChainV2 as well as an OmniLedger-like sharding protocol based on Bitcoin Core [7]. In the OptChainV2 paper, the authors modify the OmniLedger protocol to avoid excessive bandwidth usage by letting clients send requests directly to the destination shards instead of gossiping requests. We make the same modification to OmniLedger. In addition, each peer maintains a dependency graph of pending transactions to enforce dependency-respect transaction execution order. We also implemented DPV and ACPc, and measured the system performance when they are deployed together with RGP2.

6.1 Testbed

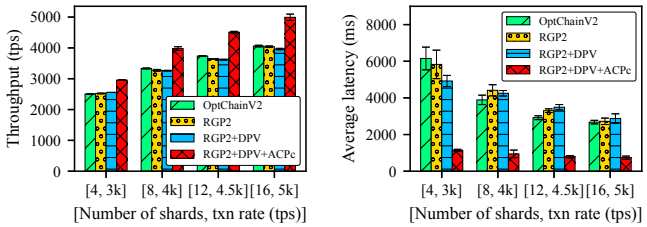
Experiments are done on a local cluster. We run up to 64 peers on 16 machines, each of which has dual Xeon E5-2620 at 2.1 GHz (12 cores) and 64GB RAM. A shard consists of four peers co-located on the same machine. Each peer is scheduled on two fixed cores for isolation purposes. To emulate a geo-distributed environment, network delay is injected between each pair of peers, and bandwidth limits are imposed on every peer using Linux `NetEm` and traffic control facilities. Properties of links between peers co-located on the same machine are also configurable through loopback interfaces. One client runs on another machine with dual Xeon E5-2630 at 2.6 GHz (12 cores, 2 hyperthreads per core) and 256GB RAM. The client reads historical transactions from the Bitcoin blockchain sequentially and sends them to the peers. To measure the maximum throughput of the system, we must saturate the peers with client requests. Considering the high processing power of the cores, we throttle each peer at 50% CPU usage.

To evaluate the performance improvement under different types of workloads, two transaction datasets are employed: transactions in Bitcoin block [0, 136k) and transactions in Bitcoin block [200k, 205k). The first dataset contains simple dependencies, whereas the second dataset is on the opposite, according to Table 3. Both datasets consist of approximately 1 million transactions. In 6.2 and 6.3, we measure the system performance under the two workloads with a 40ms round-trip delay injected between every pair of peers, and a 200-Mbps bandwidth limit imposed on every peer. This mild network configuration allows us to saturate the peers without saturating the network. The impacts of various network configurations will be evaluated in section 6.4.

6.2 Performance Under Light-Dependency Workload

We compare the performance of four systems. In the first system, the client places transactions to shards according to OptChainV2; in the second system, we replace OptChainV2 with RGP2; in the third system, we add DPV to the second system; in the fourth system, we add ACPc to the third system. In figures presenting experimental results, the four systems are denoted by OptChainV2, RGP2, RGP2+DPV, and RGP2+DPV+ACPc, respectively. Figure 12 demonstrates that all four systems scale with the number of shards. The performance of RGP2 is very close to that of OptChainV2. Adding DPV does not improve the performance due to the simple transaction dependencies, but adding ACPc improves the throughput by approximately 20% and reduces the average execution latency by 75% at the specified transaction rates.

Next, we compare the four systems from different aspects in a 16-shard environment. From Figure 13a, one can see that ACPc improves the maximum throughput by 37%. Furthermore, Figure 13b



(a) Throughput **(b) Latency**

Figure 12: Scalability

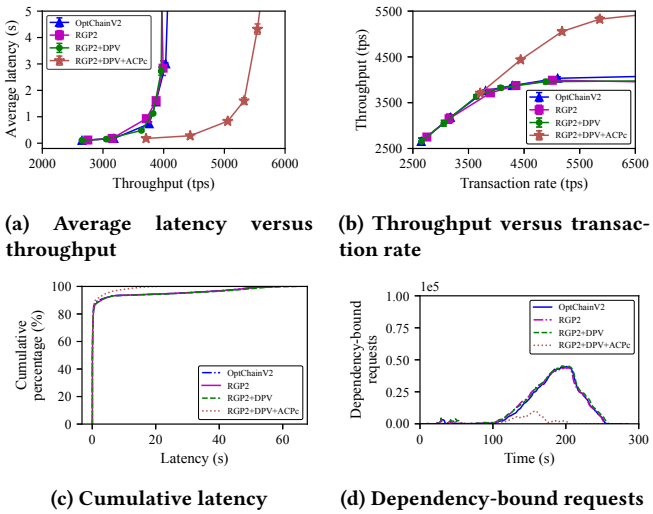
shows that, the first three systems become overloaded when transactions arrive at a 3.8k-tps rate, so the throughput can no longer grow linearly with the transaction rate. ACPc raises the turning point to 5k tps. Unsurprisingly, the latency CDF of OptChainV2, RGP2, and RGP2+DPV are fairly close as well. In contrast, ACPc significantly shortens tail latency: the 95th percentile latency is cut down by 83%, from 30s to 5s. ACPc's success in reducing tail latency implies that slow cross-shard transactions are the reason of long tail latency.

To better understand the performance of the four systems, we investigate the number of *dependency-bound requests*. In our experiments, single-shard transactions are wrapped in a `TX` requests, whereas cross-shard transactions are accomplished via `LOCK` and `COMMIT` requests as in Atomix. Dependency-bound requests refer to `TX` requests and `LOCK` requests that are waiting for their parent transactions to be executed, so that their input UTXOs become available. Figure 13d shows how the number of dependency-bound requests varies over time, where the y-axis represents the sum of dependency-bound requests across all shards. Unsurprisingly, the curve does not climb up in the first 100s because most transactions are coinbase transactions at the early stage of Bitcoin. After that, the RGP2+DPV+ACPc curve climbs much slower than the other three curves because cross-shard transactions are processed faster and thus hinder less dependent transactions. Contrary to intuition, DPV does not reduce dependency-bound requests. The reason will be given by a comparative analysis in Section 6.3.

6.3 Performance Under Heavy-Dependency Workload

Transactions in blocks [200k, 205k) contain more predecessor-dependent transactions, hence a challenging workload. Nevertheless, RGP2 can still match the performance of OptChainV2 as demonstrated in Figure 14a ~ 14c. Moreover, adding DPV to the system improves the maximum throughput from 1.9k tps to 2.7k tps (42% up), and ACPc further boosts the maximum throughput to 3.9k tps (another 44% up). In other words, DPV and ACPc together double the maximum throughput of the system. Besides, DPV notably lowers tail execution latency: the 95th percentile latency is halved. ACPc also improves execution latency due to the lower cross-shard transaction cost. As a result, DPV and ACPc collectively reduced the 50th percentile latency and 95th percentile latency by 80% and 84% respectively.

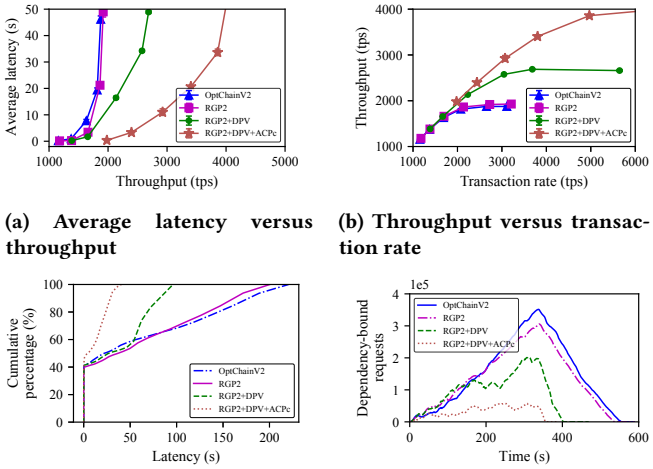
DPV improves the system performance because it reduces the number of dependency-bound requests, as shown in Figure 14d.



(a) Average latency versus throughput **(b) Throughput versus transaction rate**

(c) Cumulative latency **(d) Dependency-bound requests**

Figure 13: Performance with light-dependency transactions (16 shards). The transaction rate is 5k tps in the last two subfigures.



(a) Average latency versus throughput **(b) Throughput versus transaction rate**

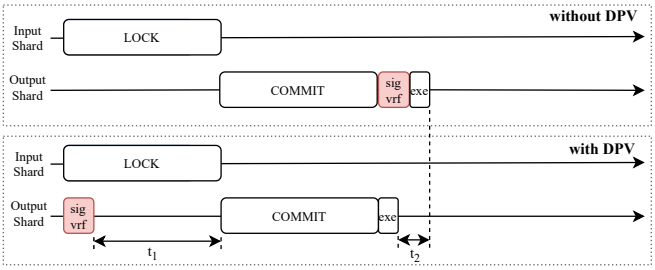
(c) Cumulative latency **(d) Dependency-bound requests**

Figure 14: Performance with heavy-dependency transactions (16 shards). The transaction rate is 3k tps in the last two subfigures.

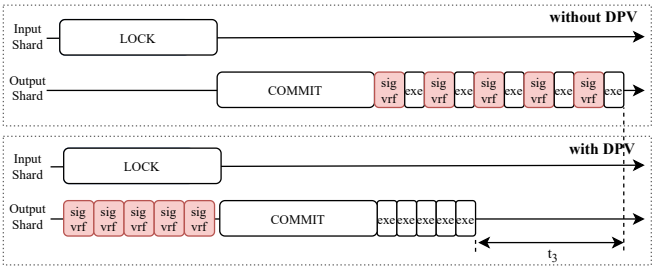
With DPV, dependency-bound requests could be pre-verified and later executed immediately after their parent transactions are executed. Therefore, such pre-verified requests have shorter execution latency and are less likely to stall other requests. ACPc also reduces the dependency-bound requests, as cross-shard transactions are processed efficiently and thus become less hindering.

The discrepancy between DPV's performance under the light-dependency workload and that under the heavy-dependency workload is due to the different ratios of dependency-bound requests to pending commit requests. For example, in Figure 15a, where the ratio is 1:1, DPV only reduces the overall processing time by

t_2, which equals the signature verification time of the dependent request. DPV cannot take advantage of the idling period t_1 since there are no more dependent requests to pre-verify. By contrast, in Figure 15b, where the ratio is 5:1, DPV significantly shortens the overall processing time.



(a) Light dependencies



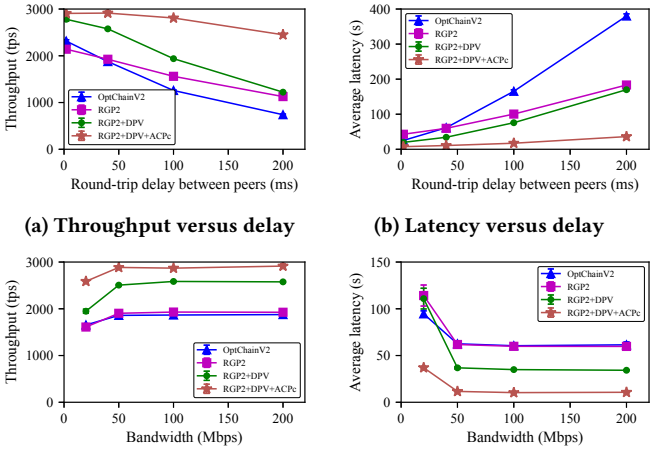
(b) Heavy dependencies

Figure 15: DPV saves more time for heavy-dependency workloads.

6.4 Performance Under Various Network Configurations

Transactions in blocks [200k, 205k) are used in this section, as dependencies in blocks [0, 136k) are too simple to show the effectiveness of DPV. Figure 16a and 16b demonstrate that all four systems suffer performance drop as network delay grows. The RGP2+DPV curve goes down faster than the RGP2 curve because long network delays "shift" the performance bottleneck from computation to network, but DPV only optimizes computation. The OptchainV2 curve also decreases more rapidly than the RGP2 curve. We suspect the reason is that OptChainV2 estimates the transaction queuing delay of a shard by sampling the recent execution rate (denoted by r_e) and the transaction queue length (denoted by n_q). Ideally, $r_e \cdot n_q$ would be the queuing delay that a new transaction would experience if placed to the shard. However, with a high network delay between peers, the high n_q value amplifies the sampling error of r_e.

In terms of bandwidth, all four systems can achieve their full potential at 50Mbps bandwidth. Extremely low bandwidth hurts the performance of all systems, with DPV suffering the most. This results from the same aforementioned reason—network becomes the bottleneck instead of computation. In general, ACPc and DPV jointly improve the system performance significantly under all network conditions, though DPV alone is not very helpful in harsh network environments.



(a) Throughput versus delay

(b) Latency versus delay

(c) Throughput versus bandwidth

(d) Latency versus bandwidth

Figure 16: Performance under various network conditions (16 shards, 3k-tps transaction rate)

7 DISCUSSION

7.1 Advantage of RGP

Although RGP does not surpass OptChainV2 in performance, it offers two big advantages in other aspects. First, RGP does not require trust in entities other than the blockchain network. Specifically, information about ancestor transactions—e.g., their respective output shards and whether they are partially spent—are all available on shard ledgers; the partition size of a shard can be estimated with the number of transactions on the shard's ledger. By contrast, to place a transaction with OptChainV2, the client must obtain the fitness score arrays of the parent transactions, which could be from other clients. Therefore, client-to-client trust is necessary to guarantee reliable fitness score array "transfer". We consulted the authors about this issue, and the two example use cases they provided are to run OptChainV2 as a public service or inside secure hardware (e.g., TEE). Second, RGP does not rely on extra information about shards, i.e., information that cannot be inferred from shard ledgers. Conversely, OptChainV2 requires clients to frequently sample the transaction queue size of every shard for transaction latency estimation. In addition to communication overhead and poor scalability with the number of clients, the sampling is also faced with a security challenge—gleaning true transaction queue sizes from Byzantine-faulty peers is not trivial.

7.2 Generalization

In this section, we discuss whether RGP, DPV, and ACPc can generalize to account-balance blockchains. Firstly, RGP is not applicable to account-balance blockchains. As mentioned in Section 3, cross-shard transaction reduction in account-balance blockchains is achieved through account placement [11] [23]. Although a graph of accounts could be employed to partition accounts, RGP does not apply to such graphs. This is because RGP is essentially a streaming graph partitioning algorithm, but accounts are not created in a streaming manner.

DPV can be adapted to account-balance blockchains by modifying the definition of dependent transactions accordingly. In an account-balance blockchain, dependencies should be established based on read-write conflicts or write-write conflicts. For example, if two transactions $tx1$ and $tx2$ both update the same account, and $tx1$ is ordered before $tx2$ on the shard ledger, then $tx2$ is a dependent transaction of $tx1$ and must be executed after $tx1$. Once dependent transactions are identified, their signatures can be verified before their parent transactions are executed, because signature verification comprises only stateless computing.

Lastly, ACPc does not generalize to account-balance blockchains due to the fundamental difference between the UTXO model and the account-balance model. In the UTXO model, a UTXO is not supposed to be spent by two or more transactions, so input shards do not have to know the commit decision if the cross-shard transaction succeeds. In other words, the two following scenarios are equivalent in terms of preventing future transactions from claiming input UTXO(s) of a cross-shard transaction: 1) the input shard receives a message confirming the transaction's successful execution, and 2) the input shard does not receive any message indicating whether the transaction succeeds or not. However, in account-balance blockchains, an account could be updated by multiple transactions. Thus, transaction isolation should be enforced to avoid concurrency issues [5][15]. Therefore, all shards involved in a transaction must be aware of the commit decision, so that they can release the involved accounts for other transactions to access. As a result, no shard can skip sending its `LOCK` response to the coordinator, which renders ACPc inapplicable.

8 CONCLUSION

Hashing placement is a common transaction placement algorithm used in blockchain sharding protocols, but produces a large number of cross-shard transactions due to dependency ignorance. We have developed Rooted Graph Placement and demonstrated that it significantly reduces cross-shard transactions. By considering two levels of ancestor transactions, RGP can match the performance of OptChainV2, a state-of-the-art transaction placement algorithm with special trust requirements. For the remaining cross-shard transactions, we have devised Dependent Transaction Pre-verification and Atomic Commit Protocol Consolidation to lower their impact on the system performance. DPV takes advantage of idling computational resources, and ACPc reduces both computational work and network usage. Our experiments have demonstrated that DPV and ACPc jointly can double the maximum throughput under heavy-dependency workloads.

REFERENCES

[1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Xi'an, China, 76–88.

[2] Blockchain.com. 2021. Bitcoin Explorer. https://www.blockchain.com/btc/tx/91c40e195524962aa3e6cd588e2038b392368382d0815aba7034f51c3ce2579b. Accessed: 2021-09-08.

[3] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. USENIX, New Orleans, 173–186.

[4] Kyle Croman, Christian Decker, Ittay Eyal, et al. 2016. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*. Springer, Barbados, 106–125.

[5] Hung Dang, Tien Tuan Anh Dinh, et al. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam, 123–140.

[6] Christian Decker and Roger Wattenhofer. 2015. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*. Springer, Edmonton, Canada, 3–18.

[7] Bitcoin Core Developers. 2020. *Bitcoin Core integration/staging tree*. GitHub. https://github.com/bitcoin/bitcoin Last accessed 15 Jun 2020.

[8] ethereum.org. 2022. Shard chains. https://ethereum.org/en/upgrades/shard-chains/. Accessed: 2022-02-27.

[9] eth.wiki. 2022. On sharding blockchains FAQs. https://eth.wiki/sharding/Sharding-FAQs. Accessed: 2022-02-28.

[10] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. 2016. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX symposium on networked systems design and implementation (NSDI 16)*. USENIX, Santa Clara, 45–59.

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

[12] James N Gray. 1978. Notes on data base operating systems. In *Operating Systems*. Springer, Berlin, 393–481.

[13] Rachid Guerraoui and Jingjing Wang. 2017. How fast can a distributed transaction commit?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, Chicago, 107–122.

[14] Jelle Hellings, Daniel P. Hughes, et al. 2020. Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing. arXiv:2008.04450 [cs.DC]

[15] Jelle Hellings and Mohammad Sadoghi. 2021. Byshard: Sharding in a byzantine environment. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2230–2243.

[16] George Karypis and Vipin Kumar. 1995. Multilevel graph partitioning schemes. In *Proceedings of The International Conference on Parallel Processing*. ACM, Urbana-Champain, 113–122.

[17] Sanghyeok Kim, Jeho Song, Sangyeon Woo, Youngjae Kim, and Sungyong Park. 2019. 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)*. IEEE, Umea, Sweden, 188–193.

[18] Eleftherios Kokoris-Kogias, Philipp Jovanovic, et al. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, 583–598.

[19] Butler Lampson and Howard E Sturgis. 1979. Crash recovery in a distributed data storage system.

[20] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. 2015. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*. Springer, Puerto Rico, 528–547.

[21] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling Nakamoto Consensus to Thousands of Transactions per Second. arXiv:1805.03870 [cs.DC]

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

[23] Avi Mizrahi and Ori Rottenstreich. 2020. Blockchain State Sharding with Space-Aware Representations. *IEEE Transactions on Network and Service Management* 18, 2 (2020), 1571–1583.

[24] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

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

[26] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. 2021. OptChain: optimal transactions placement for scalable blockchain sharding. arXiv:2007.08596v2 [cs.DC]

[27] Lawrence Page, Sergey Brin, et al. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[28] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[29] Ling Ren, Kartik Nayak, Ittai Abraham, and Srinivas Devadas. 2017. Practical synchronous byzantine consensus. arXiv:1704.02397 [cs.DC]

[30] Liuyang Ren and Paul A. S. Ward. 2021. Transaction Placement in Sharded Blockchains. arXiv:2109.07670 [cs.DC]

[31] Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. ACM, Ann Arbor Michigan, 133–142.

[32] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, Beijing, 1222–1230.

[33] Douglas R Stinson. 2005. *Cryptography: theory and practice*. Chapman and Hall/CRC, London.

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