POLITECNICO DI TORINO
Repository ISTITUZIONALE

Parallel Transaction Execution in Blockchain and the Ambiguous State Representation Problem

Original
Parallel Transaction Execution in Blockchain and the Ambiguous State Representation Problem / Capocasale, V., Pedone, F., Perboli, G.. - (2024), pp. 131-138. (19th European Dependable Computing Conference (EDCC) Leuven (BEL) 08-11 April 2024) [10.1109/edcc61798.2024.00035].

Availability:
This version is available at: 11583/2989790 since: 2024-06-22T14:29:23Z

Publisher:
IEEE

Published
DOI:10.1109/edcc61798.2024.00035

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

28 November 2026

Parallel Transaction Execution in Blockchain and the Ambiguous State Representation Problem

Vittorio Capocasale
Dept. of control and computer engr.
Polytechnic University of Turin
Turin, Italy
vittorio.capocasale@polito.it

Fernando Pedone
Computer Systems Institute
Università della Svizzera Italiana
Lugano, Switzerland
fernando.pedone@usi.ch

Guido Perboli
Dept. of management and production engr.
Polytechnic University of Turin
Turin, Italy
guido.perboli@polito.it

***Abstract*—Blockchains commonly employ tree data structures (e.g., Merkle trees) to represent state. While tree structures enable fast and compact state correctness checks, they introduce constraints when it comes to parallelizing transaction execution. In particular, concurrent transaction execution can lead to multiple trees representing the same state, hindering consensus among blockchain peers. We characterize this phenomenon as the *ambiguous state representation problem* and propose an optimistic algorithm that guarantees the creation of the same state tree across multiple peers. We integrated our solution into Cosmos SDK framework, a popular production blockchain system, allowing applications to benefit from parallel transaction execution without modifying their existing codebase. We report on the performance of parallel transaction execution under a variety of conditions in a network of up to 40 peers.**

***Index Terms*—blockchain, deterministic execution, parallel computing, Cosmos, ambiguous state representation problem**

I. Introduction

In recent years, blockchain has emerged as a promising technology due to its wide range of potential disruptive applications in various industries. In a blockchain network, all peers execute identical operations in the same order, following the state machine replication model [1], [2]. Through blockchain, separate parties can collaboratively manage a shared and distributed database, enhancing resilience, auditability, and security of business processes. As a result, improving the transaction processing efficiency of blockchain becomes imperative for widespread adoption by institutions and enterprises.

But blockchain operates differently than traditional databases, which achieve high scalability by distributing workloads across different nodes. Many blockchains currently process transactions sequentially to prevent cross-peer state inconsistencies [3]. If on the one hand, sequential execution ensures state consistency, on the other hand, it severely limits performance [4]. Figure 1 demonstrates the results of processing 10^5 payment transactions on a single-node Cosmos testnet. As explained in more detail later in the paper, DeliverTx, which includes transaction execution, accounts for approximately 75% of the processing time of a block of transactions at the application layer. This observation motivates this work: introducing parallel transaction execution in blockchain.

Although parallel execution has been extensively studied in the context of state machine replication (e.g., [5], [6]),

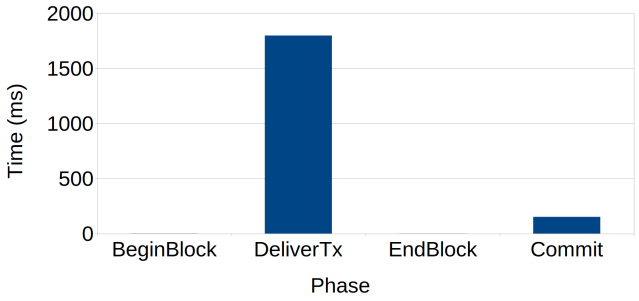
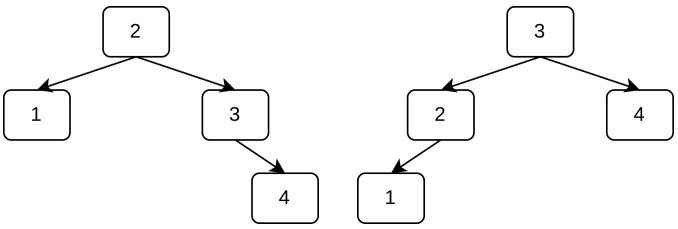


Fig. 1: Time breakdown of Gaia, a Cosmos application, in the various phases involved in processing transaction blocks.

traditional parallel execution strategies are not directly applicable to blockchain systems. Blockchains often employ hierarchical data structures such as Merkle trees to represent their state, as opposed to flat key-value stores [7]. Merkle trees provide efficient verification of data integrity even when retrieved from potentially untrusted peers [8]. Unfortunately, the order-dependent nature of some Merkle trees calls for stricter requirements for concurrency [9]. In parallel state machine replication approaches, two transactions can execute concurrently if they do not conflict, that is, they either access different keys or they only read common keys [5]. Some Merkle trees, however, introduce indirect dependencies, not visible at the level of transactions. These dependencies can lead to variations in state representation and inconsistencies in blockchain systems. Consider, for example, executions of four transactions that create keys 1..4, as depicted in Figure 2. Even though these transactions access different keys and can execute concurrently under parallel state machine replication, different transaction schedules may lead to distinct trees. Specifically, executing the four transactions in the order $(1, 2, 3, 4)$ generates the left-side tree, while executing them in the order $(4, 3, 2, 1)$ generates the right-side tree. Both trees hold the same values in the same order, but they are distinct, resulting in different Merkle root hashes. We characterize this challenge as the "ambiguous state representation problem".

Besides characterizing the ambiguous state representation problem, this paper introduces an optimistic algorithm for parallel transaction execution in blockchain systems that



(a) Insert order: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ (b) Insert order: $4 \rightarrow 3 \rightarrow 2 \rightarrow 1$

Fig. 2: Different insertion orders of the same set of values can result in different state representations in (Merkle) AVL tree.

circumvents the generation of ambiguous state representations. Our approach is universally applicable, accommodating blockchains utilizing any Merkle tree structure. To the best of our knowledge, this study is the first to introduce parallelism into blockchains using such generalized data structures. The primary contributions of this paper are as follows:

- We identify and formalize the ambiguous state representation problem in the context of blockchain parallel transaction execution.
- We present a general approach to parallel transaction execution in blockchain frameworks, irrespective of their state representation.
- We have seamlessly integrated our approach into the widely adopted Cosmos SDK framework, enabling existing applications to harness the benefits of parallel transaction execution by updating a single package dependency.
- We conducted extensive performance testing under various network configurations, offering a comprehensive analysis of the performance parallel transaction execution. We employed standard tests provided by the popular YCSB benchmark.

The remainder of this paper is structured as follows: Section II presents essential background information, introduces the ambiguous state representation problem, and overviews the Cosmos blockchain. Section III summarizes the state-of-the-art. Section IV outlines our approach to optimistic and parallel transaction execution in blockchain frameworks. Section V delves into the integration of our approach into the Cosmos SDK and presents our results under varying network configurations. Section VI concludes the paper.

II. Background

This section provides a summary of the concepts utilized in this study, introduces the problem we aim to address, and briefly presents the Cosmos blockchain, the environment we used to experimentally assess our contributions.

A. Blockchain

Informally, a blockchain is a distributed and decentralized database managed by a network of peers. Each peer controls one copy of the database. Clients may attempt to alter the database by submitting transactions. Peers agree on which transactions to execute through a consensus algorithm to keep their copies in sync. Since executing this process for each transaction would be inefficient, transactions are grouped into blocks. Each block contains the hash of the previous one to prevent modifications to the block sequence [10], [11].

However, relying solely on the block sequence is impractical as it would require a full scan through the blocks to retrieve updated information. To address this, blockchains utilize key-value stores to hold the updated state of the database at a specific block height [12]. This approach enables efficient data retrieval but lacks the ability to verify the integrity of the retrieved data, specifically the coherence between the state database and the block sequence. To ensure verifiability, blockchains employ authenticated data structures instead of flat key-value stores [13]. The most prominent example of such structures is Merkle trees, where data is stored in the leaves, and each node contains the hash of its children. Due to the properties of hash functions, any modification to the data, order, or structure of the tree would result in a root hash mismatch. Additionally, finding two different trees with the same root hash is highly improbable. Thus, by storing the root hash of the tree in a block, data integrity can be easily verified.

B. Problem statement

We define a blockchain $B_{[h]}$ with last committed block h for a set of peers $\mathcal{P} = \{0, 1, \ldots, (n-1)\}$ as:

$$B_{[h]} := \{\sigma_{[h],0}, \sigma_{[h],1}, \ldots, \sigma_{[h],n-1}\}, \tag{1}$$

where $\sigma_{[h],i}$ belongs to set S of all the Merkle trees with values in the leaves representing a key-value store with keys in K. $MRoot : S \rightarrow \{0,1\}^{\kappa}$ is the function returning the Merkle root of a Merkele tree with security parameter κ.

We define $f : S \times \Gamma \rightarrow S$ as a state transition function that produces a new state tree $\sigma_{[h+1]} \in S$ by taking as input the current state $\sigma_{[h]} \in S$ and a schedule (i.e., a total order) Γ of transactions $T_{[h+1]} = \{t_0, t_1, \ldots, t_{m-1}\}$ committed in block $h+1$. Each transaction consists of one or more read, insert, or delete operations on the state tree. Each operation is atomic and targets a single key/leaf in the state tree. Only insert and delete operations may modify the tree.

Function $Keys : T \rightarrow K$ returns the set of keys targeted by a transaction. In Section IV, we will also use functions $WriteSet : T \rightarrow K$ and $ReadSet : T \rightarrow K$, which return the mutative and read operations of a transaction, respectively. Notice that the keys in K identify only the leaves of a tree. Thus, in this study, transactions have the same read/write sets if they operate on the same leaves, independently of inner nodes.

Concurrent transaction execution may result in different peers executing different schedules. For this reason, we introduce the following definitions.

Definition II.1. Let tree $\sigma \in S$. Two schedules Γ_i and Γ_j of T are σ-equivalent, with notation $\Gamma_i =_{\sigma} \Gamma_j$, iff:

$$MRoot(f(\sigma, \Gamma_i)) = MRoot(f(\sigma, \Gamma_j)). \tag{2}$$

Definition II.2. $B_{[h]}$ is in a consistent state $\sigma_{[h]}$ iff:

$$\forall i, j \in \mathcal{P} : MRoot(\sigma_{[h],i}) = MRoot(\sigma_{[h],j}) \tag{3}$$

Theorem II.1. Let $B_{[h]}$ be in a consistent state $\sigma_{[h]}$. $B_{[h+1]}$ is in consistent state $\sigma_{[h+1]}$ iff $\forall i, j \in \mathcal{P} : \Gamma_i =_{\sigma_{[h]}} \Gamma_j$.

Proof. By the state transition function definition, $\forall i \in \mathcal{P} : \sigma_{[h+1],i} = f(\sigma_{[h],i}, \Gamma_i)$. If we apply the $MRoot$ function to both sides of the equation, we obtain: $\forall i \in \mathcal{P} : MRoot(\sigma_{[h+1],i}) = MRoot(f(\sigma_{[h],i}, \Gamma_i))$. However, $\forall i \in \mathcal{P} : \sigma_{[h],i} = \sigma_{[h]}$, as $\sigma_{[h]}$ is consistent and finding hash collisions is unlikely. Thus:

$$\forall i \in \mathcal{P} : MRoot(\sigma_{[h+1],i}) = MRoot(f(\sigma_{[h]}, \Gamma_i)). \tag{4}$$

Sufficiency. If $\forall i, j \in \mathcal{P} : \Gamma_i =_{\sigma_{[h]}} \Gamma_j$, then, by definition, $\forall i, j \in \mathcal{P} : MRoot(f(\sigma_{[h]}, \Gamma_i)) = MRoot(f(\sigma_{[h]}, \Gamma_j))$. We can now use Eq. 4 on both sides to obtain that $\forall i, j \in \mathcal{P} : MRoot(\sigma_{[h+1],i}) = MRoot(\sigma_{[h+1],j})$. Thus, $\sigma_{[h+1]}$ is consistent.

Necessity. If $\exists i, j \in \mathcal{P} : \Gamma_i \neq_{\sigma_{[h]}} \Gamma_j$, then, by definition, $\exists i, j \in \mathcal{P} : MRoot(f(\sigma_{[h]}, \Gamma_i)) \neq MRoot(f(\sigma_{[h]}, \Gamma_j))$. We can now use Eq. 4 on both sides to obtain that $\exists i, j \in \mathcal{P} : MRoot(\sigma_{[h+1],i}) \neq MRoot(\sigma_{[h+1],j})$. Thus, $\sigma_{[h+1]}$ is not consistent.

□

In flat key-value stores, transactions targeting different keys may be executed in any order and still produce the same state in the end. Merkle prefix trees behave similarly, since different orders of the same non-conflicting operations produce the same prefix tree in the end. This result, however, is not extensible to other tree structures as well, as evinced in Fig. 2. To better capture this idea, we formally introduce the ambiguous state representation problem.

Definition II.3. A tree σ is affected by the ambiguous state representation problem iff:

$$\text{there are schedules } \Gamma_i, \Gamma_j \text{ of } T \text{ such that} \tag{5}$$

$$\forall t_x, t_y \in T, t_x \neq t_y : Keys(t_x) \cap Keys(t_y) = \emptyset \text{ and} \tag{5a}$$

$$\Gamma_i \neq_\sigma \Gamma_j \tag{5b}$$

Informally, the problem states that, depending on the underlying tree structure, transactions may have read/write dependencies even if they target unrelated objects. For example, inserting a node in a tree may cause rebalancing operations affecting multiple nodes, not only the inserted one. Additionally, which nodes are affected cannot be predicted by analyzing the transactions only, but also depends on the contingent state of the tree and how the transactions are scheduled.

In our tests, we utilized the Cosmos SDK, a framework that facilitates the construction of cryptocurrency-oriented blockchains. However, in such chains, it is customary for transaction fees to be paid to the block proposer and, consequently, to the same blockchain address. This practice can introduce write conflicts among all transactions within a block. Therefore, in addition to addressing the ambiguous state representation problem, we also propose a workaround for this issue in our algorithm.

C. Cosmos

Cosmos [14] is a network of interconnected blockchains that leverage the Inter-Blockchain Communication protocol to facilitate cross-chain data transfer. The Cosmos ecosystem offers various tools to streamline the creation of new blockchains and expand the network. The typical software stack of a Cosmos blockchain is illustrated in Fig. 3. At the core of the architecture is Tendermint Core (now rebranded as Comet BFT), responsible for managing state machine replication. Tendermint Core interacts with external applications implementing the state machine logic through the ABCI interface. The following methods are invoked in order of appearance:

- BeginBlock: signals the start of a new block to the application.
- DeliverTx: delivers a single transaction to the application, with this method being invoked once per transaction.
- EndBlock: indicates the end of the block.
- Commit: instructs the application to produce the Merkle Root and return it to the Tendermint Core.

In the ABCI 2.0 specification, there is a change to the transaction processing methods. Instead of separate methods like BeginBlock, DeliverTx, and EndBlock, a single method, called FinalizeBlock, is introduced. This new method delivers all the transactions to the application in a single operation.

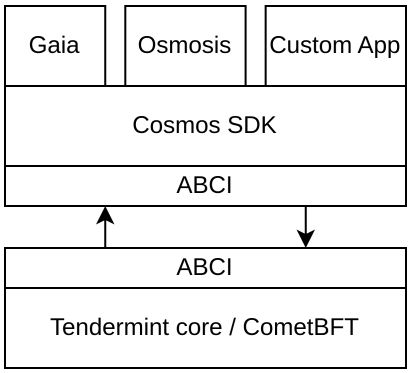


Fig. 3: Software stack of a Cosmos SDK application.

Given the shared characteristics among blockchains, developers can utilize the Cosmos SDK to further expedite the process of creating blockchains. The Cosmos SDK supports the development of multi-asset public Proof-of-Stake (PoS) blockchains, as well as permissioned Proof-of-Authority (PoA) blockchains. Functionality such as account management, gas measurement, fee payment, and staking is provided out-of-the-box to developers by the Cosmos SDK.

In this study, we showcase our approach enhancing the default transaction execution efficiency of the Cosmos SDK.

III. Related work

Parallel execution in state machine replication has been largely explored in the literature (e.g., [6], [15]–[21]). Most

proposals are based on an early observation that non-conflicting commands can be executed concurrently [1]. The strategy is justified by the fact that many workloads are dominated by non-conflicting commands (e.g., [21]–[23]). As we have argued, the order-dependent nature of some Merkle trees requires for stricter concurrency requirements.

Given the importance of transaction throughput, permissioned blockchain frameworks have considered various techniques to explore parallel execution. An example is Hyperledger Fabric [24], which departs from the traditional order-execute approach and adopts the execute-order-validate model, where transactions are optimistically executed. In the validation phase, multiversion concurrency control is employed to ensure the aborting of conflicting transactions. Performance heavily depends on the number of conflicting transactions in a block [7]. Our proposal, which also relies on optimistic transaction execution, suffers from the same limitation. Possible ways of mitigating this issue include re-ordering transactions [25] and re-executing conflicting transactions [26], which is the strategy we decided to adopt. In particular, the write set of an aborted transaction can be used to efficiently detect dependencies that can be exploited during re-executions. Block-STM [27], used by Aptos, leverages this idea.

Hyperledger Sawtooth [28] adopts a distinct approach by providing a parallel scheduler that takes into account transaction dependencies. This scheduler enables the execution of non-conflicting transactions in parallel, leading to potential performance improvements in blockchain systems [29]. The effectiveness of this approach has also been explored in other studies, where it has been applied to different blockchain frameworks [30], [31]. However, it is important to note that the parallel scheduler in Hyperledger Sawtooth requires clients to declare the read/write sets of each transaction. This task can pose challenges, as clients may need to pre-execute the transaction locally while other clients concurrently modify the state of the blockchain, which can invalidate the pre-execution [31]. Additionally, application developers must carefully allocate the key space of the state database to minimize the likelihood of transaction conflicts. We avoid these problems by dynamically discovering the read/write sets of transactions instead of burdening clients with declaring them.

Other approaches have explored static analysis [32] or semantic analysis [33] of transactions to partition them into shards for concurrent execution. Such approaches are well-suited when the state database is represented by a prefix tree structure, but they may not be easily generalized to other tree structures due to the challenges posed by the ambiguous state representation problem.

PEEP [9] introduces a mechanism for parallel execution of transactions after a sequential locking phase, where transactions acquire the required locks. The authors highlight the requirement of using tree structures that remain unaffected by the order of insertions (i.e., prefix tree). In contrast, our approach generalizes to any tree structure.

Neuchain [34] executes transactions in parallel without relying on an explicit ordering phase. Incremental identifiers associated to transactions are used to deterministically resolve conflicts, but this solution may introduce centralization and scalability issues, which are avoided by our proposal.

We note that numerous blockchains, including Ethereum [35], continue to process transactions in a sequential manner. Consequently, while parallel execution can potentially be enhanced through hardware and software co-design [36], we anticipate that the industry will adopt an incremental approach to minimize disruptive changes. Our approach goes in this direction by not requiring modifications to existing applications using the Cosmos SDK. We execute transactions optimistically and in parallel to discover their read/write sets to then re-execute only conflicting ones. Such a strategy is not new in the literature [37], but our algorithm also addresses the ambiguous state representation problem. To the best of our knowledge, no other approach proposed in the literature addresses such a problem.

IV. Algorithm Description

Our definition for the ambiguous state representation problem highlights that out-of-order write-operation execution may result in different tree representations even if such write operations target different tree leaves. Nonetheless, such tree representations only differ by one or more rotation operations, which means that they associate the same values to the same keys. Thus, such representations cannot be used for computing the Merkle root hash, but are still useful for reading/writing values. More generally, peers may exploit flat key-value stores for reading/writing values provided that they have a way to compute the same Merkle root hash in the end.

The pseudocode of our solution is provided in Algorithm 1. In short, transactions are executed independently from one another, and each transaction writes to its own copy of the state database, represented by a flat key-value store. Then, sequentially, the writes of each transaction are persisted to the state tree if no conflicts arise. Otherwise, the transaction is re-executed. The algorithm leverages the following functions.

- *Iterator (txs)*: creates an iterator over the provided array of objects.
- *Next (i)*: moves iterator *i* to the next value.
- *HasNext (i)*: checks if the iterator reached the end of the array.
- *GetValue (i)*: returns the value currently pointed by the iterator.
- *AnteHandleSync (tx, store)*: synchronously executes the ante handler for transacton *tx* by reading and writing values from/to *store*. It returns the updated store.
- *CreateArray ()*: creates an empty array.
- *WrapStore (store)*: creates a hashmap acting as a write-back cache for *store*. A dirty bit identifies modified data, allowing to separate read and write sets.
- *ExecuteAsync (tx, store)*: asynchronously executes *tx* by reading and writing values from/to *store*.
- *ExecuteSync (tx, store)*: synchronously executes *tx* by reading and writing values from/to *store*.

Algorithm 1 ProcessBlock executes the transactions in a block and coherently updates the state tree

```
1: function ProcessBlock(store, txs)
2:     i ← Iterator(txs)                                   ▷
3:     repeat
4:         i ← Next(i)                                     ▷
5:         tx ← GetValue(i)                                ▷
6:         store ← AnteHandleSync(tx, store)               ▷
7:     until HasNext(i)                                    ▷
8:     fwss ← CreateArray()                                ▷
9:     accum ← WrapStore(store)      ▷ Wrapped stores are hashmaps acting as write-back caches
10:    i ← Iterator(txs)                                   ▷
11:    repeat
12:        i ← Next(i)                                     ▷
13:        tx ← GetValue(i)                                ▷
14:        ws ← WrapStore(store)                           ▷
15:        fws ← ExecuteAsync(tx, ws)      ▷ fws is a future-like object
16:        fwss ← Append(fwss, fws)                        ▷
17:    until HasNext(i)                                    ▷
18:    i ← Iterator(txs)                                   ▷
19:    repeat
20:        i ← Next(i)                                     ▷
21:        tx ← GetValue(i)                                ▷
22:        fws ← PopFront(fwss)                            ▷
23:        ws ← Wait(fws)   ▷ Wait for future to be ready
24:        if IsRWConflict(accum, ws) then   ▷ we need to re-execute only if read-write conflicts arise
25:            accum ← ExecuteSync(tx, accum)              ▷
26:        else
27:            accum ← WriteSetUpsert(accum, ws)   ▷ accum is upserted with the values in ws
28:        end if
29:    until HasNext(i)                                    ▷
30:    WriteBack(accum, store)          ▷ Writes are propagated to the underlying store sequentially, avoiding the ambiguous state representation problem
31:    return
32: end function
```

- *Append (array, elm)*: adds *elm* to the end of *array*. It returns the newly created array.
- *PopFront (array)*: removes the first value of the array and returns it.
- *Wait (future)*: waits for *future* to complete.
- *IsRWConflict (firstCache, secondCache)*: checks if the read set of *secondCache* contains at least one value that is also present in the write set of *firstCache*.
- *WriteSetUpsert (firstCache, secondCache)*: merges *firstCache* and *secondCache*. The values in *secondCache* overwrite the ones in *firstCache*.
- *WriteBack (cache, store)*: the values in *cache* are written back to *store*.

We break down our implementation into three steps to better clarify how the algorithm operates.

A. Sequential step: reordering

Blockchains should not execute invalid transactions. For this reason, a piece of code called *ante handler* is executed before each transaction to perform stateless and stateful validity checks. Such checks may include signature verification, block expiration, and more. Often, in cryptocurrency-oriented blockchains, ante handlers manage the payment of the transaction fees to the block proposer, which implies that ante handlers must be executed sequentially because they all write to the same location. Thus, interleaving ante handlers and transactions prevents parallel transaction execution. We propose to process transactions in parallel after a sequential stage where all ante handlers are executed, as shown in Figure 4.

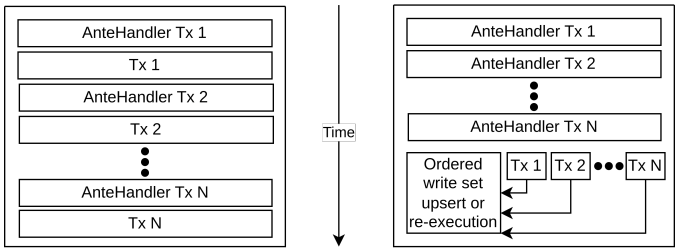


Fig. 4: Blockchain transaction execution: default (left) vs our proposal (right).

B. Parallel step: optimistic execution

After the sequential execution of all the ante handlers, transactions are executed in parallel. Each transaction is associated with a write-back cache and reads from/writes to it. Thus, each transactions is unaware of the updates performed by the others and its read and write sets can be determined based on the contents of the associated write-back cache. Consequently, peers can execute transactions in any order.

C. Sequential step: conflict resolution

To avoid the ambiguous state representation problem, we propagate the writes of the various transaction to the underlying state tree in the sequential order of how transactions appear in the block. This guarantees that the tree is updated deterministically. However, before applying updates, we need to avoid computational anomalies and may need to re-execute conflicting transactions. We decide if a transaction t_i must be re-executed based on the following observations.

- $ReadSet(\{t_0, \ldots, t_{i-1}\}) \cap ReadSet(\{t_i\}) \neq \emptyset$: no re-execution is needed as the state is not modified.
- $ReadSet(\{t_0, \ldots, t_{i-1}\}) \cap WriteSet(\{t_i\}) \neq \emptyset$: as each transaction is unaware of the modifications performed by the others, the values read by $\{t_0, \ldots, t_{i-1}\}$ are guaranteed to be unaffected by t_i's writes. Thus, no re-execution is needed.

- $WriteSet(\{t_0, \ldots, t_{i-1}\}) \cap ReadSet(\{t_i\}) \neq \emptyset$: t_i must be re-executed as it did not observe the modifications performed by its predecessors.
- $WriteSet(\{t_0, \ldots, t_{i-1}\}) \cap WriteSet(\{t_i\}) \neq \emptyset$: we differentiate two cases. If t_i performs updates, then there must also be a conflict between the read set of t_i and the write set of its predecessors. Thus, we are back to the previous case. If t_i performs blind writes, then we can propagate its writes to the underlying state tree without re-executing.

In short, re-execution is needed iff $WriteSet(\{t_0, \ldots, t_{i-1}\}) \cap ReadSet(\{t_i\}) \neq \emptyset$. In any other case, we can propagate the writes of t_i to the underlying state tree.

V. Experimental Evaluation

We integrated our algorithm within the Cosmos SDK to prove the feasibility of our solution and assess the potential performance gains. To this extent, we implemented a key-value store application on top of the Cosmos SDK and compared the average number of committed Transactions Per Second (TPS) when using the official release and our parallel implementation. The key-value store application is the same in both cases since our version of the SDK is fully compatible with the official release, allowing also other existing applications to switch between the two by only updating dependencies. We did not observe consensus disruptions in any of our tests, which hints at the determinism of our solution.

A. Environment, workload, and setup

We instantiated a network of 40 Virtual Machines (VMs) on AWS. Each VM belonged to the c6a.2xlarge instance family and was provided with 120 GB of gp3 SSD. We instantiated a Docker container in host network mode on top of each VM. Our tests are based on version 0.45.13-ics of the Cosmos SDK. Using workloads based on real data is non-trivial due to the level of cryptography involved. In particular, transactions are state-dependent, digitally signed, and associated with a nonce to prevent replay attacks, making them likely to fail if used in workloads. Thus, we used a Go porting[1] of the popular Yahoo! Cloud Serving Benchmark (YCSB) to generate the workloads for our tests. More specifically, we used the following workloads:

- update-heavy (or workload A for short);
- read-mostly (or workload B for short);
- read-latest (or workload D for short);
- Read-modify-write (or workload F for short).

Each workload's transaction generation follows a specific distribution, resulting in workloads containing conflicting transactions with varying patterns. We measured the performance of the system from an external client that uses multiple accounts to submit transactions in parallel. In each of the tests, the client submits 10^5 transactions, one to each peer in a round-robin fashion. We varied the number of peers in the network, the number of threads used by each peer, and the number of pre-existing records in the key-value store. When not differently stated, the network consists of 40 consensus nodes, and the number of pre-existing records amounts to $2{\cdot}10^5$. Additionally, when the number of threads is equal to one, the official release of the Cosmos SDK is used. When the number of threads is $n > 1$, our implementation is used with one main thread and $n - 1$ workers.

[1] https://github.com/pingcap/go-ycsb

B. Performance vs number of threads

The performance obtained by varying the number of threads is reported in Figure 5. As expected, increasing the number of threads produces better results overall. In our experiments, we noticed that increasing the block size while reducing the block production rate could increase the positive impact of transaction parallelization. Nonetheless, we decided to use the default configuration in this study. Parallel transaction execution provides better throughput independently of the workload. Even though the throughput gain is limited to a few percentage points, we note that this is achieved with workloads that perform simple computation with few operations, which is not the most favorable case for parallelism.

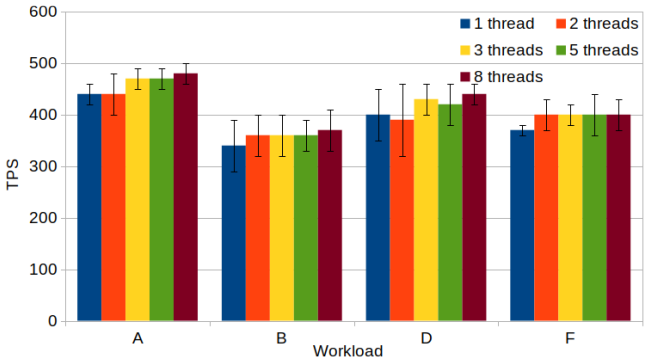
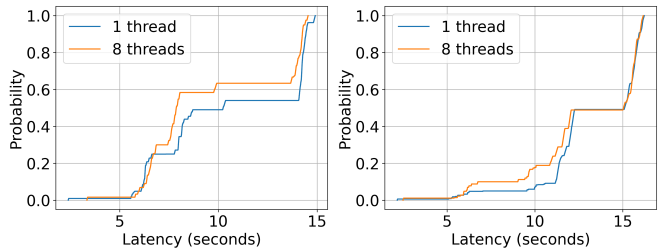


Fig. 5: Throughput while varying numbers of threads.

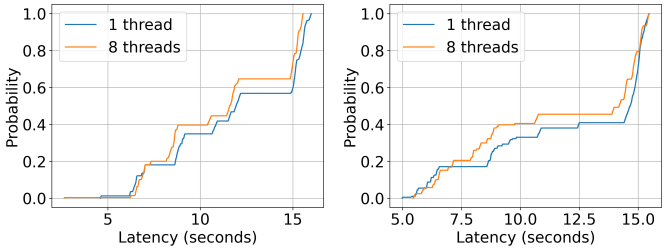
We report in Figure 6 the latency cumulative distribution with one (official release) and 8 threads. In all workloads, the parallel approach consistently provides better results than the single-threaded system. For example, in workload A, multithreaded execution reduces the median latency from 10.3 seconds to 7.9 seconds.

C. Performance vs system size

The performance obtained by varying the number of peers is reported in Figure 7. Parallel transaction execution consistently provides higher throughput except for the 4-node network with workload B. We did not observe a clear anticorrelation between the performance gains and the size of the network, which may become apparent on larger and more broadly distributed networks. Nonetheless, the number of blockchains having that kind of distribution is limited, thus we believe that our tests on 40 consensus nodes are significant for many real-word deployments.

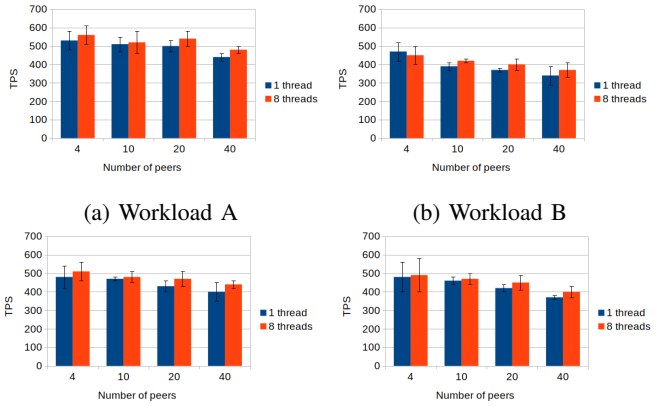


(a) Workload A (b) Workload B



(c) Workload D (d) Workload F

Fig. 6: Comparison of the latency cumulative distributions.



(a) Workload A (b) Workload B

(c) Workload D (d) Workload F

Fig. 7: Performance of different network sizes.

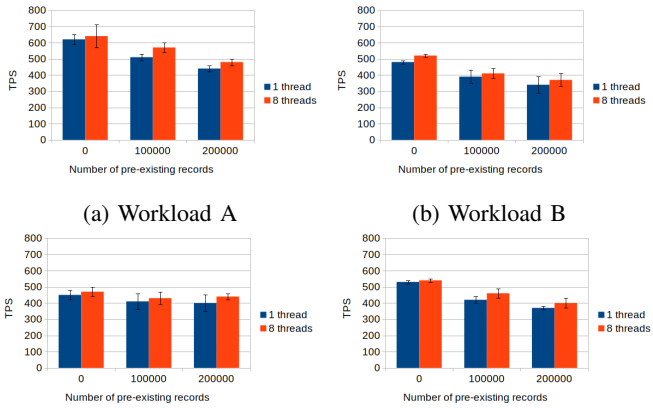
D. Performance vs database size

The performance obtained by varying the number of pre-existing records is reported in Figure 8. As for the previous cases, parallel transaction execution consistently provides higher throughput with similar gains. Nonetheless, such gains seem to increase with the number of pre-existing records, which is reasonable considering that more records imply a larger state tree that is slower to traverse. Thus, future research may explore the impact of parallel transaction execution on larger state trees.

E. Summary

In short, we can summarize our findings as follows:

- Parallel transaction execution is beneficial in terms of throughput and latency across all workloads.



(a) Workload A (b) Workload B

(c) Workload D (d) Workload F

Fig. 8: Performance while varying numbers of pre-existing records in the blockchain state.

- The ambiguous state representation problem poses significant constraints on parallel transaction execution, resulting in reduced throughput and latency gains.

VI. Conclusion

The ambiguous state representation problem may cause cross-peer state inconsistencies when transactions are executed in parallel. In this study, we proposed an algorithm that prevents the problem by guaranteeing that write operations are performed in order without completely sacrificing concurrency. We then integrated our algorithm within the popular Cosmos SDK to prove its feasibility and assess the potential performance gains. In particular, we conducted an extensive performance evaluation where our implementation consistently outperformed the sequential one by 5%-10% in terms of TPS under all conditions. However, this improvement alone is not sufficient to solve blockchain's scalability issues. Nonetheless, our measures have been taken from an external client, which means that our optimization produces tangible benefits for end users even with networks of tens of peers.

Future work will be aimed at parallelizing the execution of ante handlers and at better understanding the performance interdependencies between consensus, transaction scheduling, network connectivity, and the other modules composing blockchain.

Acknowledgment

The authors would like to thank Eliã Rafael de Lima Batista for his support with the cluster configuration; Lasaro Camargos, Adi Seredinschi, and Daniel Cason for their insights on the Cosmos ecosystem.

References

[1] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[3] S. Nathan, C. Govindarajan, A. Saraf, M. Sethi, and P. Jayachandran, "Blockchain meets database: Design and implementation of a blockchain relational database," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, p. 1539–1552, 2019.

[4] P. Akshatha and S. Dilip Kumar, "Mqtt and blockchain sharding: An approach to user-controlled data access with improved security and efficiency," *Blockchain: Research and Applications*, vol. 4, no. 4, p. 100158, 2023.

[5] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 368–377.

[6] E. a. Batista, E. Alchieri, F. Dotti, and F. Pedone, "Early scheduling on steroids: Boosting parallel state machine replication," *J. Parallel Distrib. Comput.*, vol. 163, no. C, pp. 269–282, may 2022. [Online]. Available: https://doi.org/10.1016/j.jpdc.2022.02.001

[7] V. Capocasale, D. Gotta, and G. Perboli, "Comparative analysis of permissioned blockchain frameworks for industrial applications," *Blockchain: Research and Applications*, vol. 4, no. 1, p. 100113, 2023.

[8] V. Mardiansyah and R. F. Sari, "Simblock simulator enhancement with difficulty level algorithm based on proof-of-work consensus for lightweight blockchain," *Sensors*, vol. 22, no. 23, 2022.

[9] Z. Chen, X. Qi, X. Du, Z. Zhang, and C. Jin, "Peep: A parallel execution engine for permissioned blockchain systems," in *Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part III 26*. Springer, 2021, pp. 341–357.

[10] A. Bigiotti, L. Mostarda, and A. Navarra, "Blockchain and iot integration for air pollution control," in *Advances on P2P, Parallel, Grid, Cloud and Internet Computing*, L. Barolli, Ed. Cham: Springer Nature Switzerland, 2024, pp. 27–38.

[11] O. S. Saleh, O. Ghazali, and N. B. Idris, "Enhancing academic certificate privacy with a hyperledger fabric blockchain-based access control approach," *SN Computer Science*, vol. 4, no. 5, 2023.

[12] V. Mardiansyah, A. Muis, and R. F. Sari, "Multi-state merkle patricia trie (msmpt): High-performance data structures for multi-query processing based on lightweight blockchain," *IEEE Access*, vol. 11, p. 117282 – 117296, 2023.

[13] Y. Fang, L. Cai, W. Qiu, F. Huang, and H. Hui, "Esm2-tree: An maintenance efficient authentication data structure in blockchain," in *Proceedings of the 35th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3603719.3603721

[14] J. Kwon and E. Buchman, "A network of distributed ledgers," 2019. [Online]. Available: https://v1.cosmos.network/resources/whitepaper

[15] A. Burgos, E. Alchieri, F. Dotti, and F. Pedone, "Exploiting concurrency in sharded parallel state machine replication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2133–2147, 2022.

[16] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone, "Reconfiguring parallel state machine replication," in *SRDS*, 2017.

[17] E. Alchieri, F. Dotti, and F. Pedone, "Early scheduling in parallel state machine replication," in *ACM SoCC*, 2018.

[18] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: Replication at the speed of multi-core," in *EuroSys*, 2014.

[19] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *25th Symposium on Operating Systems Principles*, 2015.

[20] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: execute-verify replication for multi-core servers," in *OSDI*, 2012.

[21] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *SRDS*, 2014.

[22] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.

[23] P. J. Marandi, C. E. B. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *ICDCS*, 2014.

[24] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[25] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich, "Blurring the lines between blockchains and database systems: the case of hyperledger fabric," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 105–122.

[26] C. Gorenflo, L. Golab, and S. Keshav, "Xox fabric: A hybrid approach to blockchain transaction execution," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020, pp. 1–9.

[27] 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.

[28] K. Olson, M. Bowman, J. Mitchell, S. Amundson, D. Middleton, and C. Montgomery, "Sawtooth: an introduction," 2018. [Online]. Available: https://www.hyperledger.org/wp-content/uploads/2018/01/Hyperledger_Sawtooth_WhitePaper.pdf

[29] V. Capocasale, D. Gotta, S. Musso, and G. Perboli, "A blockchain, 5g and iot-based transaction management system for smart logistics: An hyperledger framework," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, 2021, pp. 1285–1290.

[30] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1337–1347.

[31] X. Wei, S. Junyi, Z. Qi, and C. Fu, "Xuperchain: A blockchain system that supports smart contracts parallelization," in *2020 IEEE International Conference on Smart Internet of Things (SmartIoT)*, 2020, pp. 309–313.

[32] S. Baheti, P. S. Anjana, S. Peri, and Y. Simmhan, "Dipetrans: A framework for distributed parallel execution of transactions of blocks in blockchains," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 10, p. e6804, 2022.

[33] F. Schuhknecht, A. Sharma, J. Dittrich, and D. Agrawal, "chainifydb: How to get rid of your blockchain and use your dbms instead," 2020. [Online]. Available: https://www.cidrdb.org/cidr2021/papers/cidr2021_paper04.pdf

[34] Z. Peng, Y. Zhang, Q. Xu, H. Liu, Y. Gao, X. Li, and G. Yu, "Neuchain: a fast permissioned blockchain system with deterministic ordering," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2585–2598, 2022.

[35] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," *arXiv preprint arXiv:1901.01376*, 2019.

[36] R. Pan, C. Liu, G. Xiao, M. Duan, K. Li, and K. Li, "An algorithm and architecture co-design for accelerating smart contracts in blockchain," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[37] 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)*. IEEE, 2021, pp. 1–6.