

# ShuttleCross: An Efficient Cross-Chain Smart Contract Invocation Framework

Rongkai Zhang<sup>1</sup>, Qiuyu Ding<sup>1</sup>, Qianyi Liu<sup>1</sup>, Shengjie Guan<sup>1</sup>, Zhen Xiao<sup>1,†</sup>, and Jieyi Long<sup>2</sup>

<sup>1</sup>School of Computer Science, Peking University, Beijing, China

<sup>2</sup>Theta Labs, Inc., San Jose, USA

<sup>1</sup>rkzhang@stu.pku.edu.cn, dingqiuyu@stu.pku.edu.cn, qianyiliu25@stu.pku.edu.cn,  
guanshengjie@stu.pku.edu.cn, xiaozhen@pku.edu.cn, <sup>2</sup>jieyi@thetalabs.org

**Abstract**—Cross-chain smart contract invocation enables interoperability among different blockchain service platforms. With the growing demand for cross-chain services, the performance of cross-chain smart contract invocations is becoming increasingly important. However, current cross-chain smart contract invocation technologies suffer from long latency and high abort rates. In this paper, we present ShuttleCross, an efficient cross-chain smart contract invocation framework, which ensures the atomicity and serializability of cross-chain transactions. ShuttleCross employs a hybrid concurrency control protocol to reduce the high abort rates caused by read-write conflicts of concurrent cross-chain transactions. Furthermore, ShuttleCross adopts a read-write separation strategy to accelerate cross-chain transaction execution by executing read-only function invocations off-chain. Our experimental results demonstrate that ShuttleCross significantly improves the performance of cross-chain transaction execution.

**Index Terms**—Blockchain, cross-chain, smart contract, concurrency control.

## I. INTRODUCTION

Blockchain technology has been widely used in various domains, particularly in finance [1], healthcare [2], and the Internet of Things [3]. With numerous specialized chains deployed to serve various purposes, the need for interaction between these isolated blockchains becomes increasingly important [4], [5]. For instance, the Cosmos Network [6] facilitates such ecosystems with its stack for application-specific blockchains. Cross-chain technology [7], [8] enables interaction between distinct blockchains. Existing approaches such as sidechains [9], notary schemes [10], and hashed time lock contracts (HTLC) [11] provide interoperability but focus on token exchange, aiming to eliminate the need for centralized exchanges [12]. Cross-chain smart contract invocation technology enables seamless interaction with smart contracts [13] across different blockchains, promoting a more interconnected blockchain ecosystem.

As illustrated in Figure 1, the loan process requires cooperation across multiple blockchains. An importer needs to secure a loan from a bank and complete a purchase order. The importer submits a cross-chain transaction to the bank chain and invokes the *applyLoan* function. The *applyLoan* function subsequently interacts with the market chain to verify the order’s existence and queries the supply chain to ensure the supplier can fulfill the order. Upon receiving valid verification results, the bank chain invokes the *approveLoan* function,

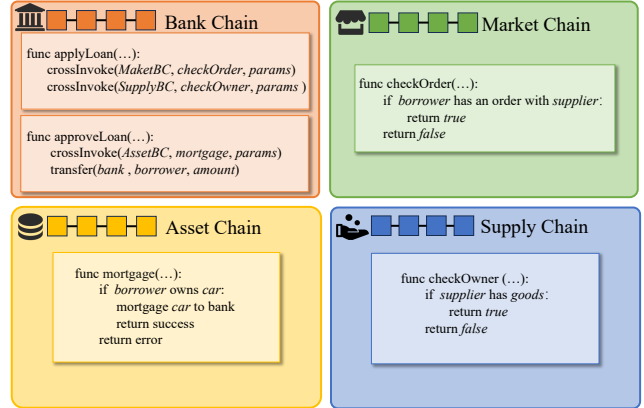


Fig. 1: An example of cross-chain smart contract invocation in a loan process.

which triggers the *mortgage* function on the asset chain to pledge the importer’s car as collateral to the bank. After that, the bank chain disburses the loan funds to the importer. This example demonstrates the interaction between contract functions across different blockchains to execute complex transaction logic.

In the context of cross-chain smart contract invocations, two critical properties must be ensured: atomicity and serializability [14]. Atomicity ensures that all invocations in the same cross-chain transaction either succeed or fail together. Since multiple concurrent cross-chain transactions may lead to potential read-write contention, serializability guarantees that concurrent cross-chain transactions behave as if executed sequentially.

While prior works [15]–[18] focus on achieving atomicity and serializability, they fail to improve the efficiency of cross-chain smart contract invocations with two major challenges. The first challenge arises from using a single concurrency control strategy that does not distinguish the levels of state contention. For instance, pessimistic strategies represented by strict two-phase locking (S2PL), ensure cross-chain transaction isolation through locking mechanisms. But, this results in lock contention and sacrifices potential concurrency opportunities. In contrast, optimistic concurrency control (OCC) strategies easily lead to higher abort rates under contention-intensive workloads. The second challenge is the long latency involved in completing cross-chain transactions. Cross-chain transac-

tions require multiple cross-chain smart contract invocations. However, existing approaches require all cross-chain invocations to wait in transaction pools for consensus, thereby causing high latency in cross-chain transactions.

To address these challenges, we propose ShuttleCross, an efficient framework for cross-chain smart contract invocation that ensures atomicity and serializability. To mitigate the first challenge, ShuttleCross introduces a hybrid concurrency control (HCC) strategy, reducing cross-chain transaction abort rates and enhances concurrency. Existing HCC schemes in database assume a single coordinator with global visibility of states and conflicts. Such assumptions do not hold in cross-chain settings, where blockchains maintain separate consensus processes and state logs without shared scheduling or lock management. ShuttleCross is the first to realize adaptive HCC under cross-chain 2PC coordination across independent consensus domains, ensuring local OCC and S2PL decisions compose into a globally serializable execution. Achieving this requires addressing challenges that do not arise in traditional single-domain transactional systems, including speculative dirty-state management across chains, distributed deadlock detection and resolution, safe state-type transitions under in-flight transactions, and commit-phase validation under Byzantine-safe message exchange.

Besides, we observe that queuing for consensus dominates cross-chain transaction latency. To mitigate the second challenge, ShuttleCross implements a read-write separation strategy that allows read-only invocations to bypass queuing and consensus processing, significantly reducing latency. However, relying on naive off-chain reading lead to non-serializable execution. So we design a dual-processing mechanism validates read-only results at the commit phase to guarantee serializability.

The main contributions of this paper are summarized as follows:

- We propose ShuttleCross, a cross-chain smart contract invocation framework that improves throughput and reduces latency while ensuring atomicity and serializability.
- ShuttleCross introduces a hybrid concurrency control mechanism that reduces cross-chain transactions abort rates and improves execution concurrency.
- ShuttleCross adopts a read-write separation strategy to reduce the latency of cross-chain smart contract invocations, while preserving execution correctness.
- We implemented the ShuttleCross on ChainMaker [19], a mature open-source blockchain platform, and conducted experiments to evaluate its performance. The results indicate that ShuttleCross can reduce the latency by 58% and increase the throughput by 55% compared to GPACT [16].

## II. BACKGROUND AND RELATED WORK

### A. Cross-Chain Smart Contract Invocation

Cross-chain smart contract invocation (CCSCI) allows users to submit their cross-chain transactions to an origin chain,

which then automatically triggers the required invocations across multiple blockchains. Current CCSCI technologies can be broadly categorized into RPC-like invocation, 2PC-based methods, and smart contract portability.

In RPC-like invocation methods, such as TrustCross [20], Cosmos/IBC [6] and Polkadot/XCMP [21], a smart contract function in origin blockchain asynchronously invokes a contract function hosted on another target blockchain and receives return values through a callback mechanism. However, these methods lack mechanisms to guarantee the atomicity of complex cross-chain transaction execution.

The 2PC-based mechanism is employed in 2PC4BC protocols [15] and GPACT [16] to ensure the atomicity of cross-chain transactions. When a user submits a cross-chain transaction, a client or a chain acts as a 2PC manager and coordinates the involved target chains to execute the cross-chain invocations. During this execution process, the involved chains lock the accessed states or maintain a record of the read/write set. Once all the involved chains have completed their execution, the manager receives the outcomes and decides whether to commit or abort the cross-chain transaction. Subsequently, the manager sends the commit or abort messages to the involved chains. Although this mechanism ensures the atomicity and integrity of the entire cross-chain transaction, the complex multi-step cross-chain coordination significantly degrades the overall execution efficiency.

Westerkamp [17] and Canton [22] adopt the concept of smart contract portability to guarantee atomicity of cross-chain transactions. They migrate all involved contracts and states of cross-chain transactions to the same blockchain before execution so that the atomicity of single-chain transactions is utilized. After the execution, the modified states will be written back to the origin chains. This scheme requires chains to be similar types because all participating smart contracts must be deployable on any involved chain. Besides, frequent migration of contracts and states also results in increased inter-chain communication overhead.

### B. Cross-Chain Concurrency Control

Concurrent cross-chain transactions execution leads to state contention, where multiple cross-chain transactions attempt to read or write the same state. State contention can compromise transaction isolation and produce unexpected execution results. Effective cross-chain concurrency control is therefore essential to manage contention and improve the efficiency and reliability of cross-chain transactions. However, existing approaches typically employ a single concurrency control strategy without distinguishing state contention levels. This can lead to inefficiencies, as a one-size-fits-all approach is not effective for all scenarios.

2PC4BC and GPACT employ the S2PL mechanism to ensure cross-chain transaction serializability. When a cross-chain transaction requires access to states with S2PL, it must acquire locks on the states. However, if a transaction attempts to lock a state that is already locked by another transaction, the entire cross-chain transaction is immediately aborted. While

this locking strategy prevents potential distributed deadlocks by eliminating lock wait scenarios, it increases the cross-chain transaction abort rate and reduces the system throughput. Besides, this mechanism also sacrifices potential concurrency opportunities for low contention states.

EOVPC [23] and Avalon [18] employ OCC strategy for cross-chain transactions. The OCC strategy eliminates the need for locks during transaction execution. It allows more cross-chain transactions to execute in parallel and improves the throughput when the state contention rate is relatively low. However, when multiple cross-chain transactions frequently access the same data, OCC may result in frequent conflicts, leading to many cross-chain transaction aborts.

### III. SYSTEM MODEL

In our system model, we assume a finite set of  $n$  blockchains denoted as  $\{B_1, B_2, B_3, \dots, B_n\}$ , where each blockchain operates under a consensus protocol that tolerates byzantine faults [24]. Additionally, we assume there exists a secure public key infrastructure and a signature scheme that enables the verification of signatures generated by the consensus committee of each blockchain. In a cross-chain transaction, the involved chains are categorized as origin chain and target chains. The origin chain refers to the blockchain where the user submits the cross-chain transaction, such as the bank chain in the Figure 1. Therefore, each transaction dynamically designates its origin chain based on the underlying business logic. The target chains are the other blockchains that need to collaborate to complete the cross-chain transaction, such as market chain and asset chain.

The malicious nodes within each blockchain may exhibit arbitrary behavior, such as dropping, omitting, or reordering messages during interactions with other nodes within the same blockchain or across different blockchains. To ensure the security and consistency guarantees, we assume that each blockchain consists of  $N$  consensus nodes, of which at most  $f$  nodes may behave maliciously. This setup requires that  $N \geq 3f + 1$ , satisfying the standard resilience condition for byzantine fault tolerance [25].

The nodes on different blockchains can directly exchange cross-chain messages through a peer-to-peer network. To ensure the correctness of these cross-chain messages, we follow the direct signing mechanism used in [16], [26]. The direct signing technique requires the validator sets of each blockchain to be registered on all other participating blockchains to perform cross-chain message verification. Whenever the validator set change, the updated set is encapsulated within a new block and synchronized to the registry via cross-chain messages. By implementing this registry as a smart contract, the storage overhead for the validator set is limited to merely a few hundred KB per chain.

When a node  $n_i \in B_i$  interacts with nodes in another blockchain  $B_j$ , it accepts the remote information once it receives authenticated replies from an appropriate threshold of validators on  $B_j$ . For messages that originate from the finalized consensus of  $B_j$ , such as on-chain execution results or

cross-chain invocation events,  $n_i$  only needs  $f + 1$  signatures, since at least one must be honest. In contrast, reading the on-chain states of  $B_j$  requires stronger guarantees, so  $n_i$  accepts it only with  $2f + 1$  authenticated replies, ensuring consistency with the unique state agreed upon by a supermajority of validators.

We model the communication networks both within each blockchain and between blockchain nodes as partially synchronous [27]. Specifically, we assume there is a known bound  $\Delta$  and an unknown Global Stabilization Time (GST). After GST, all messages sent by an honest node arrive within a bounded time  $\Delta$ .

If a transaction invokes a smart contract that subsequently calls contract functions across multiple blockchains according to contract logic, it requires several cross-chain interactions for completion. We define such a transaction as a cross-chain transaction, denoted as  $CTX$ , which consists of multiple cross-chain smart contract invocations, i.e.,  $CTX := \{invoke_i | i = 1, 2, \dots, k\}$ . Each invocation  $invoke_i$  is relayed to corresponding target chain and invokes the contract function to read or write on-chain states. The execution of  $CTX$  is deemed successful only if all involved invocations are successfully executed and committed on respective blockchains.

In addition, addressing the gas fees incurred across multiple chains is a practical necessity for cross-chain transactions. We adopt the direct deduction model, where the transaction initiator is responsible for directly paying the gas fees for the invocations on each chain. Therefore, the initiator must ensure they have sufficient tokens on each chain to cover the respective gas fees. Besides, several existing cross-chain protocols adopt a relayer-based model, where the initiator pays a consolidated fee on the source chain, and a designated relayer covers the gas fees on the target chains on their behalf [28]. ShuttleCross does not depend on any specific fee mechanism. It seamlessly supports both models, since the underlying fee logistics are entirely orthogonal to the core contributions of ShuttleCross.

### IV. SHUTTLECROSS DESIGN

#### A. Overview of ShuttleCross

ShuttleCross employs a two-phase commit (2PC) protocol to guarantee the atomicity of cross-chain transactions. During the first phase, the origin chain collaborates with multiple target chains to execute multiple cross-chain invocations. In this phase, we introduce two key innovations to improve efficiency: the hybrid concurrency control strategy and the read-write separation mechanism. The hybrid concurrency control strategy adaptively selects suitable concurrency protocols based on state contention levels to maximize cross-chain transaction concurrency and minimize abort rate. Furthermore, the read-write separation strategy reduces latency for read-only cross-chain invocations by skipping traditional transaction pool queuing and consensus processes.

During the second phase, the origin chain validates whether the cross-chain transaction can be committed. Upon successful verification, the origin chain decides to commit the cross-chain

transaction and notifies all involved target chains to commit it.

### B. Atomicity Guarantee through 2PC

Approaches to ensure atomicity of cross-chain transactions include smart contract portability, and 2PC-based protocols [4]. However, the smart contract portability approach suffers from inefficiency and compatibility issues across different blockchain networks. The 2PC-based protocol is more suitable for complex cross-chain interaction scenarios, offering greater applicability and efficiency.

In ShuttleCross, the origin chain, which receives the user's CTX, coordinates all participating chains and determines the final transaction commitment to ensure atomicity the cross-chain execution. Different transactions may be submitted to different origin chains, allowing the coordination workload to be naturally distributed across the system.

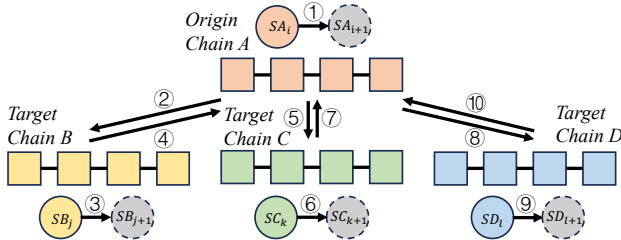


Fig. 2: The first phase of 2PC. The numbered labels represent the order of invocation and communication steps.

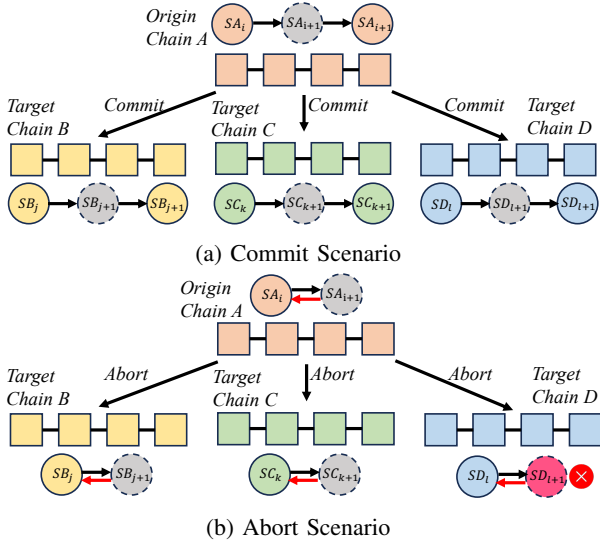


Fig. 3: The second phase of 2PC. Red arrows indicate the reverting of dirty states during the abort process.

In the first phase, the origin chain executes CTXs and emits events containing invocation parameters to trigger operations on target chains. Then each involved target chain executes the requested invocation. If the invocation writes to the blockchain state, the modified state is stored temporarily rather than immediately committed. We call the temporarily stored state the “cached dirty state”. Target chains then return the execution

results back to the origin chain and wait for the commitment. The origin chain processes these results and may trigger subsequent cross-chain invocations, enabling multi-step cross-chain workflows. As depicted in Figure 2, *chainA* serves as the origin chain, while chains *B*, *C*, and *D* are the target chains. The  $SA_i, SB_j, SC_k$ , and  $SD_t$  represent the states touched by invocations on each chain, with the subscripts indicating their version numbers. When executing invocations, target chains might need to initiate additional sub-inocations to other chains, potentially leading to recursive invocations. Once all execution of the sub-inocations are finished, the target chain returns the results back to its origin chain.

In the second phase, the origin chain decides whether to commit the CTX. As illustrated in Figure 3a, if all cross-chain invocations are successful, the origin chain sends its decision to commit the CTX to the target chains. Conversely, if any chain fails to execute the invocation, the origin chain will decide to abort the CTX. As shown in Figure 3b, when *ChainD* encounters an error during the execution, the origin chain notifies all target chains to revert the cached dirty states, such as  $SA_{i+1}, SB_{j+1}, SC_{k+1}$ , and  $SD_{t+1}$ . This mechanism guarantees atomicity in CTXs, ensuring all participating chains commit or abort uniformly.

Each participating blockchain is solely responsible for maintaining its own temporary states. Blockchains only transmit invocations and execution results without exchanging state management details. Therefore, our approach supports seamless cross-chain interactions between heterogeneous state models and distinct contract languages, without affecting local state management.

Moreover, ShuttleCross ensures the liveness of CTXs by implementing a timeout mechanism. This prevents indefinite waiting of participating chains even in the face of blockchain crashes and network disruptions. The blockchain crash refers to a temporary liveness failure where the blockchain network temporarily halts block production and cannot reach consensus. This typically occurs in a partially synchronous network due to the disconnection of a fraction of nodes or the inactivity of the leader node. The origin chain employs timers based on block height for each CTX, with expiration triggering transaction abortion if execution results are not received within the designated block intervals.

When the origin chain becomes temporarily unavailable, or when target chains do not receive commit or abort decisions for an extended period, the target chains continuously poll the origin chain to query the CTX status until a final decision is obtained. This polling mechanism preserves the liveness of CTXs and prevents target chains from making premature commit or rollback decisions that would violate atomicity. To avoid extra communication overhead, polling is rate-limited by block production. Since polling occurs infrequently, its bandwidth cost is much smaller than that of normal cross-chain invocations.

### C. Hybrid Concurrency Control Strategy

CTXs also require effective concurrency control. Since OCC and S2PL each have their own advantages [29], [30], relying on a single concurrency control strategy limits the performance optimization of cross-chain execution. Therefore, we propose the Hybrid Concurrency Control (HCC) mechanism that dynamically and adaptively applies appropriate concurrency control strategies to different states when executing cross-chain invocations.

In ShuttleCross, the states are classified into two types: *occStates* and *lockStates*. The *occState* with low access contention is accessed using OCC to improve concurrency in cross-chain transactions. In contrast, the *lockState* that may experience frequent read-write conflicts is managed with S2PL to reduce the high abort rate. Algorithm 1 outlines the process flow for HCC. Here, the *ctxId* is the unique id of cross-chain transaction, *sId* is a unique identifier for the state *s*, and *Stype* stores the type of state. The *SO* and *SL* are used to store *occState* and *lockState*, respectively.

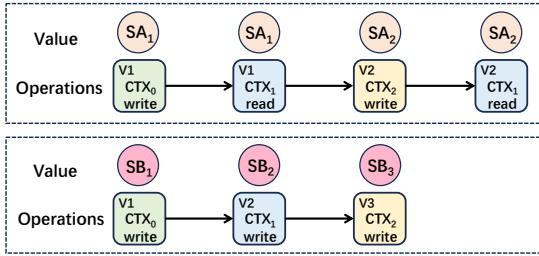


Fig. 4: Examples of *opList* for states *SA* and *SB*.

1) *OCC Component*: We first introduce the OCC component of HCC. The OCC mechanism eliminates the necessity for state locking by maintaining read and write operation histories, making it particularly effective for low contention states while enhancing cross-chain transaction concurrency. In our core design, each *occState* maintains an append-only operation list (*opList*) that caches both read and write operations with their intermediate results, as illustrated in Figure 4. When executed invocations require accessing *occStates*, new state operations and their corresponding values are appended to the *opList* (Algorithm 1, Line 9), with each new state value being computed based on preceding operations. The operations on *occStates* can be executed immediately and the execution results obtained on the target chain can be directly returned to origin chain, enhancing the overall concurrency of cross-chain transactions.

To ensure version consistency, each write operation receives a monotonically increasing version number, while read operations inherit the sequence number of the most recent preceding operation. The operations within the same *opList* establish dependencies: the operations of an *occState* in the *opList* can only be committed when all preceding operations have been committed. If  $CTX_i$  performs operations on multiple *occStates*, it can only be committed when all these operations are positioned at the front of the *opList*. Notably, the read operations with same version number are considered to occupy

the same position in the *opList*. Then the target chain will notify the origin chain that this operation can be committed safely. Once the operations are formally committed, they will be removed from the front of the *opLists*, as depicted in Line 35 Algorithm 1.

---

#### Algorithm 1 Execute CCSCI with HCC

---

```

1: Data: Stype[], SO[], SL[], ReExecList[]
2: Note: All entries in Stype[] are initialized to occState
3: procedure HANDLEINVOKE(invoke)
4:   rwSet, result  $\leftarrow$  execute(invoke)
5:   ctxId  $\leftarrow$  invoke.ctxId
6:   for s in rwSet do
7:     sId  $\leftarrow$  s.sId
8:     if Stype[sId] = lockState and s has no write
locks in  $\lambda$  blocks then
9:       convert SL[sId].locklist to SO[sId].opList
10:      Stype[sId]  $\leftarrow$  occState
11:    end if
12:    if Stype[sId] = occState then
13:      SO[sId].opList append (ctxId, s.op, s.v)
14:      if any conflict in SO[sId].opList then
15:        CTXs  $\leftarrow$  SO[sId].opList.conflict()
16:        emit_event("abort", CTXs)
17:        Stype[sId]  $\leftarrow$  lockState
18:        return
19:      end if
20:    else if Stype[sId] = lockState then
21:      if getLock(ctxId, s.op) = Fail then
22:        SL[sId].lockList append (invoke, s.op)
23:        undo(invoke)
24:        return
25:      end if
26:    end if
27:  end for
28:  return rwSet, result
29: end procedure
30: procedure HANDLECOMMIT(ctxId)
31:   states  $\leftarrow$  getStates(ctxId)
32:   for s in States do
33:     sId  $\leftarrow$  s.sId
34:     if s.type = occState then
35:       SO[sId].opList clean operations of ctxId
36:     else
37:       SL[sId].lockList.unlock()
38:       invoke  $\leftarrow$  SL[sId].lockList.front()
39:       ReExecList append invoke
40:     end if
41:   end for
42: end procedure

```

---

However, the OCC approach introduces the potential for cascading aborts under conditions of high state contention. In the OCC method, each cross-chain transaction requires that the accessed states remain unmodified by other concurrent transactions until it commits; otherwise, it must roll back. As

illustrated in Figure 4, the value of  $SA$  changes between the first and second read operations of  $CTX_1$  due to modifications by  $CTX_2$ , resulting in the abortion of  $CTX_1$ . Consequently, all transactions that follow  $CTX_1$  in the  $opList$  must also be aborted. To detect such state read-write conflicts as early as possible, each chain is tasked with identifying potential read-write conflicts within the  $opList$  whenever a new operation is added, as depicted in Line 14 of Algorithm 1. Upon detecting conflicts, the entire corresponding cross-chain transaction will be aborted.

2) *S2PL Component*: The S2PL component of HCC is employed for high contention states and requires that the accessed states be locked during the execution of cross-chain transactions. Different  $CTXs$  can share read locks but not write locks. Unlike conventional cross-chain S2PL methods in [15], [16] that directly abort cross-chain transactions which fail to acquire state locks, we introduce a novel design where each  $lockState$  maintains a  $lockList$  to cache the invocations that cannot immediately acquire state locks, as shown in Figure 5. To prevent complex queue starvation issues, we strictly manage the  $lockList$  using a First-In-First-Out (FIFO) mechanism. Meanwhile, to maximize the system’s concurrency performance, consecutive read requests at the head of the queue are permitted to acquire read locks and access the state concurrently.

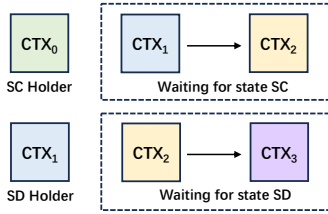


Fig. 5: Examples of  $lockList$  for states  $SC$  and  $SD$ .

The primary difference from the  $opList$  in OCC is that invocations added to the  $lockList$  are not executed immediately. Instead, they wait until the locks can be acquired. After a cross-chain invocation is committed, it releases all held locks through `unLock` function, and the queued invocations in the  $lockLists$  become the new lock holders, as shown in Line 38 of Algorithm 1. The invocations are then added to the  $ReExecList$  for re-execution, as depicted in Line 39. When the next block is processed, the invocations in  $ReExecList$  are executed in the same manner as regular invocations. This design enables target chains to delay execution of cross-chain invocations until resource availability is guaranteed. Under high state contention, this queuing strategy significantly reduces transaction rollback probability.

3) *Transaction Commitment*: A  $CTX$  may access  $occStates$  and  $lockStates$  on multiple chains under our HCC algorithm. For  $lockStates$ , the  $CTX$  reads and writes only after acquiring the lock, and it holds all acquired locks until commit. For  $occStates$ , the  $CTX$  checks conflicts while appending operations to the  $opList$ . If it finishes without rollback, it passes OCC validation. The  $CTX$  can commit only when its operations on every accessed  $occState$  are

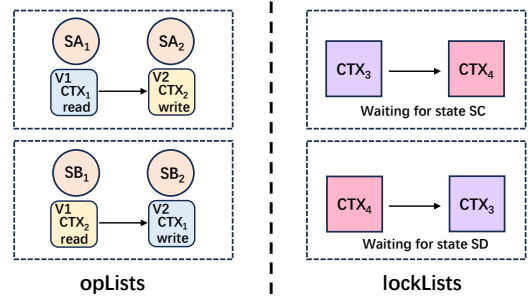


Fig. 6: Deadlock scenarios involving  $CTX_1-CTX_2$  and  $CTX_3-CTX_4$ .

at the front of the corresponding  $opList$ , indicating that all predecessors have committed. Upon commit, it releases all the locks and removes its operations from the  $opList$ .

Besides, a native single-chain transaction ( $STX$ ) and an in-flight  $CTX$  may access the same state concurrently. To ensure serializability,  $STXs$  follow the same HCC coordination. If there is no predecessor in  $opList/lockList$ ,  $STXs$  commit immediately. Otherwise, they wait for predecessors to commit or wait for locks to become available before committing. Consequently, this coordination overhead leads to a reduction in  $STX$  throughput and an exacerbation of tail latency.

4) *Deadlock Detection*: Both OCC and S2PL strategies in HCC introduce the risk of potential deadlock. As illustrated in Figure 6,  $CTX_1$  and  $CTX_2$  wait circularly on  $occStates$   $SA$  and  $SB$ , and  $CTX_3$  and  $CTX_4$  are caught in a circular wait for  $lockStates$   $SC$  and  $SD$ . Although such deadlock will ultimately be aborted due to timeout mechanisms as mentioned in Section IV-B, the waiting period still incurs considerable overhead.

To address this issue, ShuttleCross performs deadlock detection after each block is executed. During block execution, each chain constructs a transaction wait-for graph  $G = (V, E)$ , where  $V$  is the set of vertices representing cross-chain transactions, and  $E$  is the set of edges representing wait-for relationships. After a transaction is committed, its corresponding vertex along with all associated edges is removed from the wait-for graph. Upon completing each block, chains exchange their local wait-for graphs and merge them into a global wait-for graph. If a cycle is detected in the graph, it indicates the existence of deadlock in the system. To break the cycle, the most recently issued transaction in the cycle is aborted. The origin chain of the aborted transaction is responsible for initiating the abort procedure.

5) *State Classification Strategy*: Although we have clarified the processing mechanisms for  $occState$  and  $lockState$ , as well as deadlock handling, there still lacks a principled methodology for state classification. To address this gap, we develop a dynamic state classification strategy based on the recent state contention situation. This strategy supports flexible state conversion during the execution of cross-chain transactions to dynamically optimize transaction efficiency.

Initially, each state is categorized as an  $occState$ . If a transaction aborts due to a conflict when accessing a state,

we infer a high likelihood of recent contention for this state and convert it as a *lockState*, as illustrated in Line 17 of Algorithm 1. The subsequent cross-chain invocations requiring access to this state must acquire the lock. If an *occState* contains uncommitted operations in its *opList* and needs to transition to a *lockState*, it must wait until all operations in the *opList* are committed before processing the *lockList*.

When a *lockState* exhibits low contention, it can be reclassified as an *occState*. If a *lockState* does not hold any write lock and does not receive any write lock requests within  $\lambda$  blocks, we infer a statistically reduction in contention likelihood and reclassify it as an *occState*. We need to convert all read locks of this state into read operations and add them to the *opList*, as depicted in Lines 8-11 of Algorithm 1. Subsequent cross-chain invocations will then access this state with OCC strategy, thereby enhancing cross-chain transaction concurrency.

Since this classification strategy relies on recent historical statics to predict future workloads, abrupt workload shifts inevitably introduce a temporary protocol mismatch. An unexpected contention spike causes temporary high abort rates under optimistic control, whereas a sudden drop in contention incurs unnecessary locking overhead under pessimistic control. Besides,  $\lambda$  relies on empirical tuning based on offline profiling rather than a universal setting method. To address this limitation, we plan to explore reinforcement learning in our future work to enable the system to adaptively adjust control parameters.

#### D. Read-Write Separation Strategy

Although the design of HCC mechanism reduces the abort rate and enhances concurrency for cross-chain transactions, there remains an inherent challenge of significant latency. Specifically, each cross-chain invocation must queue in the transaction pool on the target blockchain, awaiting execution and consensus, leading to increased latency. By bypassing the transaction pool and the associated consensus waiting periods for certain invocations, we can accelerate execution and facilitate earlier triggering of subsequent invocations, thereby reducing the overall latency of cross-chain transactions. Therefore, we propose a read-write separation strategy that categorizes cross-chain invocations into two types: normal invocations and read-only invocations.

The execution of normal invocations remains unchanged as described in Section IV-C. Read-only invocations call smart contract functions that read *occState* without modifying the on-chain state. For read-only invocations, we bypass the transaction pool waiting and consensus process on target chain. We exclusively restrict this acceleration to *occStates* because *lockStates* inherently carry a higher probability of read-write conflicts. If a read-write separation were applied to access *lockStates* without locking, it would severely exacerbate these conflicts, subsequently increasing the overall transaction abort rate.

Upon receiving a read-only invocation, the nodes in the target chain can independently read the latest state value

from the *opList* and execute it off-chain immediately, without waiting for consensus. Then each node in the target chain transmits the execution result back to the origin chain. Once the origin chain successfully receives the execution results of read-only invocation, it can directly trigger the subsequent cross-chain invocations, thereby reducing the overall execution time of the cross-chain transaction.

Before committing cross-chain transactions involving read-only invocations on *occStates*, two key conditions must be verified by origin chain: First, no transaction has committed any updates to the involved *occStates* since the read-only invocation was executed. Second, all write operations on which the read operations depend have already been committed. We propose a simple strawman strategy to address this problem and outline its weaknesses.

In the strawman strategy, the nodes on the target chain, during off-chain execution, record the accessed state in a read set, represented as  $readSet := \{(s_i, ver_i) \mid i = 1, 2, \dots, n\}$ . Here  $s_i$  is the unique identifier of each state, and  $ver_i$  denotes the latest version number of each state. Upon completing the execution, nodes immediately return the  $readResult := (ctxId, output, readSet)$  to the origin chain. The *output* is the execution result. Before committing the cross-chain transaction, the origin chain can verify the two mentioned conditions. Once verification is successful, the origin chain can commit the cross-chain transaction. However, a critical problem exists in this strawman strategy. Consider the following scenario: While the origin chain is awaiting for committing the cross-chain transaction after receiving a verification message from the target chains, a new transaction on one target chain—unaware of the off-chain execution—modifies the states within the readSet and commits immediately. This results in the new transaction updating the states after the states were read by the cross-chain transaction, but committing earlier than cross-chain transaction. This violates the serializability of cross-chain transactions.

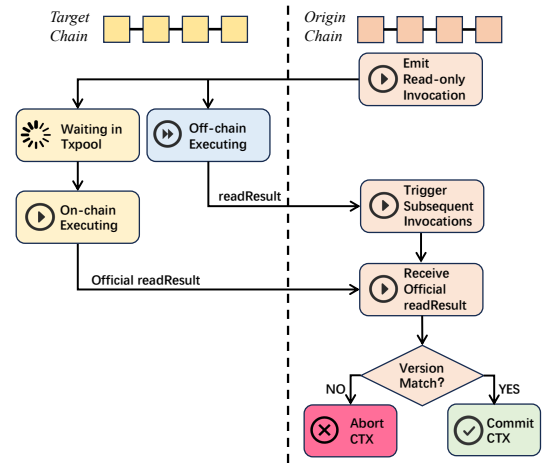


Fig. 7: Workflow of dual-processing read-write separation.

To address this issue, we implement a dual-processing read-write separation mechanism. As shown in Figure 7, when

the target chain receives a read-only invocation, nodes still execute it off-chain immediately and return *readResult* to the origin chain. However, other invocations are unaware of this off-chain executed read-only invocation. Therefore, the read-only invocation also needs to enter the transaction pool, awaiting formal execution on the target chain. Once processed, the read operations will be appended to *opLists* and given official version numbers. After that, the read-only invocation is visible to other invocations. If this official version number does not match the corresponding version number in *readResult*, it indicates that the *occState* was modified by another operation after the off-chain execution but before read-only invocation being visible, requiring the cross-chain transaction to be aborted. When version numbers match, the cross-chain transaction proceeds to commit, and the origin chain processes these *occStates* through standard workflows.

It is worth noting that the origin chain accepts an off-chain execution result from the target chain only when it receives  $2f + 1$  same return values, where  $f$  represents the maximum number of tolerated Byzantine faults in a chain. However, if insufficient matching responses arrive, it waits for the official execution result from the target chain after the read-only invocation achieves on-chain consensus finality. In this scenario, the read-only invocation reverts to a standard normal invocation, still ensuring the serializability and consistency of cross-chain transactions.

## V. SECURITY ANALYSIS

### A. Atomicity Analysis

**Lemma 1.** *ShuttleCross can guarantee atomicity of cross-chain transactions.*

*Proof.* We assume that there exists a cross-chain transaction *CTX* in ShuttleCross, including two invocations *invoke<sub>i</sub>* and *invoke<sub>j</sub>*. We assume that when *CTX* commits, *invoke<sub>i</sub>* commits but *invoke<sub>j</sub>* does not.

As described in our 2PC protocol, when *invoke<sub>i</sub>* is successfully committed, the origin chain must have committed *CTX*. Conversely, the abortion of *invoke<sub>j</sub>* indicates that the origin chain has decided to abort the *CTX* and sent abort requests to the relevant target chains.

The origin chain can make only one final decision about *CTX*, i.e., either to commit or abort *CTX*. Therefore, our assumption is contradictory. Consequently, ShuttleCross guarantees the atomicity of cross-chain transactions.  $\square$

### B. Liveness Analysis

Although malicious nodes cannot forge  $f + 1$  message signatures to deceive chains into accepting fraudulent cross-chain messages, they can still disrupt inter-chain communication by delaying or dropping messages. Besides, cross-chain communication may also fail due to network fluctuations or temporary blockchain crashes, all of which may prevent the required signatures from being collected. Such communication failures may result in four fundamental failure scenarios:

- Failed Cross-Chain Invocation: The origin chain may fail to send cross-chain invocation requests to target chains;
- Failed Execution Result Return: Target chains may fail to return the on-chain execution results to the origin chain;
- Failed On-chain State Return: Target chains may fail to return the on-chain states to the origin chain for read-only invocations;
- Failed Termination Command: The origin chain may fail to send commit or abort requests to target chains.

**Lemma 2.** *ShuttleCross can guarantee the liveness of cross-chain transactions.*

*Proof.* When the origin chain fails to send an *invoke<sub>i</sub>* to a target chain, the target chain is unable to execute the invocation and return the result back. This eventually causes the *CTX* to timeout, leading the origin chain to issue abort requests to all target chains.

Similarly, if a target chain fails to return the execution result back to the origin chain, it will also cause the *CTX* to timeout, leading the origin chain to decide to abort the *CTX* and send abort requests to all target chains. If a target chain fails to return on-chain states back to the origin chain for read-only invocations, the read-only invocations revert to normal invocations, ensuring that the cross-chain transaction does not stall.

When the origin chain fails to send commit or abort decision of *CTX* to target chains, the target chains cannot commit or abort the already executed invocations of *CTX* and must wait for inter-chain communication to recover. During this time, target chains will continuously poll the origin chain to query the cross-chain transaction status until a final commit or abort decision is obtained. Based on the queried transaction status, the target chain can uniformly commit or abort the relevant cross-chain invocations.

In addition, ShuttleCross resolves cross-chain deadlocks through its deadlock detection mechanism, ensuring that transactions do not remain indefinitely blocked. In summary, ShuttleCross eventually guarantees the liveness of cross-chain transactions even under inter-chain communication failures and deadlock risks.  $\square$

### C. Serializability Analysis

To prove that the HCC strategy guarantees the serializability of *CTXs*, we construct a conflict graph [31], where *CTXs* are represented as nodes and read-write conflicts are represented as edges. If the conflict graph does not contain any cycles, the schedule is serializable [32].

**Lemma 3.** *The *CTX* schedule generated by HCC strategy is serializable.*

*Proof.* We assume that the conflict graph generated from the schedule produced by the HCC contains a cycle  $C$ :

$$C = \{CTX_1, CTX_2, \dots, CTX_i, CTX_j, \dots, CTX_1\}$$

The directed edge  $CTX_i \rightarrow CTX_j$  in the conflict graph indicates a state contention between  $CTX_i$  and  $CTX_j$ , where

the operation of  $CTX_i$  is ahead of that of  $CTX_j$ . If both them require locking the same state with S2PL,  $CTX_i$  must commit and release its lock before  $CTX_j$  can acquire it. This ensures that  $CTX_i$  commits before  $CTX_j$ . Besides,  $CTX_i$  and  $CTX_j$  may operate the same state with OCC strategy, with  $CTX_i$ 's operation being executed first. In this scenario,  $CTX_i$  must commit before  $CTX_j$  operate the same state. Otherwise, it would violate the commitment rule of OCC.

In summary, if there exists a directed edge from  $CTX_i$  to  $CTX_j$  in the conflict graph,  $CTX_i$  always commits before  $CTX_j$ . We also can easily derive a significant corollary that if there exists a directed path in the conflict graph, the  $CTX_s$  along this path are committed in the order from the beginning to the end of the path.

Due to the existence of a directed path from  $CTX_1$  to  $CTX_j$  in the cycle  $C$ ,  $CTX_1$  must be committed before  $CTX_j$ . However, since there is also a directed path from  $CTX_j$  to  $CTX_1$ ,  $CTX_j$  must be committed before  $CTX_1$ , which creates a contradiction. Therefore, there are no cycles in the conflict graph, indicating that the schedule generated by HCC is serializable.  $\square$

## VI. EVALUATION

### A. Experimental Setup

1) *Implementation*: We have implemented ShuttleCross and baselines on ChainMaker, a mature open-source blockchain platform used across various industries. Our design requires modifying the underlying blockchain execution layer. We developed a system-level state management contract to implement the *opList* and *lockList*, managing on-chain states effectively. We also developed a system-level cross-chain management contract to further coordinate the execution of cross-chain transactions. The blockchain system actively monitors and processes timeouts at the end of each block.

2) *Experimental setup*: We conducted experiments using 32 physical machines, with each machine equipped with 16 CPU cores (Intel Xeon Platinum) and 64GB of RAM. Each physical machine hosts multiple virtual machines. In our setup, each blockchain consists of 10 nodes and use TBFT [33], a variant of PBFT, as its consensus protocol. To emulate a globally distributed network environment, we constrained the inter-node network bandwidth to 50 Mbps and introduced 50 ms latency across all communication links. The block interval for each chain is set to 1 second, and each block can contain up to 1000 transactions. Each chain receives cross-chain transactions from clients and we inject transactions to each chain at a rate of 1000 transactions per second.

3) *Metrics and baseline*: In our experiment, we evaluated throughput, latency, and abort rate. Throughput measures the number of cross-chain transactions committed per second by each chain. The latency represents the time interval from when the client submits a cross-chain transaction to the chain until the transaction is committed. The aborted transactions are excluded from the latency statistics.

To systematically evaluate the HCC mechanism's contributions, we implemented multiple baselines with ChainMaker.

We developed an optimized 2PC4BC variant, referred to as OP2PC, which incorporates the deadlock detection and lock management techniques from ShuttleCross. Both 2PC4BC and OP2PC serve as the S2PL-based baseline for cross-chain transaction coordination. Additionally, the Avalon protocol serves as the OCC-based baseline, implementing optimistic concurrency control through version validation without locking mechanism. To further assess the contributions of the read-write separation strategy, we implemented a variant of ShuttleCross called ShuttleRW. ShuttleRW employs HCC for cross-chain transaction concurrency control, but it does not adopt the read-write separation strategy. We also include GPACKT in our comparisons, in which the client locally simulates the execution of cross-chain transactions before the transactions are formally committed on-chain.

4) *Dataset*: To evaluate performance under varying state contention degree, we developed test contracts that include multiple functions for reading and writing states, with 50% of the functions being read-only. We generate 4 million CTXs designed to span a configurable number of blockchains, forming invocation trees with depths from 1 to 4. We configure parameters within the generated CTXs to control whether they read or write on-chain states. To control contention, CTXs uniformly access a finite pool of state variables. With fixed transaction number, we vary the contention by changing the pool size. A larger pool reduces the chance that concurrent CTXs touch the same state, thus lowering contention.

In addition, there is no publicly available dataset of cross-chain transactions for study. To obtain realistic cross-chain transaction data, we utilized XChainDataGen [34], a tool designed to extract cross-chain data from blockchains and generate datasets of cross-chain transactions. We extracted contracts and transactions from 8 blockchains—Ethereum [35], Avalanche [36], Binance Smart Chain [37], Arbitrum [38], Base [39], Optimism [40], Linea [41], and Polygon [42]—from October 1 to December 31, 2024. XChainDataGen integrates transactions from different chains into cross-chain transaction based on cross-chain identifiers, such as deposit, withdrawal, or message ID, resulting in a total of 4 million cross-chain transactions.

### B. Performance Without State Contention

We first conducted contention-free evaluation to isolate execution performance from state contention effects. In this setup, each cross-chain transaction reads or writes distinct state variables across all participating chains, eliminating inter-transaction contention. We conducted cross-chain experiments involving 4, 8, 16, and 32 chains, respectively. Figure 8 shows the average latency and throughput of cross-chain transactions. As the number of chains involved in each cross-chain transaction increases, the complexity of these transactions also increases, leading to a significant increase in latency and a noticeable decrease in throughput.

ShuttleCross demonstrated significant performance improvements over other schemes by employing read-write separation strategies to accelerate the execution of read-

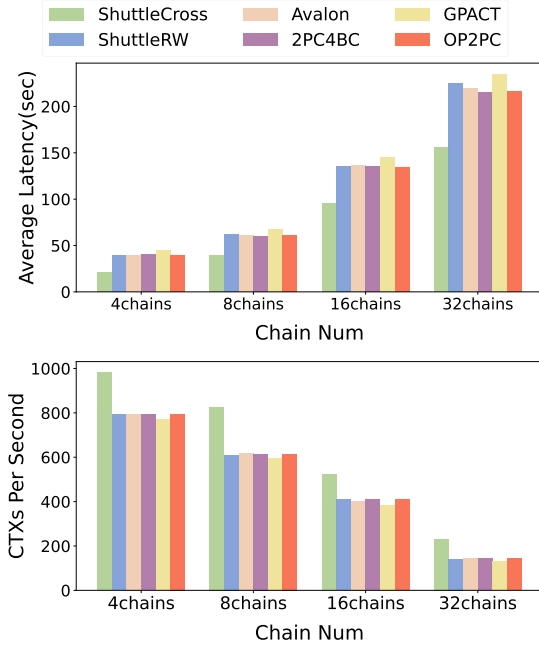


Fig. 8: Latency and throughput without state contention.

only invocations. For cross-chain transactions involving 16 chains, ShuttleCross reduced latency by 28% and increased throughput by 26% compared to ShuttleRW. However, Avalon, ShuttleRW, 2PC4BC, and OP2PC demonstrated comparable performance, as the differences between S2PL and OCC were negligible in scenarios where state contention was absent. Moreover, compared to GPACK, ShuttleCross achieved a 33% reduction in latency and a 36% increase in throughput. This is because GPACK imposes a heavier burden on clients for data fetching and execution simulation across multiple chains.

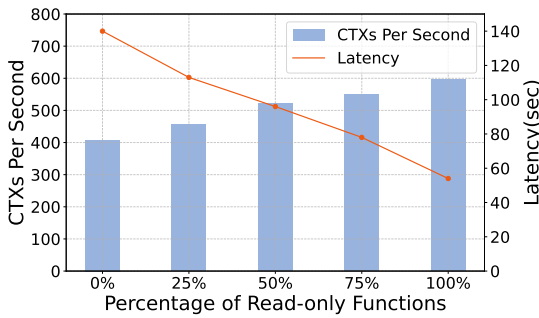


Fig. 9: Latency and throughput of ShuttleCross under different percentage of read-only functions.

We also studied the impact of varying proportions of read-only functions in our test contract on overall performance. We measured the latency and throughput of ShuttleCross involving 16 chains without state contention. In Figure 9, as the proportion of read-only functions increases, we observe a significant reduction in latency and a notable improvement in throughput.

These results highlight the efficiency of ShuttleCross’s read-write separation strategy.

### C. Performance under State Contention

We also evaluated each cross-chain mechanism using test contracts while simulating different levels of state contention by adjusting the number of accessible on-chain states. As the number of states available increases, the degree of state contention between cross-chain transactions decreases. Conversely, as the number of accessible states decreases, the level of state contention between cross-chain transactions increases.

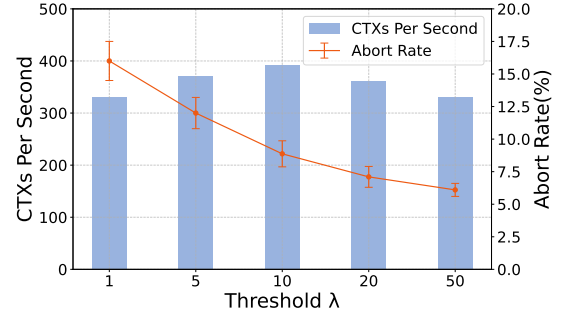


Fig. 10: Throughput and abort rate of ShuttleCross under different threshold  $\lambda$ .

Since the *lockState* will be converted into *occState* when it does not acquire a write lock within  $\lambda$  blocks, we investigated the effect of varying the threshold  $\lambda$  on the performance of ShuttleCross. We conducted the experiment across 16 chains, with 3,000 available states on each chain. As shown in Figure 10, when  $\lambda$  is 10, ShuttleCross achieves its highest throughput under current experimental conditions. If  $\lambda$  is too low, most states are categorized as *occStates*, leading to a high abort rate, resulting in throughput degradation. If  $\lambda$  is too high, most *lockStates* cannot transition to *occStates* for a long time, resulting in decreased concurrency performance.

We also conducted experiments across 16 chains, with  $\lambda$  set to 10, to calculate the abort rate of cross-chain transactions, as illustrated in Figure 11. GPACK experiences the highest abort rate because it simulates execution on the client side before committing the transaction on-chain. This approach results in a high abort rate, as the off-chain simulated outcome may not align with the on-chain execution result. Avalon also demonstrated a high abort rate due to its OCC strategy. Despite using

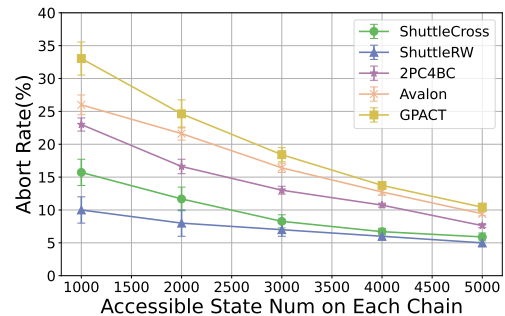


Fig. 11: Abort rate under varying state contention.

the S2PL strategy, 2PC4BC directly aborts the transaction if it fails to acquire state locks, resulting in higher abort rate than ShuttleCross and ShuttleRW. ShuttleCross effectively reduced the transaction abort rate by implementing HCC to leverage the advantages of S2PL. Additionally, ShuttleCross employs a lockList, which prevents the directly aborting the cross-chain transactions if state locks are not acquired. ShuttleRW achieved an even lower abort rate compared to ShuttleCross, as it avoids the abort risks associated with read-write separation.

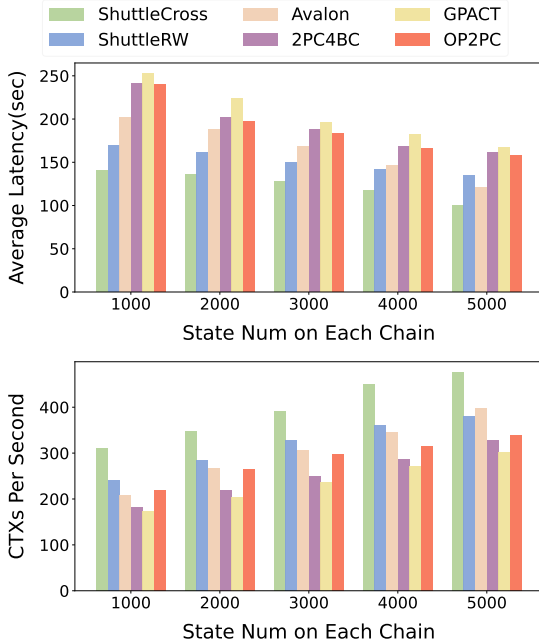


Fig. 12: Latency and throughput under varying state contention.

Figure 12 illustrates the latency and throughput of cross-chain transactions involving 16 chains under varying degrees of state contention. As the number of accessible states on each blockchain increases, state contention decreases, resulting in reduced latency and improved throughput for each cross-chain mechanism. The results demonstrate that ShuttleCross outperforms other cross-chain mechanisms. Due to higher abort rates, Avalon, 2PC4BC and GPACK exhibit lower performance under high state contention. Compared to 2PC4BC, ShuttleCross achieves up to a 36% reduction in latency and a 46% increase in throughput. Although ShuttleRW has a lower abort rate than ShuttleCross, the read-write separation strategy in ShuttleCross accelerates its performance, leading to better performance. Additionally, ShuttleRW outperforms OP2PC with an 18.7% latency reduction and up to a 14.6% TPS improvement, confirming the contributions of HCC.

Besides, we analyze the breakdown of the end-to-end latency for the transactions, as shown in Figure 13. Due to multiple cross-chain invocations, cross-chain transactions spend the majority latency queuing in the TxPool. ShuttleCross leverages read-write separation to significantly reduce this

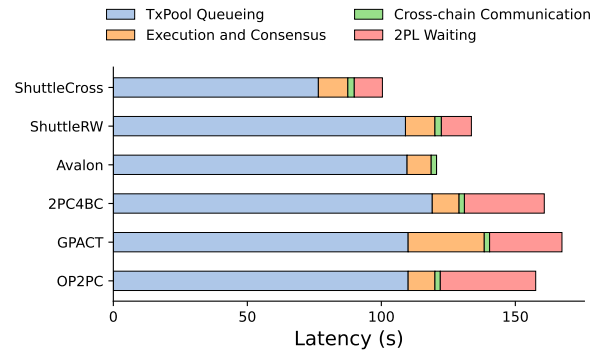


Fig. 13: Latency breakdown across 16 chains with 5000 states.

queuing time. Additionally, ShuttleCross utilizes HCC to decrease the lock-waiting latency.

State Num Per Chain	Graph Size(KB)	Detection Time( $\mu$ s)
1000	44.69	83.728
2000	27.93	57.014
3000	22.01	50.383
4000	18.59	40.806
5000	16.48	29.627

TABLE I: Deadlock detection overhead under varying state contention.

In addition, we also explored the overhead of deadlock detection under varying state contention. We measured the size of the wait-for graph exchanged between chains and the time cost for performing deadlock detection. As shown in Table I, with higher transaction conflict levels, the size of the wait-for graph increases, leading to higher execution time for deadlock detection. However, in all experiments, the cost of performing deadlock detection and exchanging wait-for graphs was minimal, meaning that deadlock detection does not significantly impact performance.

#### D. Performance on Realistic Cross-chain Data

As discussed in Section VI-A4, we utilized XChainDataGen to obtain realistic cross-chain transaction data from eight different blockchains. We used this realistic data to conduct cross-chain experiments across 2, 4, 6, and 8 chains, with the results presented in Figure 14. Compared to ShuttleRW, ShuttleCross achieved up to a 24% reduction in latency and a 33% increase in throughput. In comparison to Avalon, ShuttleCross demonstrated up to a 36% reduction in latency and a 49% increase in throughput. ShuttleCross reduced latency by up to 55% and increased throughput by up to 52% compared to 2PC4BC. Against OP2PC, ShuttleCross lowered latency by 45% and improved throughput by 34%. Furthermore, ShuttleCross achieved up to a 58% reduction in latency and a 55% increase in throughput compared to GPACK.

#### E. Robustness Evaluation

To evaluate the robustness of ShuttleCross, we conducted experiments under both network fluctuations and malicious nodes attacks. We simulated varying degrees of network

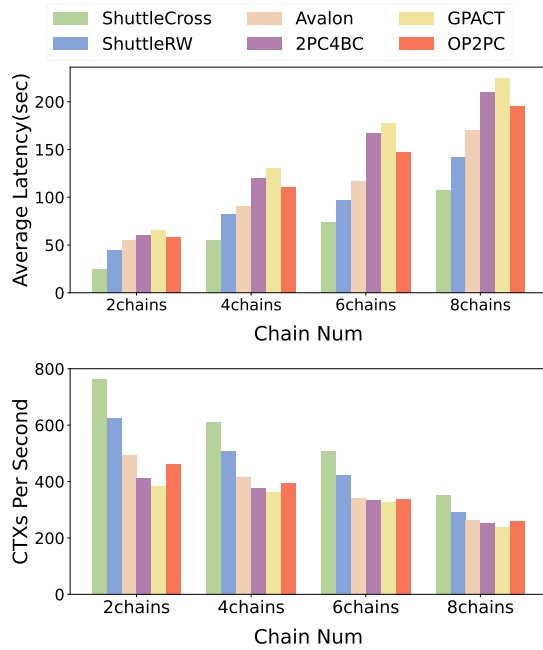


Fig. 14: Latency and throughput on realistic cross-chain data.

fluctuations by adjusting the proportion of dropped cross-chain messages with traffic control tools [43]. The experiments were performed across 16 blockchains running test contracts, each chain consisting of 10 nodes with up to  $f = 3$  malicious nodes. When malicious nodes intentionally delayed or dropped cross-chain messages, combined with network disruption, trigger a timeout and cause the CTX to abort.

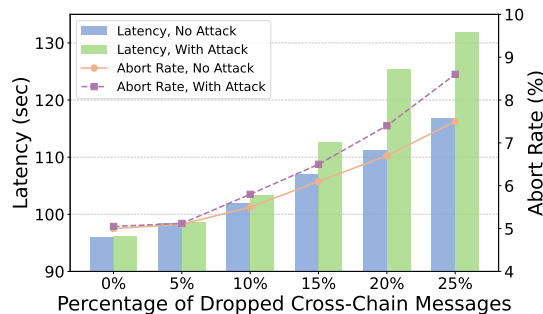


Fig. 15: Latency and abort rate of ShuttleCross under different network disruption intensities.

Across all experimental conditions, ShuttleCross consistently maintained the atomicity and liveness of CTXs. Figure 15 presents that as the percentage of dropped cross-chain messages increases, the latency and abort rate of CTXs also rises accordingly, which aligns with our theoretical expectations. Moreover, when malicious nodes attack by delaying or dropping cross-chain messages, the latency and abort rate of CTXs become even more pronounced. Thus, while network fluctuations and malicious attacks do affect ShuttleCross’s performance, we are still able to ensure the security of cross-chain transactions.

## F. Overhead Analysis

As shown in Figure 13, ShuttleCross exhibits a 7% increase in average execution time due to computational overhead. Our experiments demonstrate that HCC introduces no significant memory overhead, maintaining the client memory footprint around 400 MB. Furthermore, off-chain execution of read-write separation runs concurrently with on-chain execution. Under evaluations using realistic dataset, the off-chain execution limits the overall memory consumption increase to 26%.

As the number of involved blockchains increases, the latency of cross-chain transactions naturally rises due to extended queuing in the TxPool, but this hardly affects memory or computational overhead. Additionally, since transactions within a block are executed sequentially, increasing the transaction volume does not significantly change the average memory footprint or computational overhead.

## VII. DISCUSSION

ShuttleCross targets permissioned chains, leveraging direct signing for inter-chain message verification via P2P connection. Some system employ relay nodes to forward cross-chain messages. ShuttleCross seamlessly supports both P2P and relay-nodes for message routing.

Besides, Cosmos and Polkadot use relay chain to verify and route messages, avoiding pairwise links that scale poorly with many chains. Relay chain consensus on cross-chain transactions ensures all nodes maintain the same transaction pool. Relay chain is compatible with our 2PC and HCC protocols, which only require verifiable messages. However, supporting read-write separation requires minor modifications on relay chain. As read-only invocation skips consensus, when the relay chain receives a signed read-only result, it should bypass its own consensus, perform signature checks, and directly forward the result back to origin chain.

## VIII. CONCLUSION

In this paper, we present ShuttleCross, an efficient cross-chain smart contract invocation framework. We design a HCC strategy that not only guarantees serializability of CTXs but also enhances their concurrency and reduces the abortion rate. Furthermore, we design a read-write separation strategy that effectively reduces the latency of CTXs. Evaluation results demonstrate that ShuttleCross exhibits enhanced performance in handling complex cross-chain scenarios, offering a robust and efficient solution for enabling interoperability in different blockchains. Additionally, Future research could apply reinforcement learning to classify *occState* and *lockState*, further optimizing CTX performance.

## IX. ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their comments. This work was supported by the National Key R&D Program of China under Grant 2023YFB2703800. The contact author is Zhen Xiao.

## REFERENCES

- [1] P. Treleaven, R. G. Brown, and D. Yang, "Blockchain technology in finance," *Computer*, vol. 50, no. 9, pp. 14–17, 2017.
- [2] S. Tuli, S. Tuli, G. Wander, P. Wander, S. S. Gill, S. Dustdar, R. Sakellariou, and O. Rana, "Next generation technologies for smart healthcare: challenges, vision, model, trends and future directions," *Internet technology letters*, vol. 3, no. 2, p. e145, 2020.
- [3] M. Samaniego, U. Jamsrandorj, and R. Deters, "Blockchain as a service for iot," in *2016 IEEE international conference on internet of things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartiData)*. IEEE, 2016, pp. 433–436.
- [4] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *Acm Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–41, 2021.
- [5] H. Yuan, S. Fei, and Z. Yan, "Technologies of blockchain interoperability: a survey," *Digital Communications and Networks*, vol. 11, no. 1, pp. 210–224, 2025.
- [6] J. Kwon and E. Buchman, "Cosmos whitepaper," *A Netw. Distrib. Ledgers*, vol. 27, pp. 1–32, 2019.
- [7] K. Ren, N.-M. Ho, D. Loghini, T.-T. Nguyen, B. C. Ooi, Q.-T. Ta, and F. Zhu, "Interoperability in blockchain: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12750–12769, 2023.
- [8] B. Pillai, K. Biswas, Z. Hóu, and V. Muthukumarasamy, "Cross-blockchain technology: integration framework and security assumptions," *IEEE access*, vol. 10, pp. 41239–41259, 2022.
- [9] L. Yin, J. Xu, and Q. Tang, "Sidechains with fast cross-chain transfers," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3925–3940, 2021.
- [10] Z. Yin, B. Zhang, J. Xu, K. Lu, and K. Ren, "Bool network: An open, distributed, secure cross-chain notary platform," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3465–3478, 2022.
- [11] V. Buterin, "Chain interoperability," *R3 research paper*, vol. 9, pp. 1–25, 2016.
- [12] H. Tian, K. Xue, X. Luo, S. Li, J. Xu, J. Liu, J. Zhao, and D. S. L. Wei, "Enabling cross-chain transactions: A decentralized cryptocurrency exchange protocol," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3928–3941, 2021.
- [13] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu, "Hyperservice: Interoperability and programmability across heterogeneous blockchains," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 549–566.
- [14] G. Falazi, U. Breitenbücher, F. Leymann, and S. Schulte, "Cross-chain smart contract invocations: a systematic multi-vocal literature review," *ACM Computing Surveys*, vol. 56, no. 6, pp. 1–38, 2024.
- [15] G. Falazi, U. Breitenbücher, F. Leymann, S. Schulte, and V. Yusupov, "Transactional cross-chain smart contract invocations," *Distributed Ledger Technologies: Research and Practice*, 2023.
- [16] P. Robinson and R. Ramesh, "General purpose atomic crosschain transactions," in *2021 3rd Conference on blockchain research & applications for innovative networks and services (BRAINS)*. IEEE, 2021, pp. 61–68.
- [17] M. Westerkamp and A. Küpper, "Smartsync: Cross-blockchain smart contract interaction and synchronization," in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2022, pp. 1–9.
- [18] Y. Cai, R. Cheng, Y. Zhou, S. Zhang, J. Xiao, and H. Jin, "Enabling complete atomicity for cross-chain applications through layered state commitments," in *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2024, pp. 248–259.
- [19] ChainMaker, "Chainmaker-go project," <https://git.chainmaker.org.cn/chainmaker/chainmaker-go>, 2024, accessed: 2024-10-05.
- [20] Y. Lan, J. Gao, Y. Li, K. Wang, Y. Zhu, and Z. Chen, "Trustcross: Enabling confidential interoperability across blockchains using trusted hardware," in *Proceedings of the 2021 4th International Conference on Blockchain Technology and Applications*, 2021, pp. 17–23.
- [21] J. Burdges, A. Cevallos, P. Czaban, R. Habermeier, S. Hosseini, F. Lama, H. K. Alper, X. Luo, F. Shirazi, A. Stewart *et al.*, "Overview of polkadot and its design considerations," *arXiv preprint arXiv:2005.13456*, 2020.
- [22] Digital Asset Canton Team, "Canton: A daml based ledger interoperability protocol," Digital Asset, Technical Report, 2020, available online: <https://www.digitalasset.com>.
- [23] W. Wang, Z. Zhang, G. Wang, and Y. Yuan, "Efficient cross-chain transaction processing on blockchains," *Applied Sciences*, vol. 12, no. 9, p. 4434, 2022.
- [24] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.
- [25] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2003, pp. 235–248.
- [26] M. Nissl, E. Sallinger, S. Schulte, and M. Borkowski, "Towards cross-blockchain smart contracts," in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2021, pp. 85–94.
- [27] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.
- [28] W. Fu, J. Du, Y. Zhang, and Z. Wang, "A blockchain cross-chain solution based on relays," *International Journal of Knowledge and Innovation Studies*, vol. 2, no. 2, pp. 70–80, 2024.
- [29] R. Gelashvili, A. Spiegelman, Z. Xiang, G. Danezis, Z. Li, D. Malkhi, Y. Xia, and R. Zhou, "Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 232–244.
- [30] J. Xiao, S. Zhang, Z. Zhang, B. Li, X. Dai, and H. Jin, "Nezha: Exploiting concurrency for transaction processing in dag-based blockchains," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 269–279.
- [31] P. A. Bernstein, D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Transactions on Software Engineering*, no. 3, pp. 203–216, 1979.
- [32] G. Weikum and G. Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.
- [33] J. Zhang, J. Gao, K. Wang, Z. Wu, Y. Li, Z. Guan, and Z. Chen, "Tbft: efficient byzantine fault tolerance using trusted execution environment," in *ICC 2022-IEEE International Conference on Communications*. IEEE, 2022, pp. 1004–1009.
- [34] A. Augusto, A. Vasconcelos, M. Correia, and L. Zhang, "Xchain-datagen: A cross-chain dataset generation framework," *arXiv preprint arXiv:2503.13637*, 2025.
- [35] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, no. 22-23, pp. 5–7, 2013.
- [36] A. Labs, "Avalanche docs," Ava Labs, Tech. Rep., 2020, accessed: 2025-05-08. [Online]. Available: <https://build.avax.network/docs>
- [37] Binance, "Bnb chain documentation," Binance, Tech. Rep., 2024, accessed: 2025-05-08. [Online]. Available: <https://docs.bnbchain.org/bnb-smart-chain/introduction/>
- [38] O. Labs, "A gentle introduction to arbitrum," Arbitrum Foundation, Tech. Rep., 2025, accessed: 2025-05-08. [Online]. Available: <https://docs.arbitrum.io/welcome/arbitrum-gentle-introduction>
- [39] Base, "Base documentation," Base, Tech. Rep., 2024, accessed: 2025-05-08. [Online]. Available: <https://docs.base.org/>
- [40] O. Foundation, "Optimism docs," Optimism Foundation, Tech. Rep., 2025, accessed: 2025-05-08. [Online]. Available: <https://docs.optimism.io/>
- [41] C. Inc., "Linea docs," Consensys Inc., Tech. Rep., 2025, accessed: 2025-05-08. [Online]. Available: <https://docs.linea.build/get-started/>
- [42] P. Labs, "Polygon knowledge layer," Polygon Labs, Tech. Rep., 2025, accessed: 2025-05-08. [Online]. Available: <https://docs.polygon.technology/>
- [43] M. Kerrisk, *Linux Traffic Control Utility*, accessed: 2025-01-18. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc.8.html>