Latency-First Smart Contract: Overclock the Blockchain for a while

Huayi Qi*, Minghui Xu*, Xiuzhen Cheng*, Weifeng Lyu†
* School of Computer Science and Technology, Shandong University
† State Key Laboratory of Software Development Environment, Beihang University
Email: qi@huayi.email, {mhxu, xzcheng}@sdu.edu.cn, lwf@buaa.edu.cn

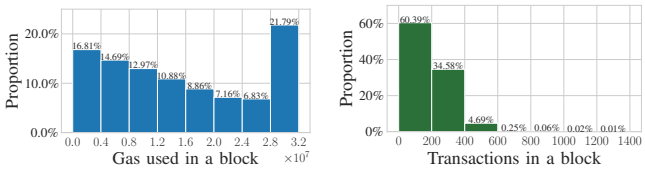
***Abstract*—Blockchain systems can become overwhelmed by a large number of transactions, leading to increased latency. As a consequence, latency-sensitive users must bid against each other and pay higher fees to ensure that their transactions are processed in priority. However, most of the time of a blockchain system (78% in Ethereum), there is still a lot of unused computational power, with few users sending transactions. To address this issue and reduce latency for users, we propose the latency-first smart contract model in this paper, which optimistically accepts committed transactions. This allows users to submit a commitment during times of high demand, and then complete the rest of the work at a lower priority. From the perspective of the blockchain, this temporarily "overclocks" the system. We have developed a programming tool for our model, and our experiments show that the proposed latency-first smart contract model can greatly reduce latency during the periods of high demand.**

***Index Terms*—latency-first smart contract, Ethereum, blockchain**

I. Introduction

With the growing popularity of blockchain platforms, transaction latency becomes a bottleneck leading to a poor user experience. This latency issue matters in both permissionless and permissioned chains. Ethereum developers found that during a busy time, only a small proportion of transactions get confirmed into the incoming block, causing others to wait for the demand to decrease before being included and thus resulting in a frustrating user experience [1]; as a consequence, latency-sensitive users have to bid against each other and pay higher fees to miners in order for their transactions to be processed in priority. Performance test results of Hyperledger Fabric [2], [3] also demonstrate that the workload of a blockchain system strongly affects latency, and latency can increase significantly when the number of transactions exceeds its capacity. Many efforts have been made to address this latency issue and improve user experience. On-chain scaling techniques such as sharding [4], [5] and paralleling [6], [7], [8] reduce latency by providing additional computing resources and increasing throughput of the chain. Off-chain scaling techniques such as state channels [9], [10], [11], Plasma [12], and optimistic rollups [13] allow users to send their transactions to state channels or child chains for preventing the main chain from being overwhelmed.

Corresponding author: Minghui Xu (mhxu@sdu.edu.cn).



(a) (b)

Fig. 1. The distribution of transactions and gas in 100,000 blocks.

In this paper, we present a latency-first smart contract scheme that intends to solve the latency issue by an approach without introducing additional machines and computing resources – our idea is to rebalance the computing tasks with time to increase blockchain utilization. Our statistical analysis of the load distribution based on the recent 100,000 blocks in Ethereum[1], as shown in Fig. 1, indicates that the blockchain does not consistently operate at maximum capacity. After London Upgrade, the maximum block gas limit (the maximum computational capability of the Ethereum network) is about 30 M. Fig. 1(a) indicates that only 21.79% of the blocks use more than 28 M gas, which is close to the maximum block gas limit and therefore can not process more transactions, and the remaining 78.21% of the blocks have a significant amount of available computational capacity. Fig. 1(b) illustrates that 94.97% of the blocks contain fewer than 400 transactions, while 0.03% contain more than 1,000 transactions. This result indicates that a block can contain more transactions if it requires a significantly little gas to process. In conclusion, the transactions are not balanced with time, leading to periods of low utilization and heavy load on the blockchain. During times of high demands, more delays are expected for users because of the increased gas consumption of the transactions. Of course, a blockchain system is unable to control the time when a user prefers to send a transaction. It would be helpful if there exists a way to temporarily increase the performance, similar to the concept of overclocking in traditional computing. This is what our proposed latency-first smart contract model does. It optimistically accepts committed transactions, enabling users to "promise" to provide the proof of a transaction at a later

[1]We collected the data on May 06 2022. The heights of the recent 100,000 blocks are 14622039 – 14722038 (Apr. 20 2022 – May 06 2022). https://github.com/SDU-IIC-Blockchain/ethereum-tx-distribution-analyze

time. By doing so, the load on the network can be balanced over time and the latency can be reduced.

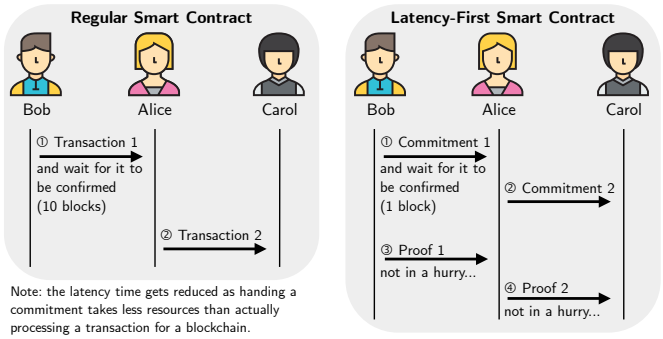


Fig. 2. The workflow of a smart contract on a heavy loaded blockchain, with and without our proposed latency-first model.

Fig. 2 illustrates the workflow of a smart contract on a heavily loaded blockchain. In this example, Alice sells an item to Bob in exchange for money (Transaction 1) and then uses that money to purchase another item from Carol (Transaction 2). However, due to a heavy load on the blockchain, it takes 10 blocks for Transaction 1 to be confirmed, which is much longer than the typical 1 block confirmation time. In our proposed model, Alice and Bob can submit a commitment (Commitment 1) instead of a regular transaction during the heavy load time to reduce latency. Commitments can often be confirmed within 1 or 2 blocks because they require fewer resources to process. Alice can then make Commitment 2 with Carol. To prove Commitment 1, Alice sends a transaction in low priority. The major challenge in designing such a latency-first smart contract model is to allow a large number of users to submit commitments simultaneously without creating unnecessary dependencies, as well as to address the issues that may happen when a party refuses to prove a commitment.

Challenges and contributions are summarized as follows:

- **Reduce latency and balance loads.** A blockchain system usually has spare capabilities to handle more transactions in a majority of the time, but sometimes it is overwhelmed with a large number of transactions. The loads are not balanced with time, which can lead to the increased latency during periods of high demands. Our model aims to balance the blockchain computational resources, making it resilient to high-frequency requests in short periods of time. This improves utilization during periods of low demands and reduces latency during periods of high demands.
- **Avoid unnecessary data conflicts and reduce the impact of revocations.** A smart contract model that optimistically accepts committed transactions allows users to change the contract state in advance and "promise" to prove a transaction at a later time. An unproved transaction causes a revocation that rolls back the contract state. However, during the time it takes for a latency-first transaction to be proved or revoked (which could take several hours or even days), the contract state may have changed due to other transactions, leading to data conflicts. In our proposed latency-first smart contract model, a contract has independent states managed by users. This allows future transactions not to rely on unrelated transaction commitments, thereby reducing the influence of revocations.
- **Provide a friendly programming tool.** In order to effectively develop smart contracts that follow the latency-first model, a developer-friendly compiler is necessary. By providing a friendly programming tool, we aim to make it easier for developers to create and deploy smart contracts that follow our proposed latency-first model.

The paper is organized as follows: In Sec. II, we introduce the most related work and provide the necessary background information. In Sec. III, we propose our latency-first smart contract model and perform the security analysis. In Sec. IV, we introduce the programming tool and demonstrate the performance evaluation. Finally, we provide a conclusion in Sec. V.

II. Related Work and Preliminaries

A. Related Work

We overview the related work focusing on scalability, optimistic techniques, and modifiable blockchain.

Scalability Techniques. More computational resources can be added to a blockchain system, thereby increasing its throughput to reduce latency. Such techniques include sharding, concurrent execution, and state channels. Huang et al. [5] addressed the hot-shard issue to reduce cross-shard transactions, resulting in an improved system throughput and decreased transaction confirmation latency. Qi et al. created a shepherded parallel workflow [7] for permissioned blockchain networks to replace the sequential workflow and increase throughput. Chen et al. proposed SChain [8], which provides intra-block concurrency and inter-block concurrency, leveraging the capacity of multi-core processors and multiple peers to enable simultaneous processing. Dziembowski et al. presented a general state channel network [9] to allow the execution of arbitrary complex smart contracts as off-chain protocols, in which a massive amount of transactions are executed without requiring the costly interactions with the blockchain, resulting in latency reduction.

Optimistic Techniques. Our latency-first model optimistically accepts pending transactions. This concept has also been applied in Layer-2 network and BFT consensus. Optimistic rollup [13] improves blockchain scalability by not requiring any computation by default but relying on fraud proofs to prevent invalid state transitions from happening. Kotla et al. presented Zyzzyva [14], using speculation to reduce the cost and simplify the design of BFT state machine replication.

Modifiable Blockchain. Modifiable blockchain shares the same revocation challenges as our latency-first model, as revoking previous transactions may have a significant impact on successors. Most works focus on replacing a block or transaction by only editing the text data that does not affect

other transactions, thereby bypassing the revocation issue. Ateniese et al. proposed a framework [15] to re-write blocks and remove inappropriate contents. Deuber et al. developed an efficient redactable blockchain under the permissionless setting [16] by consensus-based voting. They noted that removing a transaction entirely may result in serious inconsistencies in the chain; thus only redactions that do not affect a transaction's consistency are allowed to be modified.

B. Preliminaries

1) Blockchain: A blockchain is a decentralized and append-only database, consisting of blocks, which are appended periodically after consensus. Each block contains a number of transactions, which are validated by miners and stored by full nodes. Denote itv as the block interval, i.e the time interval between two contiguous blocks. Note that itv varies in a small proportion, the average of which is close to a preset value defined in the blockchain system, which is ensured by a consensus algorithm. For example, the preset block interval of Bitcoin [17] is 10 min, while Ethereum [18] has a 15 sec interval. This brings the definition of latency. Denote T as the time when a user submits a transaction to a blockchain, and C as the time when the transaction is included in a block, then $C - T$ is the confirmation delay for this transaction, or the latency. One can see that by definition, $C - T$ can also be roughly expressed as $n \cdot \mathsf{itv}$, where n stands for the number of blocks passed by. Each block has a capacity limit because a larger block usually brings additional network latency during consensus, which increases the chance of forks and causes less performant full nodes to gradually loose its ability to keep up with the network due to space and speed requirements. Since block capacity is limited, the latency issue occurs when the blockchain system is overwhelmed by transactions. Only a limited number of transactions can be confirmed into a block. Other transactions may be either queued or deleted from miners' transaction pool. A transaction contains a fee for miners, so miners are preferring transactions with higher fees. As a consequence, latency-sensitive users have to bid against each other and pay more fees to the miners to make sure that their transactions are processed in priority.

2) Smart Contract: Smart contracts, which were first introduced by Ethereum, are programs that run on the Ethereum Virtual Machine (EVM). They consist of program codes and variables, and are triggered by transactions that contain call data. When a transaction is submitted, every miner on the network executes the code and modifies the variables of the smart contract. The block capacity of the Ethereum network is defined by the block gas limit, which is the maximum amount of resources that can be used by all transactions in a block. The gas expended by a transaction is the sum of the cost of each EVM instruction used in the transaction. Transactions that require more gas to execute consume more CPU and IO resources, and take longer to validate. Since the time consumed by a smart contract transaction varies significantly, the latency issue can also occur when a few transactions in a block require a tremendous amount of time to validate, making the users suffer from the latency issues.

TABLE I
NOTATIONS

Symbol	Description
TX	Transaction
U	User
S	State
d	Deposit
f	State transition function
x	Input of f
$\mathbf{S}$	Vector of S, transited by f
$\mathbf{S}'$	Vector of S, the transition result of f
H	Hash function
$c = (h_{\mathsf{state}}, h_{\mathsf{input}})$	Commitment
$\mathbf{U}, \mathbf{c}, \mathbf{d}$	Vectors of corresponding items involved in a transaction

III. Latency-First Smart Contract

In this section, we introduce our latency-first smart contract model. First, we brief the core concepts. Then, we detail the algorithms regarding how to commit, prove, and revoke a commitment considering both simple and nested cases. The major notations and their semantic meanings are listed in Table I

A. Core Concepts

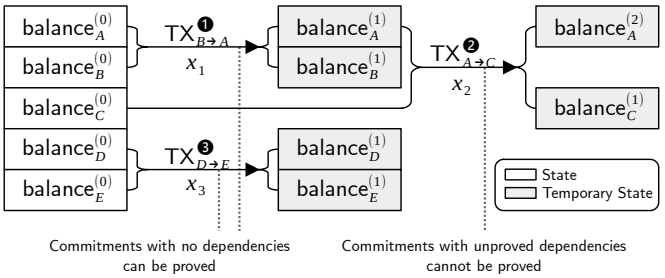


Fig. 3. A wallet example in the latency-first smart contract model. Numbers inside black circles identify the order of occurrence.

Our proposed latency-first smart contract model involves two types of entities, namely users and miners. Each user U has a "state", i.e., a set of variables, stored on chain. A state can only be updated after the corresponding commitment is proved; but a temporary state can be computed and stored locally (off-chain) by a user after receiving the state transition input x. Accordingly, a user may maintain a sequence of temporary states following the sequence of unproven commitments. One can see that our latency-first smart contract model differs from the traditional one in two aspects: variables in the latter do not have a designated owner and a user state in the former does not have to be updated immediately after a commitment is received. A commitment is the pair containing the hash of the input x and that of the temporary state resulted from the transition function on input x; therefore a transaction has multiple commitments, with one for each user. Each user involved in a transaction also needs to determine its deposit

value d_U, which is used to compensate for any potential losses and ensure fairness in trading. The commitments and deposits from all users need to be signed by each user in the transaction, then sent to the smart contract for future approval. Note that each unproved transaction in the smart contract is associated with a timer, ensuring that if the users do not provide the correct input x within a predetermined time frame, the commitments are revoked and the deposits are applied. By such a design one can see that when the blockchain is in a high demand, our latency-first scheme allows a group of users to submit commitments as "promises" that the corresponding proofs will be uploaded later, which can help to rebalance the workload and reduce average latency. Also note that we allow the commitments involved in a transaction to be approved at any time but revoking them before the associated timer times-out is not permitted, to enhance the effectiveness of our latency-first scheme and protect serious and benign users.

Fig. 3 demonstrates an example of a latency-first smart contract, in which the state of a user is denoted by $\mathsf{balance}_U$. In this example, the initial states of all users, denoted by superscript $^{(0)}$, are on-chain ones. Temporary states are labeled by superscripts $^{(1)}, ^{(2)}, \cdots$. A state transition involves transferring certain amount of one user to another, which modifies the balances (states) of both the sender and the receiver. The transaction $\mathsf{TX}_{B\to A}$ updates Bob's and Alice's states to temporary states $\mathsf{balance}_B^{(1)}$ and $\mathsf{balance}_A^{(1)}$, respectively, and the transaction $\mathsf{TX}_{A\to C}$ updates Alice's temporary state $\mathsf{balance}_A^{(1)}$ and Carol's on-chain state $\mathsf{balance}_C^{(0)}$ to temporary states $\mathsf{balance}_A^{(2)}$ and $\mathsf{balance}_C^{(1)}$, respectively, as the commitments for the transaction $\mathsf{TX}_{B\to A}$ have not been proved before the ones for $\mathsf{TX}_{A\to C}$ are received. Our example also shows a transaction between Dave and Eve $\mathsf{TX}_{D\to E}$, which is independent of the states of Alice, Bob and Carol.

For each transaction, the two parties independently compute their commitments based on the input and their temporary states. Specifically, Alice and Bob compute their commitments for the transaction $\mathsf{TX}_{B\to A}$ based on x_1 as well as $\mathsf{balance}_A^{(1)}$ and $\mathsf{balance}_B^{(1)}$. These commitments and the deposit values agreed by Alice and Bob (d_A and d_B) are signed and then uploaded to the miners. After receiving the signed commitments/deposits, miners lock the users' on-chain states, if unlocked, append the commitment and deposit value on chain for each user to confirm that the commitments are related to $\mathsf{TX}_{B\to A}$. Alice and Bob also store their temporary states locally before the commitments get proved.

When the blockchain has spare capacity, either Alice or Bob can submit x_1 to miners to prove their commitments for the transaction $\mathsf{TX}_{B\to A}$. If the state transition and hashes are successfully validated, the miners update Alice and Bob's on-chain states; otherwise, the transaction $\mathsf{TX}_{B\to A}$ gets revoked. In such a case, $\mathsf{TX}_{A\to C}$ can still be successful if $\mathsf{balance}_A^{(0)}$ is sufficient to pay Carol.

B. Latency-First Operations: Commit, Prove, and Revoke

In this subsection, we first leverage a simple case of applying the basic operations of latency-first to help understand how our smart contract concretely works. A simple case means that a group of users submit commitments that are resulted from a single input, whose on-chain states are not locked. Then we illustrate how to handle a more sophisticated nested case where one or more new commitments are dependent of previously created temporary ones.

1) Simple Case: Simple cases are quite popular in practice. For example, when a customer pays an amount x for a video to an online store who keeps amount y to itself and transfers amount z to the video producer as requested by copyright agreement, where $x = y + z$. In such a case, the three parties form a group, and the money transfer from the customer to the store as well as the one from the store to the video producer must be considered together to guarantee the atomicity of the transaction. Formally speaking, a simple case involves a transaction TX, a group of users $\mathbf{U}$ affected by TX, and input x, where the on-chain states of all users are not locked. Each user U_i computes a temporary state S_i with input x, then creates a commitment c_i. Let $\mathbf{d} = (d_1, d_2, \ldots, d_{|\mathbf{U}|})$ be the deposits of the users in $\mathbf{U}$. Note that d_i can be either positive or negative, which indicates earning or spending money. Besides, $\sum_{i=1}^{|\mathbf{U}|} d_i = 0$ since money cannot come out of thin air throughout the life of a transaction. Deposits are applied to "un-do" the transaction in case a revocation occurs. Suppose a commitment c_i gets revoked, the owner of it, i.e., U_i, pays d_i if $d_i \geq 0$ or gets $-d_i$ as compensation if $d_i < 0$. In the following we employ a two-user example to illustrate our latency-first smart contract model for the sample case.

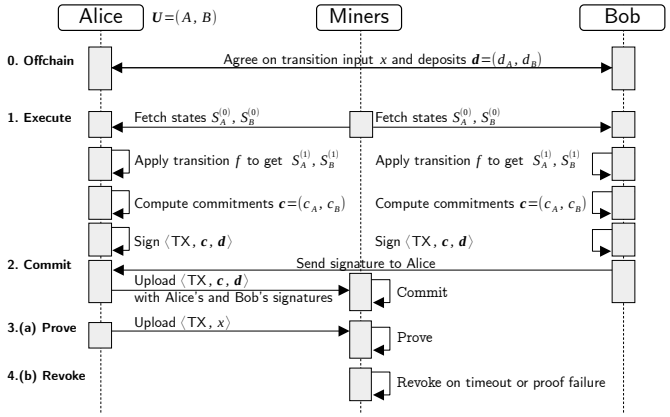


Fig. 4. Simple case example.

The complete workflow of the simple example is shown in Fig. 4, in which Alice trades digital money defined in the smart contract for Bob's real money. Before the transaction TX starts, Alice and Bob agree on the transfer amount x and deposits $\mathbf{d} = (d_A, d_B)$. We have $d_A > 0$ and $d_B = -d_A < 0$, so that when the transaction is revoked, Bob gets d_A as compensation. Next, both Alice and Bob locally and independently `Execute`: fetch the on-chain states $S_A^{(0)}, S_B^{(0)}$, obtain the temporary states $S_A^{(1)}, S_B^{(1)}$ by computing the state transition function f with input x, calculate commitments $\mathbf{c} = (c_a, c_b)$, and finally sign TX, $\mathbf{c}$, and $\mathbf{d}$. After that, either

Alice or Bob sends her/his signature to the other, who then transfers TX, $\mathbf{c}$, $\mathbf{d}$ and both users' signatures to a miner. The miners execute `Commit` to save the two users' commitments on chain, setup a timer, and lock their deposits and states after signature verification. Later, either Alice or Bob sends a miner TX and the transfer amount x to prove their commitments when the blockchain is not busy. The miners execute `Prove` to compute the new states for Alice and Bob by applying the state transition function with input x, verify correctness by checking against the previously submitted commitments, and finally replace their on-chain states with the newly computed ones. If their commitments are not proved after a designated time interval (the timer times out), miners call `Revoke` to revoke the transaction.

2) Nested Case: Nested cases emerge when proofs are not validated in time as they are less prioritized than commitments. For example, a customer pays x_1 for an online video, in which y_1 is for the video store and the rest $z_1 = y_1 - x_1$ needs to be transferred to the producer. Before the commitments of this transaction get proved, the customer pays x_2 for a meal, in which y_2 is transferred to the restaurant and $z_2 = y_2 - x_2$ is for tip. The second group of commitments depend on the first one as the second state transition depends on the customer's temporary state resulted from the first transaction. Formally speaking, a nested case involves a sequence of transactions $\mathsf{TX}_1, \mathsf{TX}_2, \ldots$, multiple user groups $\mathbf{U}_1, \mathbf{U}_2, \cdots$, with one for each transaction, and multiple inputs $x_1, x_2, \cdots$, with one for each transaction, where the number of transactions keeps growing before all committed ones get proved. A transaction triggered by x_i can rely on the temporary states whose corresponding commitments are submitted but not proved. In the following we illustrate our latency-first smart contract model for the nested case by an example where each transaction involves two users.

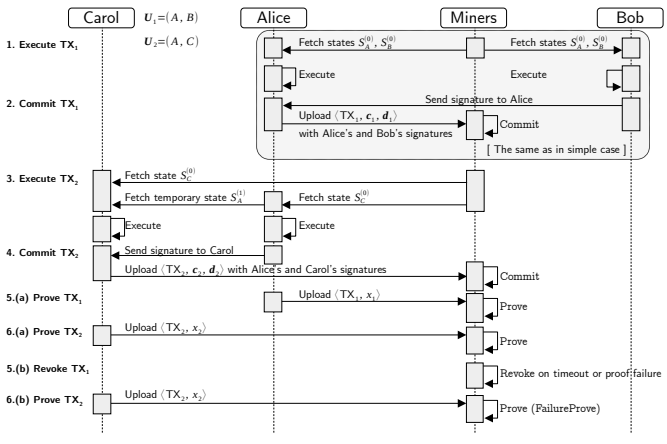


Fig. 5. A nested case example.

The workflow of the nested case example is demonstrated in Fig. 5, where Alice trades with Carol TX_2 after her previous transaction with Bob TX_1. First, Alice submits TX_1, $\mathbf{c}_1$, and $\mathbf{d}_1$ with Alice's and Bob's signatures to a miner as in the simple case. Miners execute `Commit` to save commitments about TX_1 on-chain. Then, Alice trades with Carol. They also locally `Execute`, but the procedure is slightly different from the simple case: Carol fetches Alice's temporary state $S_A^{(1)}$ from Alice. Each of them then independently performs the rest: fetch Carol's state $S_C^{(0)}$ from the chain, obtain the temporary states $S_A^{(2)}$, $S_C^{(1)}$ by computing the state transition function f with input x_2, and sign TX_2, $\mathbf{c}_2$, and $\mathbf{d}_2$. Either Alice or Carol sends a miner TX_2, $\mathbf{c}_2$, and $\mathbf{d}_2$ with both users' signatures. The miners execute `Commit` to save the commitments about TX_2 on chain. Later, if Alice and Bob cannot prove TX_1 on time, TX_1 is revoked. Carol sends TX_2 and x_2 to prove TX_2. Miners carry out `FailureProve` after performing commitment verification to handle the situation where a previous related transaction is revoked. Depending on whether or not Alice has sufficient balance to pay Carol, after revoking TX_1, TX_2 might either be proved or revoked.

C. The Full-fledged Protocol

Four procedures are involved in our latency-first smart contract: `Execute` as a user protocol shown in Algorithm 1; `Commit`, `Prove`, `Revoke` as miner protocols, with two subroutines `FailureProve` and `VerifyCommit`, shown in Algorithm 2. A transaction TX has one of the following five statuses: Processing, Committed, Proved, Revoked, and FailureProved. Initially, a transaction is under Processing.

Algorithm 1: User Protocol

upon receiving $\langle \texttt{Execute}, \mathsf{TX}, \mathbf{U}, x, \mathbf{d} \rangle$ **do**:
  foreach $U_i \in \mathbf{U}$ **do**
    Fetch U_i's latest state as S_i
    if S_i *is a temporary state* **and** $H(S_i)$ *is invalid* **then**
      return
  $\mathbf{S} = \{S_1, \cdots, S_{|\mathbf{U}|}\}$
  Compute state transition $\mathbf{S}' = f(\mathbf{S}, x)$
  Compute $\mathbf{c} = \{(H(S_i'), H(x)) | S_i' \in \mathbf{S}'\}$ as commitments for TX; Broadcast $\langle \mathsf{TX}, \mathbf{c}, \mathbf{d} \rangle_\sigma$ to $\mathbf{U}$
  Aggregate received messages and send them to miners

Algorithm 1 is executed off-chain by each user in $\mathbf{U}$, where $\mathbf{U}$ is a group whose states are affected by TX. To execute a transaction TX with input x, a user first fetches the latest states of all users, where the state of each user is either an on-chain state or the latest temporary one, depending on whether or not the corresponding user has committed not-yet-proved transactions. In the latter case, the temporary state S_i is fetched from the corresponding user U_i instead of chain, and thereby a verification is processed to check whether $H(S_i)$ matches the value in the commitment of U_i's latest committed transaction, which can be obtained from the blockchain. After fetching all states, the user then computes the temporary state for each user, calculates each user's commitment, and finally signs TX, commitments $\mathbf{c}$, and deposits $\mathbf{d}$. A commit message

Algorithm 2: Miner Protocol

upon receiving $\langle$`Commit`, TX, $\mathbf{c}, \mathbf{d}, \mathbf{U}\rangle$ **do**:
  Lock deposits $\mathbf{d}$
  Save $(\mathbf{U}, \mathbf{c}, \mathbf{d})$ on chain with TX as the key
  Mark TX as Committed

upon receiving $\langle$`Prove`, TX, $x\rangle$ **do**:
  Retrieve $(\mathbf{U}, \mathbf{c}, \mathbf{d})$ with TX
  Fetch $\mathbf{U}$'s on-chain states as $\mathbf{S}$
  if `VerifyCommit`(TX, $\mathbf{S}, x, \mathbf{c}$) = TRUE **then**
    Unlock deposits $\mathbf{d}$
    Compute state transition $\mathbf{S}' = f(\mathbf{S}, x)$
    Update users' on-chain states to $\mathbf{S}'$
    Mark TX as Proved

upon receiving $\langle$`Revoke`, TX$\rangle$ from a timer or `VerifyCommit`() **do**:
  Retrieve $(\mathbf{U}, \mathbf{c}, \mathbf{d})$ with TX
  Apply deposits $\mathbf{d}$
  Apply penalty and ban $\mathbf{U}$ for a time period
  Mark TX as Revoked

Function `FailureProve`(TX, TX, $x, \mathbf{S}$)**:**
  Retrieve $(\mathbf{U}, \mathbf{c}, \mathbf{d})$ with TX
  Fetch $\mathbf{U}$'s on-chain states as $\mathbf{S}$
  Compute state transition $\mathbf{S}' = f(\mathbf{S}, x)$
  if *transition failed* **then**
    Apply deposits $\mathbf{d}$
    Mark TX as Revoked
  else
    Unlock deposits $\mathbf{d}$
    Update users' on-chain states to $\mathbf{S}'$
    Mark TX as FailureProved

Function `VerifyCommit`(TX, $\mathbf{S}$, x, $\mathbf{c}$)**:**
  Compute state transition $\mathbf{S}' = f(\mathbf{S}, x)$
  foreach $c_i \in \mathbf{c}$ **do**
    Extract $(h_{\mathsf{state}}, h_{\mathsf{input}})$ from c_i
    if $H(x) \neq h_{\mathsf{input}}$ **then**
      return FALSE
    if $H(S_i') \neq h_{\mathsf{state}}$ **then**
      if *U_i's transaction before S_i is* Revoked *or* FailureProved **then**
        Call `FailureProve`(TX, $x, \mathbf{S}$)
      else
        Call `Revoke`(TX)
      return FALSE
  return TRUE

that includes TX, $\mathbf{c}$, and $\mathbf{d}$ as well as all users' signatures is submitted to the smart contract by any user (Step 9 does not need to be carried out by all users).

Algorithm 2 demonstrates the miner protocol. After receiving a `Commit` message that contains TX, $\mathbf{c}$, and $\mathbf{d}$, miners lock the deposit for each user, then save the user addresses, commitments, and deposit values on chain. The transaction TX is then marked as Committed.

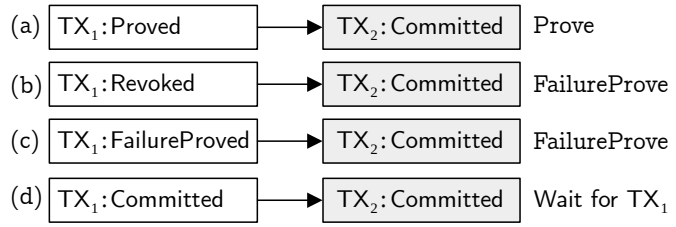


Fig. 6. Transaction predecessor status and the corresponding prove method.

To prove a transaction TX, the predecessors of TX must be proved or revoked, which ensures the execution order of the transactions. Miners use different methods to prove the transaction. Fig. 6 illustrates four cases in proving TX_2 where the statuses of the predecessors differ. Case (a) shows the situation where all predecessors of TX_2 are successfully Proved, and therefore miners `Prove` TX_2 by `VerifyCommit`. Case (b) and (c) indicate that when at least one of the predecessors of TX_2 is either Revoked or FailureProved, miners prove TX_2 by calling `FailureProve` to handle the failure brought by the predecessor. Case (d) demonstrates an example where TX_2 can not be proved as at least one of its predecessors, see TX_1, has a state of Committed. In such a case, TX_1 must be proved or revoked before TX_2 can be proved.

Procedure `Prove` is executed by the miners after receiving TX and the transition input x. Miners first fetch the saved commitments and the users' on-chain states, then call `VerifyCommit` to verify whether TX is ready to be proved. If YES, all users involved in TX have their deposits unlocked and the on-chain states updated, and TX is then marked Proved.

Procedure `VerifyCommit` pays a very critical rule. It takes inputs TX, $\mathbf{S}, x, \mathbf{c}$, where $\mathbf{S}$ is the set of on-chain states of all users associated with TX. First, the temporary states transitioned from $\mathbf{S}$ based on x are computed. Then each commitment of TX, which is stored on-chain, is verified against the hash of x and that the newly obtained temporary state. If there is a mismatch in h_{input}, it means someone who does not know the true value x intends to prove, and the verification terminates while authentic users are still allowed to prove later by submitting the correct x. If the mismatch happens in h_{state}, there are two possible reasons: either at least one predecessor transaction of TX is not successfully Proved, or users collude to commit tampered states which are different from the output of $f(\mathbf{S}, x)$. In the former case, miners call `FailureProve` to handle the failures which will be discussed next. In the latter case, miners call `Revoke` to immediately revoke the commitments without waiting for timeouts as it

would be impossible to prove.

In the situation where the transaction has at least one predecessor transaction Revoked or FailureProved, hash mismatches are detected in `VerifyCommit`. Miners call `FailureProve` to handle such a failure. Specifically, miners transit the fetched users' on-chain states $\mathbf{S}$ with input x to get new states $\mathbf{S}'$. If the transition succeeds, the miners update users' on-chain states as $\mathbf{S}'$ and the transaction TX is marked as FailureProved. Otherwise, miners have to revoke the commitments and apply deposits. Then TX is marked as Revoked in this case.

The `Revoke` procedure is called when required by `VerifyCommit`, or after a timeout event occurs. To revoke a transaction, the miners transfer the deposits to each user as a compensation to "un-do" the state transition, and apply additional penalty to users if required. Moreover, this group of users is considered malicious and therefore is banned to submit latency-first commitments and regular transactions for a relatively long period, so that other users have sufficient time to handle failures by `FailureProve`. Such a consideration can also prevent the malicious users from pushing more malicious commitments that depend on the failure branch for chaos.

Note that since it takes a much longer time to execute `Prove` and `FailureProve` than `Commit`, to make sure `Commit` can be confirmed on chain within about a block interval, a QoS (Quality of Service) mechanism can be implemented in the chain to prioritize `Commit` by reserving a certain amount of block capacity for it, with `Prove` and `FailureProve` being processed at a lower priority. It may also be possible to further optimize this feature at the network level [19].

D. Security Analysis

In our scheme, an adversary $\mathcal{A}$ is able to manipulate locally-stored temporary states, reject to provide the correct proof, and refuse to make a revocation. The goal of $\mathcal{A}$ is to get profit by tempering or canceling state transitions. We focus on the most important attack, dependency attack, where $\mathcal{A}$ first submits commitments about TX_1 with user B colluding with $\mathcal{A}$, then makes another latency-first transaction TX_2 with user U. $\mathcal{A}$ tries to get profit by tempering or canceling TX_2 by not providing the correct proof about TX_1.

Theorem III.1. *Under the assumption that the hash function satisfies the second preimage security, the probability of $\mathcal{A}$ successfully performing the dependency attack is negligible.*

Proof. $\mathcal{A}$ first submits commitments $\mathbf{c}_1 = (c_{\mathcal{A},1}, c_B)$ for transaction TX_1 with user B, $\mathbf{S}'_1 = f(\mathbf{S}_1, x_1)$, $\mathbf{U}_1 = (\mathcal{A}, B)$, $\mathbf{S}_1 = (S_{\mathcal{A}}^{(0)}, S_B^{(0)})$, and $\mathbf{S}'_1 = (S_{\mathcal{A}}^{(1)}, S_B^{(1)})$. Then, $\mathcal{A}$ makes another transaction TX_2 with a user U by committing $\mathbf{c}_2 = (c_{\mathcal{A},2}, c_U)$ and deposits $\mathbf{d}_2 = (d_{\mathcal{A},2}, d_U)$ to miners. $\mathbf{U}_2 = (\mathcal{A}, U)$, $\mathbf{S}_2 = (S_{\mathcal{A}}^{(1)}, S_U^{(0)})$, and $\mathbf{S}'_2 = f(\mathbf{S}_2, x_2) = (S_{\mathcal{A}}^{(2)}, S_U^{(1)})$.

$\mathcal{A}$ tries to perform the dependency attack by three types of attempts:

1) providing $\mathcal{A}$'s tampered temporary state $S_{\mathcal{A}}^{(1),\mathsf{fake}}$ when user U fetches $\mathcal{A}$'s temporary state in a hope that the commitments about TX_2 are mistakenly computed by U;
2) tampering temporary states to get $S_{\mathcal{A}}^{(1),\mathsf{fake}}, S_B^{(1),\mathsf{fake}}$ when $\mathcal{A}$ and B are computing $\mathbf{S}'_1 = f(\mathbf{S}_1, x_1)$. $\mathcal{A}$ and B compute and upload the commitments $c_{\mathcal{A},1}, c_B$ that correspond to the fake temporary states;
3) not providing the proof about TX_1.

For Attempt 1, user U fetches $\mathcal{A}$'s tampered temporary state $S_{\mathcal{A}}^{(1),\mathsf{fake}}$ when locally executing TX_2. U can check whether $H(S_{\mathcal{A}}^{(1),\mathsf{fake}}) = H(S_{\mathcal{A}}^{(1)})$, where $H(S_{\mathcal{A}}^{(1)})$ is obtained on chain. Since the hash function H satisfies the second preimage security, the chance of $\mathcal{A}$ finding a fake state with the same hash is negligible. Therefore, U refuses to process TX_2 and no commitment is uploaded. Our scheme is resistant against the dependency attack in Attempt 1.

For Attempt 2 and Attempt 3, commitments about TX_2 are submitted. No one provides the correct proof about TX_1, and therefore miners revoke TX_1 after timeout. Then later, TX_2 can be either failure-proved or revoked, i.e., if the state transition does not fail, TX_2 is proved under the circumstance where TX_1 is revoked; otherwise, deposit $\mathbf{d}_2$ is applied to revoke TX_2. In either situation, $\mathcal{A}$ does not gain profit. Therefore, our scheme is resistant against the dependency attack in Attempt 2 and Attempt 3. □

IV. Implementation

A. The Programming Tool

We propose a programming tool that facilitates developers in creating latency-first smart contracts following our model. This tool performs lexical and grammatical analysis on the input Solidity file, adds supplementary code to the generated abstract syntax tree, and finally produces a new Solidity file that can be directly deployed on Ethereum or other EVM-compatible blockchain networks. A developer utilizes this tool to define the state S as well as the state transition function f and its input x, counting on our tool to finish the rest:

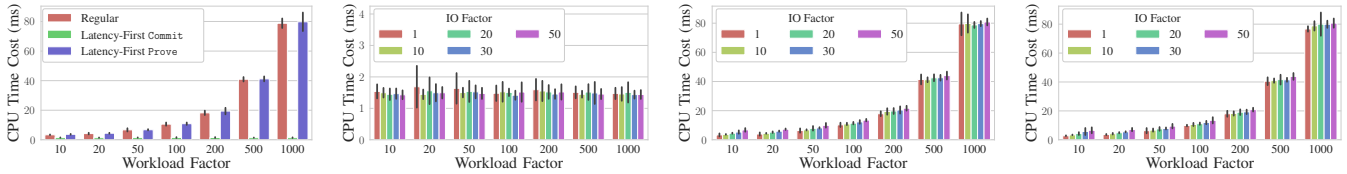
- the contract storage of the online states and commitments;
- the online transactions to directly update the on-chain states if users are not sensitive about latency;
- the commitments and proof transactions allowing users to submit a latency-first commitment and prove later.

In our demonstration, a developer writes about 50 lines of code to support experiments in Sec. IV-B and gets 180 lines of deployable codes.

B. Performance Evaluation

We conduct a series of experiments to evaluate the performance of our latency-first model. We implement a smart contract using our tool in Sec. IV-A, with KECCAK-256[2] as the hash algorithm H. In real-world applications, executing smart contract transactions can be time-consuming due to tasks such as performing elliptic curve operations, calculating multiple hashes, and validating zero-knowledge proofs. To simulate the CPU time required to perform tasks like those

[2]KECCAK-256 implemented in Ethereum has different padding scheme with the final SHA-3 standard.



(a) The time cost of transactions (IO Factor = 10).

(b) The time cost of `Commit` transac-(c) The time cost of `Prove` transaction.(d) The time cost of regular transaction.
tion.

Fig. 7. The time cost of latency-first operations and regular transactions.

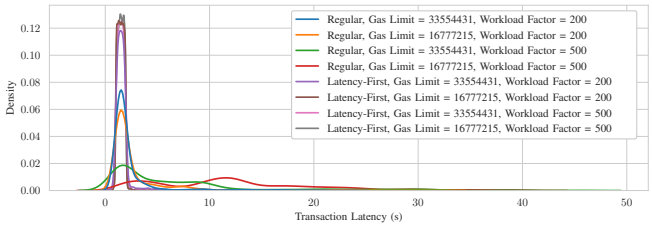


Fig. 8. The kernel density estimation of transaction latency. The probability of having a low latency time is significantly higher in our latency-first scheme.

mentioned above, we define the workload factor in our smart contract, achieved by repeatedly invoking the KECCAK-256 algorithm for a specified number of times. We also define the IO factor to control the number of updated variables during a state transition to simulate the IO impact. Besides, the block gas limit of a blockchain has a direct impact on the number of transactions by limiting the total workload in a block, and therefore has impact on the latency time. In our experiment, we vary these three variables to analyze the time cost and latency distribution under different conditions.

We design the time cost experiments to measure the CPU time cost of `Commit` and `Prove` with different workload factors and IO factors, showing our latency-first model does not have much overhead. Then, we design the latency distribution experiments where a number of users are simulated simultaneously and measure the latency – the time interval between a user submitting a transaction and the transaction gets confirmed into the block after consensus.

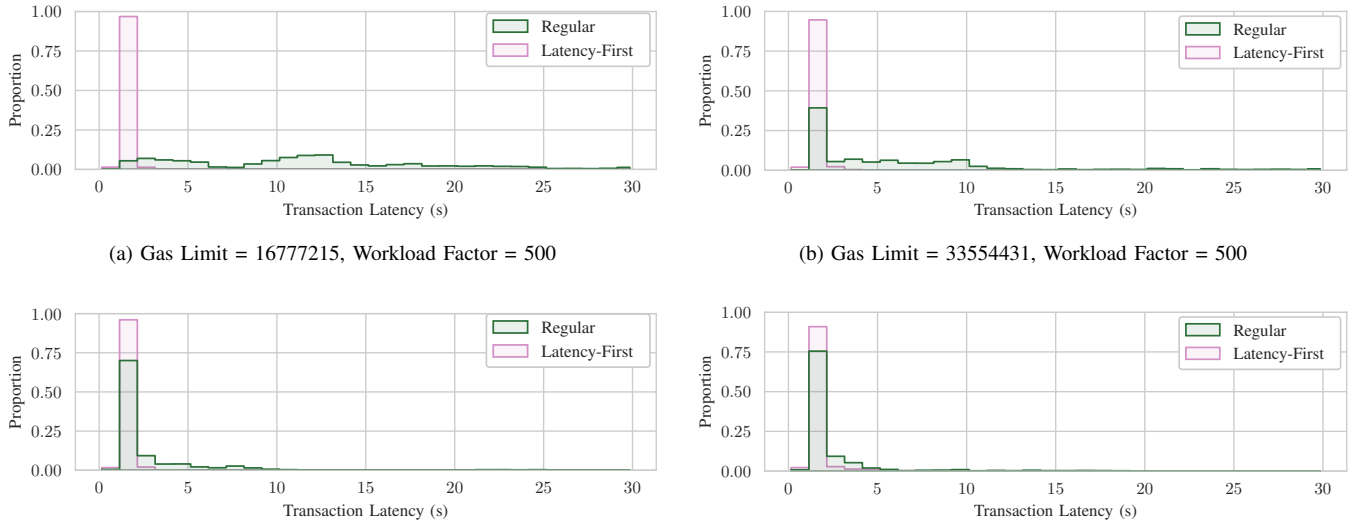
Our experiments are run on a server with two Intel Xeon Silver 4214R CPU @ 2.40 GHz (24 cores in total) and 64 GB RAM, running Arch Linux on GNU/Linux 5.18.0-arch1-1. We run a private Ethereum network with Proof-of-Authority (PoA) consensus and the time interval between two blocks is set to 1 sec. The version of the Ethereum client is v1.10.20 and we make a modification in order to more precisely measure the CPU time cost of a transaction[3].

1) Time Cost: The purpose of this set of experiments is to determine how much overhead is brought by our latency-first model, and how the state transition influences the overhead. The workload factor varies from 10 to $1,000$, and the IO factor varies from 1 to 50. The gas limit is set to $33,554,431$ as we focus on measuring the CPU time cost on a single transaction. Fig. 7 shows the experiment results. Fig. 7(a) compares the time costs between the regular transactions and the `Prove` procedure in our model, and it indicates that the overhead of our model is sufficiently low. Fig. 7(b) indicates that the time cost of a commit transaction is around 1.5 ms, regardless of how the workload factor varies. Compared with the time cost of the regular transactions shown in Fig. 7(d), one can see that it is efficient enough to submit a commitment in our latency-first model, especially when a state transition takes a great amount of time. It can also be inferred that the number of commitments that can be confirmed in a block is much larger, as the time cost of `Commit` in Fig. 7(b) is significantly smaller than that of `Prove` in Fig. 7(c), and therefore latency should be reduced.

2) Latency Distribution: We focus on how exactly our model can reduce latency compared to the regular model. $1,000$ users are simulated and all of them simultaneously submit commitments, with one by each user. The latency time for each commitment is recorded from the time of submission until it is confirmed on-chain. After all of the commitments are confirmed, the users in the same group simultaneously submit regular transactions, with one by each user, and the latency time for each of these transactions is also recorded. We compare the latency times of the two types of transactions. The gas limit is set to $16,777,215$ (low) and $33,554,431$ (high)[4]. In this set of experiments, the IO factor is set to 10, while the workload factor varies from $\{200, 500\}$. Fig. 8 demonstrates the kernel density estimation of transaction latency in all cases. If the y-axis value at a particular point of the x-axis is high, it means that the probability density of latency time at that value is high, which indicates that the latency time is more likely to take on that value. Fig. 8 indicates that applying our latency-first model is effective at reducing latency as the density of the users that only wait for 1 or 2 blocks are much greater than that in the regular smart contract cases. Fig. 9 shows the detailed experiment results. In Fig. 9(a) and Fig. 9(b), the blockchain is heavily overwhelmed with transactions having a larger

[3]In some papers, the time cost of a transaction is misled with its latency, by measuring the time interval between a transaction is submitted through JSON-RPC and the callback is received. This is not accurate since the measurement result is also related to the number of committed transactions and the block interval. Each test case is repeated thirty times.

[4]The higher value is close to the gas limit in the Ethereum public chain since Aug. 2021, and the lower one is close to the public gas limit before Aug. 2021.



(a) Gas Limit = 16777215, Workload Factor = 500

(b) Gas Limit = 33554431, Workload Factor = 500

(c) Gas Limit = 16777215, Workload Factor = 200

(d) Gas Limit = 33554431, Workload Factor = 200

Fig. 9. The latency time distributions for different cases. The latency-first model is able to greatly reduce latency, especially under a high load.

workload factor. For regular transactions, about 50% (low gas limit) or 25% of the users (high gas limit) wait for more than 10 blocks to confirm their transactions, which means that these users suffer from a 10 or 20 times of latency than usual, as when the blockchain has spare capacity, they only need to wait for 1 block. Only 5% (low gas limit) or 40% (high gas limit) regular transactions are confirmed in 1 block. As a contrast, more than 90% latency-first commitments are confirmed in 1 block. Thus, our scheme is able to greatly reduce the latency time and improve user experience compared with the regular smart contract model under a heavily overwhelmed workload. Fig. 9(c) and Fig. 9(d) show a situation where the blockchain is slightly overwhelmed with transactions and the workload factor is smaller. About 70% (low gas limit) or 75% (high gas limit) users wait for only 1 block until their regular transactions are confirmed, while more than 90% latency-first commitments are confirmed in 1 block. Our scheme is also able to reduce the latency time when the blockchain is slightly overwhelmed.

In summary, after applying our latency-first smart contract model, most users are able to get their transactions confirmed in 1 or 2 blocks, as `Commit` takes little overhead compared to the regular transactions. The users prove the commitments at a later time, but in this situation, the latency does not matter much, since they are able to submit nested commitments before proving the transactions the new commitments depend on. The experiment results indicate that our model is able to greatly reduce the latency time especially when the workload is high; and therefor is particularly suitable for blockchains with imbalanced spare capabilities over time.

V. Conclusion and Future Research

We propose a latency-first smart contract model in this paper, allowing users to submit commitments during the heavy-load time and then prove them later during the spare time. The experiment results indicate that our proposed latency-first smart contract model is able to significantly reduce the latency when the blockchain is under a heavy load. Therefore, our model is able to improve the user experience and balance the blockchain computational resources, making it robust against high-frequent requests within a short time. We have provided public access to our code and data at https://github.com/SDU-IIC-Blockchain/latency-first-smart-contract-repos. More works can be done to improve our latency-first smart contract model, such as allowing a user to check all related state transitions and get the ability to prove all dependent transactions, extending our latency-first smart contract to support privacy-preserving scenarios, and implementing a global state management model to support variables that do not have a user ownership.

Acknowledgement

This work was partially supported by the National Natural Science Foundation of China (62232010) and the Blockchain Core Technology Strategic Research Program of Ministry of Education of China (2020KJ010301). The authors thank the developers of seaborn [20], solidity-parser[5], and prettier-plugin-solidity[6] for making their codes available on a free and open-source basis.

References

[1] T. Beiko, "Gas and fees," Jul 2022, (Date accessed: July 23, 2022). [Online]. Available: https://ethereum.org/en/developers/docs/gas/

[2] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, "Performance characterization of hyperledger fabric," in *2018 Crypto Valley conference on blockchain technology (CVCBT)*. IEEE, 2018, pp. 65–74.

[5]https://github.com/solidity-parser/parser

[6]https://github.com/prettier-solidity/prettier-plugin-solidity

[3] M. Kuzlu, M. Pipattanasomporn, L. Gurses, and S. Rahman, "Performance analysis of a hyperledger fabric blockchain framework: throughput, latency and scalability," in *2019 IEEE international conference on blockchain (Blockchain)*. IEEE, 2019, pp. 536–540.

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

[5] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *IEEE INFOCOM*, 2022.

[6] A. Valtchanov, L. Helbling, B. Mekiker, and M. P. Wittie, "Parallel block execution in socc blockchains through optimistic concurrency control," in *2021 IEEE Globecom Workshops (GC Wkshps)*, 2021, pp. 1–6.

[7] J. Qi, X. Chen, Y. Jiang, J. Jiang, T. Shen, S. Zhao, S. Wang, G. Zhang, L. Chen, M. H. Au, and H. Cui, "Bidl: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 18–34. [Online]. Available: https://doi.org/10.1145/3477132.3483574

[8] Z. Chen, H. Zhuo, Q. Xu, X. Qi, C. Zhu, Z. Zhang, C. Jin, A. Zhou, Y. Yan, and H. Zhang, "Schain: A scalable consortium blockchain exploiting intra- and inter-block concurrency," *Proc. VLDB Endow.*, vol. 14, no. 12, p. 2799–2802, jul 2021. [Online]. Available: https://doi.org/10.14778/3476311.3476348

[9] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 949–966.

[10] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry, "Sprites and state channels: Payment networks that go faster than lightning," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 508–526.

[11] M. M. Chakravarty, S. Coretti, M. Fitzi, P. Gazi, P. Kant, A. Kiayias, and A. Russell, "Hydra: Fast isomorphic state channels," *Cryptology ePrint Archive*, 2020.

[12] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, pp. 1–47, 2017.

[13] K. Floersch, "Ethereum smart contracts in l2: Optimistic rollup," 2019, https://medium.com/plasma-group/ethereum-smart-contracts-in-l2-optimistic-rollup-2c1cef2ec537.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 45–58. [Online]. Available: https://doi.org/10.1145/1294261.1294267

[15] G. Ateniese, B. Magri, D. Venturi, and E. Andrade, "Redactable blockchain – or – rewriting history in bitcoin and friends," in *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, 2017, pp. 111–126.

[16] D. Deuber, B. Magri, and S. A. K. Thyagarajan, "Redactable blockchain in the permissionless setting," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 124–138.

[17] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[18] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[19] D. Yang, K. Gong, J. Ren, W. Zhang, W. Wu, and H. Zhang, "Tc-flow: Chain flow scheduling for advanced industrial applications in time-sensitive networks," *IEEE Network*, vol. 36, no. 2, pp. 16–24, 2022.

[20] M. L. Waskom, "seaborn: statistical data visualization," *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021. [Online]. Available: https://doi.org/10.21105/joss.03021