SpecPool: Storage-Layer Speculation for Parallel Smart Contract Execution

Francisco Rola
INESC-ID, Instituto Superior Técnico,
University of Lisbon, Portugal
francisco.rola@tecnico.ulisboa.pt

Miguel Matos
INESC-ID, Instituto Superior Técnico,
University of Lisbon, Portugal
miguel.marques.matos@tecnico.ulisboa.pt

Michał Nazarewicz
No affiliation
mina86@mina86.com

Paolo Romano
INESC-ID, Instituto Superior Técnico,
University of Lisbon, Portugal
paolo.romano@tecnico.ulisboa.pt

***Abstract*—Modern blockchains face execution bottlenecks as rising transaction loads exceed the limits of sequential processing, threatening scalability. Exploiting parallelism safely, while preserving determinism, is essential to address this challenge.**

SPECPOOL transforms the blockchain mempool—traditionally a passive queue—into an active preprocessing engine. While transactions await consensus, SPECPOOL predicts a likely block execution order using mempool-visible signals such as fees, arrival time, and coarse batching windows while respecting per-sender nonce ordering. It then predicts potential conflicts between transactions by combining static analysis of contract bytecode with historical access patterns. Transactions are subsequently speculatively pre-executed in parallel according to the predicted order within sandboxed environments, extracting transaction read/write sets and invocation chains. This metadata enables conflict-aware validation, maximizing concurrency while preserving consensus-consistent ordering. Operating at the storage layer, SPECPOOL requires no modifications to virtual machine semantics or consensus protocols, simplifying adoption in existing blockchain systems.

We implement SPECPOOL in the Cosmos framework and evaluate it using workloads derived from the Injective blockchain. Experiments demonstrate up to a $60\times$ reduction in block validation latency and 20% throughput improvement over prior parallel execution approaches. We additionally analyze the effects of block predictability and transaction mempool visibility in a high-frequency blockchain, showing that SPECPOOL maintains accurate order prediction and effective parallelism even under Injective's block time of 0.63 seconds.

***Index Terms*—mempool optimization, blockchain systems, parallel transaction validation, conflict-aware scheduling**

I. INTRODUCTION

Modern blockchains face increasing pressure to deliver efficient and performant execution engines [1]–[3]. While consensus protocols have seen significant performance improvements [4], [5], transaction execution and validation remain constrained by largely sequential processing models that fail to fully exploit modern multicore machines. Blockchains such as Ethereum [1] and Cosmos [6] follow an *Order-Execute (OE)* paradigm, in which transactions are first totally ordered by consensus and then executed sequentially during block validation. Although this design simplifies correctness and determinism, it fundamentally limits scalability by serializing state modifications and placing transaction execution on the consensus critical path. Several works [7]–[12] have attempted to improve performance within the OE model by exploiting commutativity and optimistic concurrency during block validation. In particular, systems such as Block-STM [8] demonstrate that substantial gains are possible by parallelizing transaction execution after block finalization; however, these approaches begin execution only once the final block order is known, leaving the time spent waiting for consensus unexploited.

Speculative execution has a long history in distributed systems, from optimistic concurrency control [13] to state machine replication [14]–[17]. The core insight is that useful computation can proceed based on likely future states before they are definitively known. This observation is particularly compelling in blockchains, where transactions typically spend non-trivial time in the mempool before being included in a block. Hyperledger Fabric [18], for example, executes transactions before ordering only to generate read/write sets that peers use to approve/authorize the transaction result; conflicts are not prevented or resolved early, but are detected after ordering during block validation (e.g., MVCC/version checks). FORERUNNER [19] explores speculative execution across multiple possible transaction orderings by deriving global constraints that must hold for all feasible permutations, instantiating them once the final order is revealed. While effective, such approaches introduce significant coordination, constraint-solving, or execution-engine complexity, making them difficult to deploy in existing blockchain systems.

In this work, we build on the observation that the mempool—traditionally treated as a passive buffer—offers an opportunity to perform substantial work outside the block validation critical path. However, transforming this opportunity into a practical system that integrates with existing blockchains raises several challenges that prior approaches do not fully address. First, mempools typically contain more transactions than a block can include, and a transaction that appears in a finalized block may not have been observed in the local mempool of every node, making exact block prediction infeasible. Second, effective pre-execution does not require

perfect prediction of the entire block order; instead, it suffices to correctly anticipate relative ordering and conflicts among transactions that interact, particularly those targeting the same smart contracts. Third, speculative execution must remain lightweight, avoiding intrusive changes to the virtual machine or consensus protocol, while enabling parallel block validation and minimizing re-execution when speculation is inaccurate.

We address these challenges with three key ideas. First, rather than speculating over multiple possible orders, SPECPOOL predicts a single likely block execution order derived from observable mempool signals and protocol constraints. This significantly reduces speculative overhead while achieving low misprediction rates in practice. Second, SPECPOOL combines lightweight static analysis of smart contract bytecode with a cache of historical contract access patterns to predict transaction conflicts ahead of execution. This hybrid approach enables conflict-aware scheduling even when mempool information is incomplete. Third, metadata extracted during pre-execution—including read/write sets and invocation chains—is used to schedule transactions across worker threads during block validation, reducing cascading aborts and limiting the cost of incorrect speculation. Crucially, our design operates entirely at the storage layer, requiring no modifications to the transaction execution engine or virtual machine, easing the integration into existing blockchains.

A key concern for any mempool-based speculative system is the presence of Miner/Maximal Extractable Value (MEV) transactions. MEV transactions are often injected opportunistically and may bypass public mempool visibility, making them difficult to predict and potentially invalidating speculative results for subsequent transactions. To address this, we study transaction visibility in a real-world public blockchain mempool and show that approximately 85% of transactions are observable prior to block inclusion. We further evaluate worst-case scenarios in which MEV transactions appear at the beginning of a block, maximizing their disruptive potential. Our results show that even under such adversarial conditions, SPECPOOL's performance degrades gracefully.

We implement SPECPOOL as an extension to CosmWasm [6] and evaluate it using workloads derived from the Injective blockchain, a Cosmos SDK–based Layer-1 (https://docs.injective.network/). Injective is a particularly challenging deployment environment due to its low average block time (approximately 0.63 s), which limits the available window for speculative preprocessing. Despite this constraint, our evaluation shows that transactions typically remain visible in the mempool long enough to enable effective pre-execution. Across our workloads, SPECPOOL achieves up to a **60× reduction in block validation latency** and up to **20% higher throughput** compared to state-of-the-art speculative execution systems such as Block-STM [8]. SPECPOOL scales effectively with increasing thread counts and maintains robust performance even under high transaction reordering rates, thanks to its conflict-aware scheduling strategy.

Contributions. Our contributions are: (i) a speculative execution framework that exploits the temporal gap between mempool admission and consensus to reduce block validation latency while remaining compatible with existing blockchain execution models; (ii) a block order prediction mechanism that infers likely intra-block transaction ordering from mempool transaction visibility and protocol constraints; (iii) a conflict-aware pre-execution and validation algorithm that combines static smart contract analysis with historical access patterns to minimize re-execution under inaccurate speculation; (iv) SPECPOOL, an implementation of this design for the CosmWasm execution environment; and (v) an extensive experimental evaluation using real and synthetic workloads demonstrating up to 60× lower block validation latency and up to 20% higher throughput.

II. BACKGROUND & RELATED WORK

This section provides an overview of the blockchain execution model and discusses prior work.

1) Blockchain Execution Model: Blockchain transactions follow a lifecycle: users submit transactions to the *mempool*, validators select a subset for inclusion, and consensus fixes their total order within a block. Once finalized, transactions are executed during block validation and their state updates are committed. Although the final order is unknown prior to consensus, it is constrained by protocol rules and validator heuristics—a property that SPECPOOL exploits to increase the likelihood of correct successive speculative execution.

Transaction Nesting. Smart contracts are deterministic, stateful programs with private key–value storage and the ability to invoke other contracts. Such calls may trigger further nested invocations, forming dynamic *invocation chains*. Nesting enables modular contract design but introduces nontrivial dependencies across contracts, making accurate conflict prediction essential for safe parallel execution.

Environment Variables. Transaction execution may depend on environment-specific parameters such as block height and timestamp. Since these values are fixed only after block inclusion, speculative systems must conservatively account for environment-sensitive operations to avoid invalid assumptions.

Ordering Constraints. The main hard constraint restricting transaction ordering is sender nonces, which enforce strict intra-sender sequencing. Furthermore, on Cosmos-based chains, IBC channels require ordered delivery of cross-chain packets [20]. In addition, fee-based prioritization and proposer-specific policies influence transaction selection and ordering, though they are not consensus-mandated. Together, these constraints significantly narrow the space of feasible block orders.

Importantly, speculative execution does not require predicting a total order of all transactions. Instead, it suffices to correctly identify relative ordering and dependencies among transactions that interact through shared state, while transactions accessing disjoint state can safely execute in parallel.

Scope of Execution. In this work, we focus on optimizing transaction execution and validation rather than the consensus layer. We assume an underlying consensus protocol that produces a totally ordered block of transactions and treat this order as a fixed input to the execution layer.

2) Related Work: Most traditional blockchains adopt an *Order-Execute* (OE) paradigm [8], [10]–[12], [21], in which transactions are first totally ordered by consensus and then executed during block validation. This model guarantees deterministic state transitions but underutilizes multicore hardware, shifting the scalability bottleneck from consensus to execution [22]–[24]. Recent work has shown that exploiting commutativity and optimistic concurrency within OE can significantly improve performance, but execution in these systems begins only after block finalization.

Block-STM [8] leverages software transactional memory (STM) to speculatively execute transactions after consensus has finalized a block, when the transaction order is already determined. Its core idea is to exploit intra-block parallelism by executing transactions as STM transactions that dynamically track read and write sets and detect conflicts at runtime. When concurrent transactions access overlapping state in conflicting ways, STM aborts and re-executes them according to the predetermined order. This design allows Block-STM to improve throughput over sequential execution while maintaining deterministic semantics.

However, since speculation is confined to the post-consensus phase, Block-STM cannot exploit mempool visibility to perform early conflict detection, pre-execution, or scheduling. As a result, it misses opportunities to overlap computation with consensus and to use speculative insights to guide transaction grouping and execution. In contrast, SPECPOOL extends speculative execution into the pre-consensus window, using mempool-derived predictions to inform both pre-execution and parallel validation, while preserving correctness.

An alternative *Execute-Order-Validate* (EOV) paradigm is employed by systems such as Hyperledger Fabric [18], [25], [26]. In Fabric, transactions are speculatively simulated prior to ordering to generate read/write sets for endorsement, and conflicting transactions are aborted during validation. While this enables concurrency, aborted transactions still consume block capacity and speculative execution is local to individual smart contracts, limiting scalability under contention.

SPECPOOL is also related to parallel state-machine replication and ordered concurrency-control systems such as Eve [27], Rex [28], and Saad et al. [29], which exploit concurrency while preserving a deterministic request or transaction order. These systems are complementary but distinct: they parallelize execution once the workload/order is known, whereas SPECPOOL uses mempool visibility to speculate before consensus and validates results against the finalized block order. SPEEDEX [30] also demonstrates application-aware parallel execution for decentralized exchanges (DEX), but targets a specialized DEX design, while SPECPOOL operates at the storage layer of a general smart-contract runtime.

Other work leverages application-level semantics to reduce conflicts. Analyses of Ethereum workloads show that contention can limit parallel speedup [22], [24], [31]. ParBlockchain [25] constructs dependency graphs to guide parallel execution, but requires adherence to a predefined execution order and does not exploit pre-consensus speculation.

FORERUNNER [19] speculatively executes transactions across multiple possible orders, deriving global constraints that must hold regardless of the final block permutation. While this approach increases concurrency, it incurs significant coordination and constraint-solving overhead. SPECPOOL instead predicts a single likely block order, combines static contract analysis with historical access patterns, and dynamically schedules transactions to threads, achieving high parallelism with lower overhead and minimal re-execution.

III. SPECPOOL

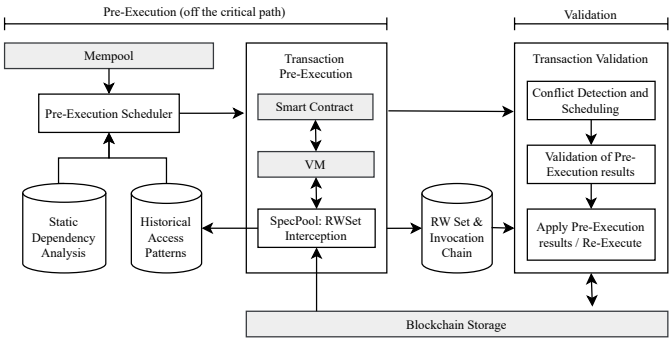

Fig. 1. SPECPOOL overview. Shaded boxes represent unmodified components.

SPECPOOL comprises a two-phase pipeline, illustrated in Figure 1: (i) speculative pre-execution of mempool transactions and (ii) conflict-aware transaction validation.

In the pre-execution phase (§III-A), transactions entering the mempool are tentatively executed in a sandboxed environment to extract detailed metadata, including per-transaction read/write sets and invocation chains. Transactions are executed in parallel according to a predicted block order derived from mempool information, while preserving semantic constraints such as sender nonces and protocol-specific ordering rules. This metadata is accumulated over time and combined with historical access patterns and static analysis of contract bytecode to inform conflict-aware grouping and scheduling for both pre-execution and subsequent validation.

The validation phase (§III-B) leverages this metadata to construct a schedule that maximizes parallelism without compromising correctness. Transactions are considered conflicting if they access the same smart contracts or overlapping invocation chains. Non-conflicting transactions are validated concurrently, while those with detected conflicts are serialized according to consensus order. Pre-execution results are replayed during validation, and transactions are re-executed only if speculation was incorrect or if dynamic dependencies emerge at runtime. When a transaction is validated without conflicts, its pre-executed read/write set is applied directly to state. A sequential fallback mechanism ensures that correctness is preserved.

By overlapping speculative computation with consensus, SPECPOOL reduces the latency between block commitment and state finalization. Unlike approaches that parallelize execution only after consensus, SPECPOOL exploits the entire consensus window to overlap communication and processing.

Furthermore, it achieves these gains without modifying the core execution engine, operating solely at the storage layer and simplifying integration with existing blockchain systems [19].

A. Transaction Pre-execution

The pre-execution phase transforms the mempool from a passive buffer of pending transactions into an active preprocessing engine that anticipates and prepares for block execution. During this phase, transactions are tentatively executed in a sandboxed environment built on top of the smart contract runtime, without persisting any state changes.

The proposed pre-execution engine proceeds in four conceptual steps. First, it predicts a likely block-level ordering of transactions using information available in the mempool, guided by the ordering constraints imposed by the smart contract execution model (§II-1), such as fee-based prioritization. Second, it identifies potential conflicts between transactions by combining static analysis of contract bytecode and transaction payloads with historical access patterns collected from prior executions. This analysis enables the construction of conflict groups that capture transactions likely to interact due to overlapping smart contract state.

Third, transactions are scheduled for parallel execution by respecting both the predicted block order and the conflict groups identified in the previous step: transactions within the same group are serialized, while independent groups are executed concurrently. Finally, each group is pre-executed in parallel within isolated sandbox environments, allowing SPECPOOL to observe the read and write sets of individual transactions and to record detailed execution metadata. By collecting read/write sets across all mempool transactions, SPECPOOL can infer inter-transaction dependencies, detect conflicts early, and construct an efficient parallel validation schedule for the subsequent phase.

1) Transaction Block Order Prediction: The first step of pre-execution is to predict a likely execution order of transactions in the next block using information available from the mempool. Since mempool sizes are typically much larger than individual blocks, not all transactions will be included in the next block and predicting the exact set of included transactions is infeasible. Instead, SPECPOOL focuses on predicting the relative order of transactions that are most likely to interact (e.g., those targeting the same smart contract or lying on overlapping invocation chains), which is most critical for conflict-aware scheduling.

We factor ordering into **hard constraints** and a **configurable tie-breaker**. Hard constraints are protocol rules that must always hold, most notably per-sender nonce ordering: for any account a, transactions from a must appear in increasing nonce order. These constraints induce a partial order that SPECPOOL respects by maintaining per-sender FIFO queues.

When multiple transactions are admissible under the hard constraints, SPECPOOL breaks ties using mempool-visible signals that are broadly available across blockchains: (i) **fee priority** (e.g., total fee, tip, or gas price), (ii) **arrival-time** (first-seen timestamp), and (iii) **batching effects** modeled as coarse time windows. The predictor is parameterized by $\Theta = (W, \Delta)$, where W is the window duration and Δ is the fee-bucket width used to group transactions with similar fees. These parameters bound the comparison scope of the predictor by restricting ordering decisions to transactions that are close in time and fee, avoiding global comparisons across the entire mempool while preserving locality among transactions most likely to co-occur in a block. In practice, W and Δ are tuned offline using historical mempool traces and the corresponding finalized block orders (obtained via node queries), and then fixed for deployment on the target chain.

Order construction proceeds as follows: transactions are first partitioned into temporal windows (based on mempool reception time) of length W and processed in increasing window id; within each window, transactions are grouped into fee buckets of width Δ and processed in decreasing bucket id; within each fee bucket, transactions are grouped by sender and sorted by arrival time, and the final sequence is produced by repeatedly selecting among senders the head-of-line transaction with the highest fee (breaking ties by earlier arrival). Portability across blockchains is achieved by instantiating the same template with chain-specific fee definitions and deployment-specific parameter choices for W and Δ. In §V-D, we evaluate how well SPECPOOL predicts the relative order of dependent transactions from mempool contents.

2) Static Dependency Analysis and Historical Access Patterns: Before pre-execution, SPECPOOL performs static analysis of contract bytecode and transaction payloads to extract predictive information about execution behavior and dependencies. Contract code is analyzed at deployment time, while each transaction is analyzed upon submission. A primary goal of this analysis is to identify environment-sensitive operations, such as references to block height, timestamp, or balances exposed through system APIs. Transactions that depend on such values are conservatively flagged for deferred validation or re-execution under the finalized state snapshot.

The analysis also anticipates contract-to-contract invocation chains. Many call targets are statically determinable, allowing SPECPOOL to reconstruct likely invocation paths and schedule transactions in a way that reduces speculative conflicts. Payload inspection further classifies transactions into categories such as code upload, instantiation, execution, migration, and administrative updates. For example, code upload transactions are computationally expensive but do not access contract state and can therefore be safely parallelized. Instantiation transactions typically operate on fresh state unless they include nested sub-messages, while execution transactions often access shared state and trigger additional contract calls, requiring deeper dependency analysis.

Despite these efforts, some dependencies remain unpredictable at compile time—for example, when call targets are determined by runtime data or by query results. To mitigate this limitation, SPECPOOL incorporates profiling of historical access patterns, including observed read/write sets and invocation chains from prior executions. Specifically, for each smart contract instance and entry point, SPECPOOL maintains

a cache of the invocation chains observed during system execution, together with their associated read/write sets (e.g., which state items are typically read or updated). During pre-execution, SPECPOOL then uses these empirical observations to refine dependency predictions over time, improving the accuracy of conflict detection and scheduling, especially when transaction-specific pre-execution metadata is unavailable, as evaluated in Section V-C.

3) Conflict-aware Parallel Pre-execution: Given a predicted block order and dependency information from static analysis and historical access patterns, SPECPOOL performs conflict-aware parallel pre-execution as shown in Algorithm 1.

Algorithm 1: Pre-execution Phase

Input: **tx_list**: list of transactions to pre-execute;
num_threads: number of worker threads
Output: **pre_exec_results**: list of results produced by pre-executing each transaction, including read-write sets per contract

```
1  Function PreExecuteTxs:
2      conflict_groups ← BuildPreExecutionGroups(tx_list) ;
3      foreach group in conflict_groups parallel do
4          foreach tx in group do
5              tx.result ← Process(tx)
6      DiscardTemporaryState() ;
7      return CollectResults(tx_list) ;
8  Function BuildPreExecutionGroups:
9      contract_map ← ∅;
10     foreach tx in tx_list do
11         foreach contract in GetInvolvedContracts(tx) do
12             contract_map[contract] ← contract_map[contract] ∪ {tx} ;
13     return UnionFind(contract_map) ;
```

The pre-execution process begins by partitioning transactions into conflict-aware groups (line 2). To construct these groups, SPECPOOL first determines, for each transaction, the set of contracts it may interact with (line 11). This is accomplished by the `GetInvolvedContracts` function, which combines statically determinable call targets extracted from contract bytecode with historical information such as previously observed invocation chains and read/write sets. This hybrid approach allows SPECPOOL to anticipate interactions that cannot be resolved through static analysis alone, including those involving dynamic dispatch or indirect calls.

Using this information, SPECPOOL builds a reverse index mapping contracts to the transactions that access them (lines 10–12). Transactions that share access to at least one contract are conservatively merged into the same group using a Union-Find data structure (line 13). Each transaction initially forms its own group, and groups are merged whenever a shared contract dependency is detected. Union-Find enables efficient construction of these groups and ensures that potentially conflicting transactions are serialized, while transactions with disjoint contract access remain in separate groups.

Once conflict-aware groups are established, SPECPOOL pre-executes them in parallel (line 3). Each group is assigned to a dedicated worker thread and executed within an isolated sandbox environment. Transactions within a group are processed sequentially (lines 4–6) to preserve intra-group dependencies and respect the predicted block order, while different groups execute concurrently. Execution follows the standard transaction processing pipeline, but all state changes are confined to the sandbox and are not persisted.

During execution, SPECPOOL records detailed metadata for each transaction, including per-contract read and write sets and observed invocation behavior. After all groups complete execution, the sandboxed state is discarded (line 6), and the collected metadata is returned (line 7). This information is used during validation to construct a dependency-aware execution schedule, detect conflicts, and determine whether speculative results remain valid under the finalized block.

In summary, SPECPOOL uses conflict-aware grouping to increase pre-execution parallelism and to extract per-transaction metadata (e.g., predicted dependencies and read/write summaries). We next show how this metadata is used during validation to reduce re-execution and accelerate block processing.

B. Transaction Validation

The validation phase begins after consensus and leverages the metadata built in the previous phase to construct a parallel conflict-aware schedule subject to the constraint that the transaction ordering is determined by consensus. The schedule determines which transactions can be validated concurrently without violating dependency constraints. When the speculation is correct, transaction execution is skipped and the write-set obtained in pre-execution is directly applied to the final state. Otherwise, the affected transaction is re-executed together with any dependent transactions.

1) Conflict Detection and Scheduling: Since transactions can only conflict if they involve a common smart contract, SPECPOOL starts by assigning transactions that target completely disjoint sets of contracts to different groups. This information comes from the access patterns obtained during pre-execution. Two transactions conflict when one transaction attempts to read a key that has been modified by another transaction that comes before it in consensus order. These conflicting transactions are grouped together and scheduled to run within the same thread, which ensures that they are processed serially and in consensus order. Non-conflicting transactions are assigned to different groups and can run independently regardless of the consensus order.

It is important to note that some transactions may fail to execute during pre-execution, in which case we do not have detailed metadata about the invocation chain and predicted read and write set for that transaction. These failures happen because the contract state might not be semantically valid

during pre-execution. For example, a transaction may attempt to transfer tokens from an account that was modified by a previous transaction and those tokens might no longer be available. In these scenarios where a transaction does not have metadata from pre-execution, we rely on the historical data to approximate the accesses and invocation chains to make an informed decision on where to place these transactions.

2) Validation of Pre-Execution Results: The validation process for individual transactions follows a two-stage approach: read validation followed by write application. It begins by checking whether pre-execution completed successfully for the entire transaction invocation chain. If all calls in the chain succeeded, the system proceeds to validate the transaction's reads against the current state. However, if any part of the invocation chain failed during pre-execution, the transaction is re-executed from scratch. In this case, no further validation is necessary, as re-execution occurs in consensus order.

During read validation, the system checks that the read accesses recorded during pre-execution match the current global state of the blockchain. This step ensures that the transaction has not read stale or invalid data during pre-execution and that its assumptions about the state remain valid.

In this case, the transaction's write set is applied to the global state and the transaction execution is entirely skipped. These writes are applied to each contract involved in the transaction's invocation chain, ensuring that all affected contract states are updated in accordance with the pre-execution results. However, the transaction is only committed if every sub-transaction in the invocation chain passes the validation stage. If one verification fails, the entire transaction must be re-executed, similar to how failed pre-execution cases are handled. This final verification ensures that SPECPOOL only commits transactions whose write sets are consistent with the expected execution order, while all others are re-executed.

3) Handling Validation Failures and Dynamic Dependencies: Despite conflict-aware scheduling informed by pre-execution, transactions may still fail validation if the assumptions made during pre-execution no longer hold. Such failures can arise when the speculative ordering of conflicting transactions differs from the finalized block order, or when transaction behavior depends on dynamic state or environment-sensitive conditions that were not fully captured during pre-execution.

SPECPOOL treats pre-execution only as an optimization: speculative writes are kept in per-transaction buffers and are never applied to the global state until validation succeeds. A speculative result is committed only if all reads recorded during pre-execution still match the finalized state at validation time. If validation fails, if pre-execution failed, or if a transaction was valid in the mempool but becomes invalid because a preceding block changed the relevant state, SPECPOOL discards the buffered speculative result and re-executes the affected transaction in the finalized consensus order. Any dependent transactions whose speculative results may have been invalidated are also revalidated or re-executed in consensus order. Thus, no rollback of committed state is required, and all nodes execute any deferred transactions in the same deterministic order fixed by consensus. Prediction errors and incomplete mempool visibility therefore affect only performance, by increasing re-execution, while the final state remains equivalent to baseline sequential execution.

More complex scenarios arise when re-execution reveals previously unobserved execution paths, such as conditional branches or environment-triggered logic that cause a transaction to invoke additional smart contracts not predicted during pre-execution. In such cases, the transaction's dynamic access pattern may violate the original conflict grouping assumptions used during validation scheduling. To mitigate this risk, SPECPOOL adopts a conservative scheduling strategy that assigns validation threads based on the full invocation tree observed during pre-execution, rather than only top-level contract accesses. This ensures that transactions with known or historically observed nested interactions are assigned to the same validation thread, significantly reducing the likelihood of missed inter-contract dependencies.

If, despite these precautions, validation reveals new contract interactions that span multiple validation threads, SPECPOOL triggers a fallback mechanism. The affected transaction, along with any transitively dependent transactions, is re-executed sequentially in consensus order. The new interaction pattern is recorded in the historical profile, reducing the likelihood of encountering the same unexpected path in future blocks.

By treating parallel validation as an optimization and preserving a well-defined sequential fallback, SPECPOOL ensures that correctness is never compromised even in the presence of dynamic, state-dependent contract interactions.

IV. IMPLEMENTATION

SPECPOOL is implemented in Rust and Python ($\sim$ 7000 lines of code) and built as an extension to the CosmWasm SDK (https://github.com/CosmWasm/cosmwasm), a Rust-based framework for developing systems within the Cosmos ecosystem. This implementation choice leverages CosmWasm's smart contract execution environment and is agnostic of any specific Cosmos blockchain. The implementation closely follows the design discussed in §III with a few practical modifications that we discuss below. To evaluate SPECPOOL with a realistic workload we also developed a tool to extract and parse the blocks from any Cosmos blockchain. A portion of the resulting trace is used to evaluate SPECPOOL in realistic conditions as discussed in §V.

CosmWasm Integration. The implementation integrates SPECPOOL's two-phase execution pipeline directly into the CosmWasm runtime by instrumenting the storage layer. This design ensures compatibility with existing smart contract execution semantics as it does not require changes to the core virtual machine (VM) that executes the transactions. By isolating modifications to the storage backend and execution orchestration, the CosmWasm VM remains decoupled from SPECPOOL, requiring updates only when the underlying storage representation changes.

Pre-Execution Phase: Capturing Accesses and Invocation Chains. During pre-execution, transactions run against a

sandboxed state and do not persist changes. We intercept CosmWasm's storage and query interfaces to record read/write events and reconstruct contract invocation chains, including submessages and reply callbacks. To support this tracing, we add lightweight runtime instrumentation that propagates an execution context across nested calls.

Memoization and Access Pattern Caching. Each transaction's pre-execution result includes its read/write sets and full invocation chain. SPECPOOL memoizes these results in a data structure used by the validation scheduler to detect conflicts and by the validator to ensure consistency with pre-execution outcomes. This memoization avoids recomputing access patterns and improves scheduling efficiency. After a transaction commits, its metadata is stored in a historical access-pattern cache if it introduces previously unseen patterns. This cache then informs future scheduling decisions.

Validation Phase: Deterministic Parallel Execution. During validation, transactions are re-executed deterministically using CosmWasm's execution guarantees and strict contract isolation, where each contract has its own isolated storage. The scheduler uses metadata from pre-execution to identify independent transactions and execute them in parallel without state conflicts. Conflict detection is based on comparing read/write sets and cross-contract invocation dependencies.

Mempool. Our implementation models the mempool as an in-memory array with a configurable size that receives transactions at a configurable rate, loaded from a trace file. This modeled mempool represents one validator's local view: real validators may observe different transaction orders, but such discrepancies only affect speculation accuracy, since committed results are always validated against the finalized consensus order. After each mempool transaction batch retrieval (simulated by reading the next batch of pending transactions from the trace), SPECPOOL applies the block order prediction procedure to determine a likely execution order for transactions with potential dependencies, before proceeding with the rest of the pre-execution pipeline.

V. EVALUATION

We evaluate SPECPOOL against Cosmos sequential execution and Block-STM. We first report block validation latency and throughput under varying thread counts and pre-execution/validation splits (§V-A). We then measure mempool visibility and mempool-to-inclusion latency, and use these results to study robustness to unseen/private (MEV) transactions (§V-B, §V-C). Next, we evaluate intra-contract block order prediction accuracy and study how transaction arrival rate and shuffling affect prediction and misspeculation (§V-D, §V-E). Finally, we analyze sensitivity to block size (§V-F).

The workloads used for the throughput and block validation latency experiments are derived from the Injective blockchain, focusing on interactions with the Astroport decentralized exchange, a sophisticated automated market maker (AMM) built on Injective. We emulate a common DeFi swap scenario by invoking the token swap entry point of the Astroport Router contract, which internally interacts with multiple contracts such as pair and token modules to execute decentralized exchange trades. To study varying contention levels, we generate two workload variants: one with 50 markets (low contention) and one with 5 markets (medium contention), where each swap selects a market uniformly at random.

In addition to these controlled workloads, we analyze measurements collected from live Injective nodes. By querying mempool contents and observing blocks as they are produced, we construct a dataset that captures real transaction visibility, block inclusion latencies, and the predictability of intra-block transaction ordering.

All experiments were conducted on a machine equipped with two Intel® Xeon® Gold 5320 CPUs (52 cores each, 2.2 GHz) and 192 GB of RAM, running Ubuntu 22.04 LTS.

Implementation and Evaluation Simplifications. To isolate the impact of speculative pre-execution and parallel validation, our implementation uses two controlled simplifications. First, all experiments operate over in-memory storage, removing noise from disk I/O and database overhead. Second, we abstract away the consensus layer by treating transactions as immediately available for validation once pre-executed. This abstraction implicitly assumes that consensus is not the throughput bottleneck, i.e., that the consensus pipeline sustains higher throughput than the transaction execution engine in the regimes we study. This assumption is consistent with prior measurements on modern blockchain systems [22]–[24] and allows us to focus on our primary contribution: accelerating the transaction execution and validation pipeline. In practice, these simplifications do not limit the applicability of our results: they provide an upper bound on execution-layer performance and quantify the benefits of SPECPOOL without confounding effects from network latency or consensus dynamics.

Implementation of Baselines. For comparison, we implemented Block-STM [8] in CosmWasm. Block-STM assumes a single global ledger view, enabling speculative parallel execution with conflict detection and resolution based on overlapping transaction accesses. Since CosmWasm natively uses per-contract segregated storage, we extended it with a global storage abstraction that preserves contract isolation while providing a unified key-value view for conflict analysis. This modification allows us to faithfully reproduce Block-STM's speculative execution behavior while maintaining CosmWasm's deterministic and secure execution semantics.

A. Latency and Throughput Analysis

We begin by measuring *block validation latency* and *throughput* under varying contention levels and thread counts. Block validation latency is defined as the interval between the moment a block is committed by consensus and the point at which all transactions within the block have been fully processed by the execution engine. Throughput is measured as the number of transactions processed per second (TPS).

For the sequential baseline and Block-STM, which execute transactions after block finalization, throughput is given by:

$$\text{Throughput}_{\text{baseline}} = \frac{N}{Time_{\text{exec}}}$$

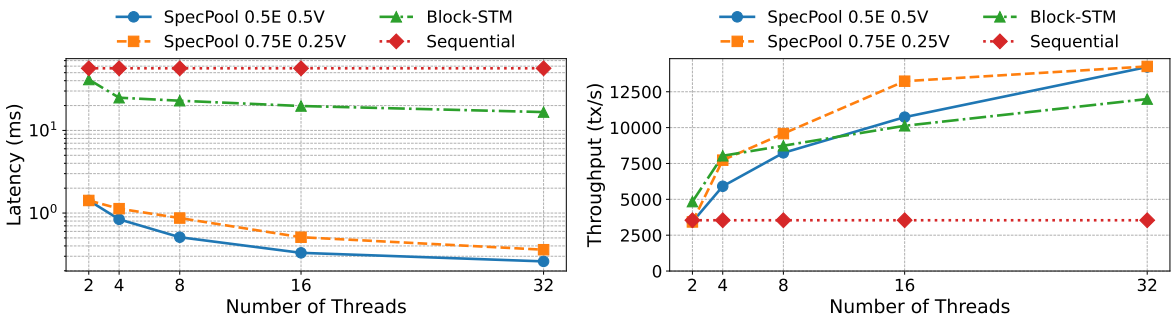

Fig. 2. Latency (left) and throughput (right) - low contention workloads.

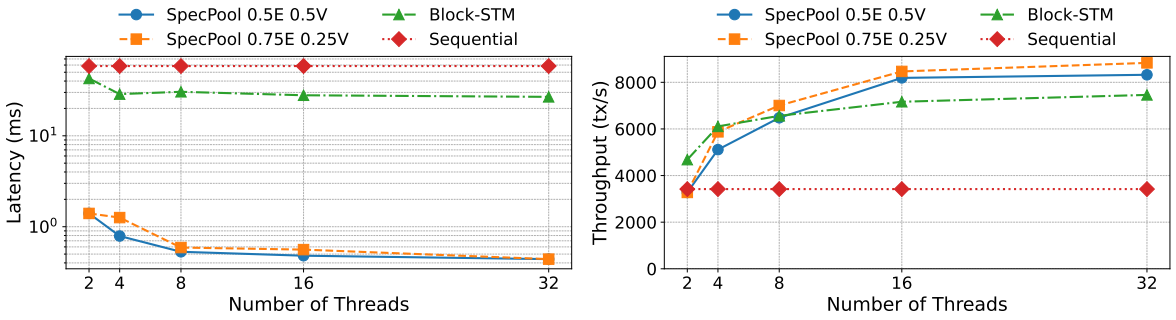

Fig. 3. Latency (left) and throughput (right) - medium contention workloads.

where N is the number of transactions in the block and $Time_{\text{exec}}$ is the total execution time after block finalization.

SPECPOOL, in contrast, overlaps pre-execution and validation across consecutive blocks. Its throughput is therefore bounded by the slower of the two phases:

$$\text{Throughput}_{\text{SPECPOOL}} = \frac{N}{\max(Time_{\text{pre}}, Time_{\text{val}})}$$

where $Time_{\text{pre}}$ and $Time_{\text{val}}$ correspond to pre-execution and validation durations, respectively. This formulation captures SPECPOOL's ability to overlap consensus and block execution. For these experiments, we fix the block size to 200 transactions; the impact of varying block sizes is studied in §V-F.

Configurations are denoted as SPECPOOL xE yV, where x and y indicate the fraction of threads allocated to pre-execution and validation, respectively. For all experiments, SPECPOOL and Block-STM are evaluated under the same total thread budget. While Block-STM assigns all threads to post-consensus speculative execution, SPECPOOL partitions the same threads between pre-execution and validation.

The results for the low contention workload are shown in Figure 2. SPECPOOL configurations consistently outperform both Cosmos and Block-STM baselines. Latency decreases steadily with more threads, reaching as low as 0.26 ms with 32 threads, while throughput scales near-linearly, peaking at over 14,200 TPS for SPECPOOL 0.75E 0.25V, where more threads are dedicated to pre-execution.

Block-STM achieves higher throughput at low thread counts because it uses all threads for speculative execution after block finalization. In contrast, SPECPOOL splits resources between pre-execution and validation. For example, in the 0.5E 0.5V configuration with 4 threads, only 2 threads are available for pre-execution, which becomes the throughput bottleneck. Allocating more threads to pre-execution, as in 0.75E 0.25V, solves this bottleneck and improves overall throughput.

These observations highlight the key trade-off in SPECPOOL: increasing validation threads reduces block validation latency, whereas prioritizing pre-execution threads increases the effective throughput across blocks.

Figure 3 shows results for the medium contention workload. Despite increased conflicts, SPECPOOL maintains a clear lead. Latency improvements plateau beyond 16 threads due to reduced parallelism. The 0.75E 0.25V configuration achieves 8,800 TPS with 32 threads, compared to Block-STM's 7,460 TPS. The advantage comes from conflict-aware scheduling and dynamic thread allocation.

The experiments in §V-A establish baseline performance under conditions where speculative transaction order perfectly matches the final block order. The next section focuses on mempool dynamics and block inclusion behavior, providing empirical evidence that speculative pre-execution is feasible even in a low-latency, high-throughput blockchain system.

B. Mempool Visibility and Transaction Inclusion

The effectiveness of speculative pre-execution depends on transactions being visible in the mempool before block inclusion. In particular, MEV transactions may be inserted opportunistically or withheld from the public mempool, potentially

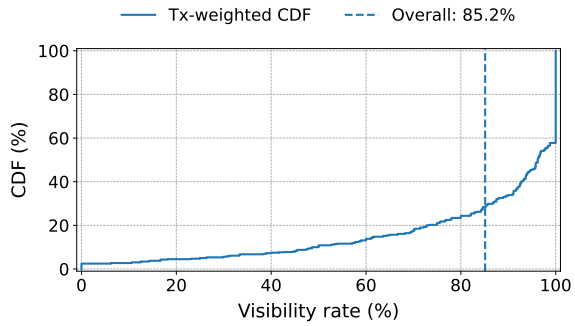

Fig. 4. CDF of per-block transaction visibility.

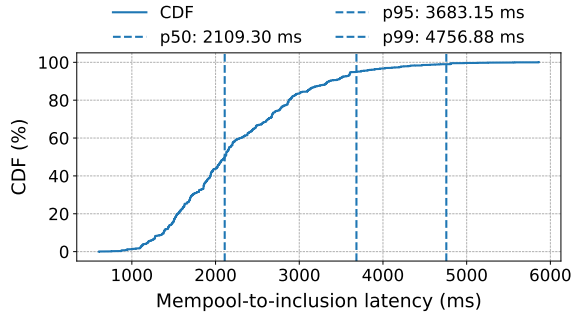

Fig. 5. CDF of mempool observation to block inclusion latency.

invalidating speculative results for subsequent transactions. We therefore quantify (i) *mempool visibility* and (ii) *mempool-to-inclusion latency* to assess whether SPECPOOL has a practical window to pre-execute transactions.

a) Measurement setup: We collected mempool snapshots and finalized blocks from live Injective nodes from 2026-01-06 00:00 UTC to 2026-01-07 00:00 UTC. Across this interval, we observed approximately 2.5M transactions.

b) Visibility: Figure 4 plots the CDF of per-block visibility rates. Overall, 85.2% of included transactions are observed in our mempool trace prior to inclusion, indicating that the majority of transactions are available for pre-execution.

c) Inclusion latency: Figure 5 shows the CDF of the time from first observation in the mempool to block inclusion. The median mempool-to-inclusion latency is 2109 ms, with p95 and p99 latencies of 3683 ms and 4757 ms, respectively. These latencies imply a substantial pre-execution window: given Injective's average block time of ≈0.63 s, a median latency of ≈2.1 s corresponds to roughly 3–4 blocks of lead time for SPECPOOL to pre-execute transactions.

While MEV-related behavior can reduce public visibility for a subset of transactions, the measured visibility and latency distributions suggest that speculative pre-execution remains effective for most blocks under realistic conditions, consistent with prior observations in public mempools [19].

C. Robustness to Unseen Transactions

Our mempool measurements (§V-B) indicate that transaction visibility is high but not complete, and that a fraction of transactions may bypass the public mempool (e.g., via private channels/MEV). We therefore evaluate robustness to *unseen* transactions using the low-contention workload with a fixed block size of 200 transactions and 8 total worker threads. We vary the unseen fraction p and, for each block, mark $p\%$ of transactions as unavailable to pre-execution by discarding their speculative outputs; these transactions (and any subsequently conflicting transactions) must then be re-executed during validation. Figure 6 reports validation latency as a function of p. As p increases, SPECPOOL's latency rises due to more re-execution, but remains below Block-STM up to approximately 85% unseen (for the 0.5E-0.5V balanced configuration). This robustness stems from SPECPOOL's conflict-aware scheduler, which leverages historical access patterns to place even unseen transactions into effective parallel validation groups, limiting cascading re-executions despite reduced mempool visibility.

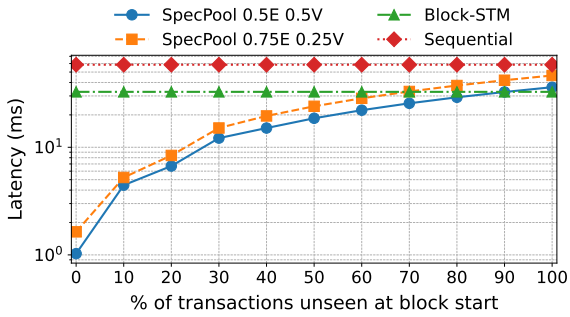

Fig. 6. Validation latency vs. fraction of transactions treated as unseen.

D. Block Order Prediction Accuracy

A key enabler of speculative pre-execution is anticipating the relative execution order of transactions, especially those that interact (e.g., target the same contract). SPECPOOL does so via its *block order prediction* procedure (§III-A1), which uses mempool-visible signals (fees, arrival time, and coarse batching windows) to produce a likely partial order; in our deployment we use $W = 0.05$s and $\Delta = 10{,}000$ (fee units).

We quantify prediction using *pairwise accuracy*: for a contract with transactions $T = \{t_1, \ldots, t_n\}$, consider all $\binom{n}{2}$ distinct pairs (t_i, t_j). Pairwise accuracy is the fraction of pairs where predicted order matches the finalized block:

$$\text{Pairwise Accuracy} = \frac{1}{\binom{n}{2}} \sum_{(i,j),i\neq j} \mathbf{1}[\text{pred}(t_i, t_j) = \text{actual}(t_i, t_j)].$$

Weighted global accuracy is computed by weighting each contract's accuracy by its number of transactions.

To evaluate the effectiveness of this approach, we use the same Injective dataset as in §V-B and compare the predicted order of transactions against the actual execution order finalized at consensus. Figure 7 visualizes the prediction accuracy for all contracts in the trace. Each point represents a single contract, with the x-axis showing the number of

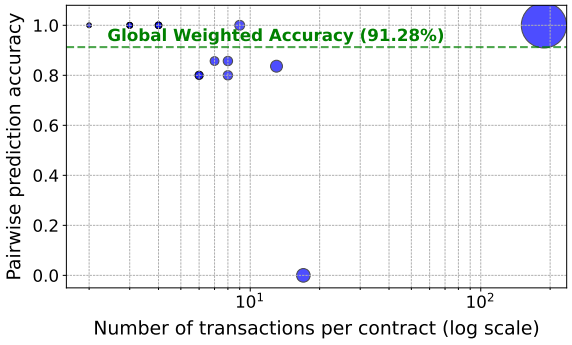

Fig. 7. Intra-contract block order prediction accuracy.

transactions and the y-axis showing the fraction of intra-contract transactions for which the predicted order matched the consensus order. Point size is proportional to the number of transactions for the corresponding smart contract.

Our analysis shows that the majority of contracts exhibit extremely high predictability. Many contracts, including those with dozens or even hundreds of transactions, have intra-contract ordering accurately predicted in nearly 100% of cases. A few contracts show lower accuracy due to irregular or adversarial transaction sequences. Overall, we observe a *weighted global accuracy* of 91.3%.

These results confirm that mempool information provides strong signals for ordering transactions with potential dependencies. By reliably predicting intra-contract ordering, SPECPOOL can pre-execute dependent transactions with minimal risk of validation conflicts. This high predictability directly translates into reduced re-execution overhead and improved parallelism during block validation.

E. Transaction Arrival Order Analysis

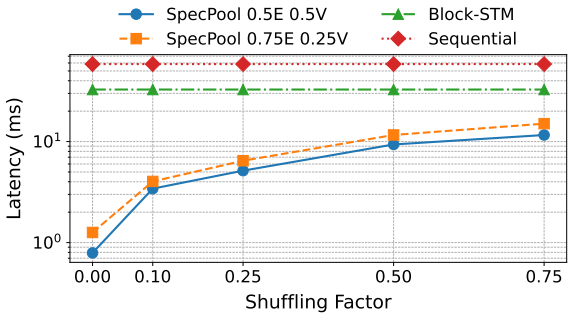

Fig. 8. Latency vs Transaction Order Shuffling.

We study the impact of misspeculations caused by divergences between pre-executed and final consensus orders. To simulate this, we introduce probabilistic local shuffling: iterating through the transaction list, at each index i we flip a biased coin, and if the outcome is below a user-defined *shuffling factor*, we swap transaction i with $i+1$. Higher shuffling factors increase disorder. This experiment uses a medium contention workload, with 4 threads and blocks of 200 transactions, isolating the shuffling factor as the sole variable.

Figure 8 shows block validation latency. As shuffling increases, more transactions are pre-executed out of order, causing validation failures and re-executions, which increase latency. Pre-execution remains more costly than validation, so throughput behavior is similar to previous experiments.

For instance, with no shuffling, SPECPOOL 0.5E 0.5V achieves 0.79 ms, rising to 11.61 ms at a shuffling factor of 0.75. SPECPOOL 0.75E 0.25V increases from 1.26 ms to 15.04 ms. Cosmos and Block-STM show constant latency (32.7 ms and 58.5 ms) independent of shuffling. Even at high disorder, SPECPOOL maintains 3× lower latency.

F. Block Size Analysis

We evaluate performance under varying block sizes to study how latency and throughput scale for the medium contention workload, shown in Figure 9. We fix the thread count to 4 and assume no shuffling (mempool order matches consensus) to isolate block size effects. Similar trends were observed for other configurations but are omitted for brevity.

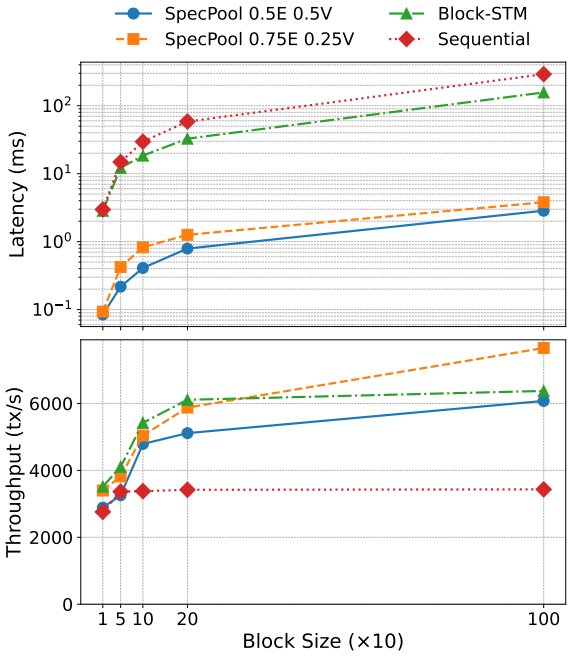

Fig. 9. Block size vs Latency (top) and Throughput (bottom).

As expected, larger blocks incur higher latency due to more transactions. SPECPOOL 0.5E 0.5V reduces block validation latency by roughly 33×, 41×, and 55× compared to Block-STM across increasing block sizes. Regarding throughput, Block-STM outperforms SPECPOOL 0.5E 0.5V across all block sizes. This is because, with a low thread count, it uses all four threads during speculative execution post-consensus, whereas SPECPOOL 0.5E 0.5V uses only two threads for pre-execution. However, Block-STM degrades as block size grows due to more concurrent conflicting transactions, causing costly re-executions and higher latency.

VI. Conclusion

This paper introduces SpecPool, a storage-layer speculative execution framework that uses mempool visibility to pre-execute transactions before consensus and reduce block validation latency. Measurements on Injective show that about 85% of transactions are visible before inclusion and remain in the mempool long enough to enable useful pre-execution, even with 0.63 s block times. Across Injective-derived workloads, SpecPool achieves up to 60× lower validation latency and up to 20% higher throughput than state-of-the-art speculative execution, while requiring no changes to consensus or VM semantics and easing practical adoption.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT), under projects UID/50021/2025, UID/PRR/50021/2025, and project no. 16539 (DOIs: 10.54499/UID/50021/2025 and 10.54499/UID/PRR/50021/2025). This work was also co-funded by the European Union through the Lisboa 2030 Programme (ERDF).

References

[1] V. Buterin, "A next-generation smart contract and decentralized application platform," 2014. [Online]. Available: https://ethereum.org/en/whitepaper/

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008, accessed: 2025-05-14. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] CosmWasm Team, "Cosmwasm documentation," accessed: 2025-05-14. [Online]. Available: https://docs.cosmwasm.com/

[4] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 35–48.

[5] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus," in *Proc. 17th European Conference on Computer Systems*, ser. EuroSys '22. ACM, 2022, pp. 34–50. [Online]. Available: https://doi.org/10.1145/3492321.3519594

[6] J. Kwon and E. Buchman, "Cosmos whitepaper," 2016, accessed: 2025-05-14. [Online]. Available: https://github.com/cosmos/cosmos/blob/master/WHITEPAPER.md

[7] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proc. ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. ACM, 2017, pp. 303–312. [Online]. Available: https://doi.org/10.1145/3087801.3087835

[8] 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 *Proc. 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. ACM, 2023, pp. 232–244. [Online]. Available: https://doi.org/10.1145/3572848.3577524

[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*. Springer, 2021, pp. 341–357.

[10] Z. Chen, T. Yang, Y. Zheng, Z. Zhang, C. Jin, and A. Zhou, "Spectrum: Speedy and strictly-deterministic smart contract transactions for blockchain ledgers," *Proc. VLDB Endowment*, vol. 17, no. 10, pp. 2541–2554, 2024.

[11] Z. Lai, C. Liu, and E. Lo, "When private blockchain meets deterministic database," *Proc. ACM on Management of Data*, vol. 1, no. 1, pp. 1–28, 2023.

[12] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 11, pp. 5070–5083, 2021.

[13] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[14] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, "On speculative replication of transactional systems," *Journal of Computer and System Sciences*, vol. 80, no. 1, pp. 257–276, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002200001300144X

[15] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, pp. 71–98, 2003.

[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[17] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[18] 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 *Proc. 13th EuroSys conference*, ser. EuroSys '18. ACM, 2018, pp. 1–15. [Online]. Available: https://doi.org/10.1145/3190508.3190538

[19] Y. Chen, Z. Guo, R. Li, S. Chen, L. Zhou, Y. Zhou, and X. Zhang, "Forerunner: Constraint-based speculative transaction execution for ethereum," in *Proc. ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 570–587.

[20] C. Goes, "The Interblockchain Communication Protocol: An overview," Jun. 2020. [Online]. Available: https://arxiv.org/abs/2006.15918

[21] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 83–92.

[22] P. Garamvölgyi, Y. Liu, D. Zhou, F. Long, and M. Wu, "Utilizing parallelism in smart contracts on decentralized blockchains by taming application-inherent conflicts," in *Proc. 44th International Conference on Software Engineering*, ser. ICSE '22. ACM, 2022, pp. 2315–2326. [Online]. Available: https://doi.org/10.1145/3510003.3510086

[23] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao, "Scaling nakamoto consensus to thousands of transactions per second," *arXiv preprint arXiv:1805.03870*, 2018.

[24] Y. Li, H. Liu, Y. Chen, J. Gao, Z. Wu, Z. Guan, and Z. Chen, "FASTBLOCK: Accelerating blockchains via hardware transactional memory," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 250–260.

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

[26] P. Singh Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "OptSmart: A space efficient optimistic concurrent execution of smart contracts," *Distributed and Parallel Databases*, vol. 42, pp. 245–297, Jun. 2024.

[27] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-Verify replication for Multi-Core servers," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 237–250.

[28] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: replication at the speed of multi-core," in *Proc. 9th European Conference on Computer Systems*, ser. EuroSys '14. ACM, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592800

[29] M. M. Saad, M. J. Kishi, S. Jing, S. Hans, and R. Palmieri, "Processing transactions in a predefined order," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 120–132.

[30] G. Ramseyer, A. Goel, and D. Mazières, "{SPEEDEX}: A scalable, parallelizable, and economically efficient decentralized {EXchange}," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 849–875.

[31] Y. Fang, Z. Zhou, S. Dai, J. Yang, H. Zhang, and Y. Lu, "Pavm: A parallel virtual machine for smart contract execution and validation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 1, pp. 186–202, 2023.