Spectrum: Speedy and Strictly-Deterministic Smart Contract Transactions for Blockchain Ledgers

Zhihao Chen
East China Normal University†
chenzh@stu.ecnu.edu.cn

Tianji Yang
East China Normal University†
tjyang@stu.ecnu.edu.cn

Yixiao Zheng
East China Normal University†
yxzheng@stu.ecnu.edu.cn

Zhao Zhang*
East China Normal University†
zhzhang@dase.ecnu.edu.cn

Cheqing Jin
East China Normal University†
cqjin@dase.ecnu.edu.cn

Aoying Zhou
East China Normal University†
ayzhou@dase.ecnu.edu.cn

ABSTRACT

Today, blockchain ledgers utilize concurrent deterministic execution schemes to scale up. However, ordering fairness is not preserved in these schemes: although they ensure all replicas achieve the same serial order, this order does not always align with the fair, consensus-established order when executing smart contracts with runtime-determined accesses. To preserve ordering fairness, an intuitive method is to concurrently execute transactions and re-execute any order-violating ones. This in turn increases unforeseen conflicts, leading to scaling bottlenecks caused by numerous costly aborts under contention. To address these issues, we propose **Spectrum**, a novel deterministic execution scheme for smart contract execution on blockchain ledgers. Spectrum preserves the consensus-established serial order (so-called strict determinism) with high performance. Specifically, we leverage a speculative deterministic concurrency control to execute transactions in speculation and enforce an agreed-upon serial order by aborting and re-executing any mis-speculated ones. To overcome the scaling bottleneck, we present two key optimizations based on speculative processing: operation-level rollback and predictive scheduling, for reducing both the overhead and the number of mis-speculations. We evaluate Spectrum by executing EVM-based smart contracts on popular benchmarks, showing that it realizes fair smart contract execution by preserving ordering fairness and outperforms competitive schemes in contended workloads by 1.4x to 4.1x.

PVLDB Reference Format:
Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. Spectrum: Speedy and Strictly-Deterministic Smart Contract Transactions for Blockchain Ledgers. PVLDB, 17(10): 2541 - 2554, 2024.
doi:10.14778/3675034.3675045

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jacklightChen/spectrum.

*Corresponding author.
†Engineering Research Center of Blockchain Data Management, Ministry of Education.
This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.
doi:10.14778/3675034.3675045

1 INTRODUCTION

Blockchain ledger systems [2, 3, 5, 10, 48] have emerged to facilitate collaborative business among multiple distrusting parties. These systems use state machine replication to replicate data in hostile environments. Transactions specified in user-defined smart contracts [52, 57] are ordered by the Byzantine consensus [13] and executed deterministically by all replicas for consistent state transitions.

In order to meet the determinism requirement, conventional approaches enforce each replica to follow sequential execution of the agreed-upon order but exhibit poor performance and limited multi-core scalability. To overcome this limitation, several works exploit transaction parallelism by proposing concurrent deterministic execution schemes based on various concurrency controls [9, 11, 15, 19, 20, 24, 29, 33, 44] and different execution paradigms [10, 40, 47, 49]. Among them, the recent integration of deterministic concurrency control (DCC) schemes, originally used in deterministic databases, into blockchain ledgers provides an ideal solution. It offers independent parallelism for replicas without changing the traditional blockchain order-execute paradigm.

However, merely ensuring determinism is not sufficient for blockchain ledgers, as it may vary from the consensus-established serial order. Unlike traditional databases [28, 51], where all replicas are controlled by the same party, different replicas in a blockchain ledger are controlled by multiple parties that are not trusted by each other. In blockchain ledgers, the specific total order of transactions can have significant financial implications for involved parties in order-sensitive applications (e.g., auctions and flash minting scenarios [37, 50]). Hence, replicas are motivated to prevent other parties from manipulating the ordering to their advantage. To address this concern, modern Byzantine consensus protocols [31, 32, 61] incorporate fairness designs to ensure the ordering is not manipulated by a single party (even if the consensus leader). Given that the ordering brings about fairness concerns, it is required for execution to strictly adhere to the fair ordering to preserve such fairness.

But most DCC schemes fail to meet this requirement. They merely guarantee deterministic serializability, which can lead to a deterministic yet different serial order that disrupts ordering fairness. Fig. 1 provides an example involving an order-sensitive application: the *AccessControl* smart contract with a *grantAccess* function, where a caller who has access permission can grant permission to a specified address. Consider three transactions in the agreed-upon fair order: $T_1 \rightarrow T_2 \rightarrow T_3$, where initially $addr_A$ has permission while others lack it. After serially executing the fair ordering, its semantics ensure that all four addresses $addr_{A\sim D}$ get

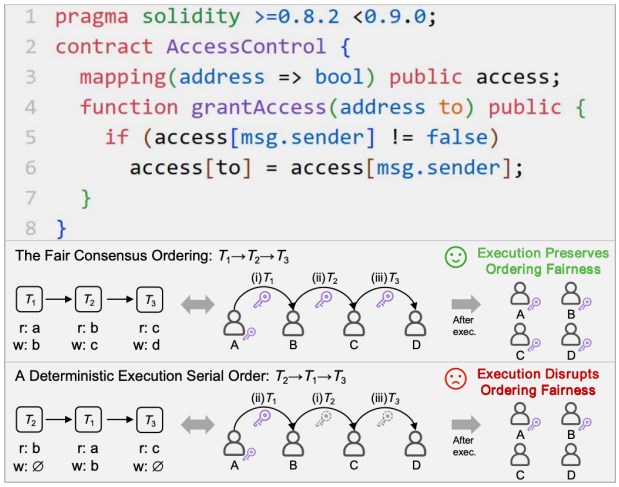


Figure 1: Merely Determinism Does NOT Preserve Ordering Fairness

permission. However, DCC schemes can produce another deterministic serial order due to deterministic reordering: $T_2 \rightarrow T_1 \rightarrow T_3$, resulting in only $addr_A$ and $addr_B$ getting permission, whose execution results destroy ordering fairness. Moreover, transactions invoking smart contracts may exhibit mutable read/write sets due to unpredictable, runtime-determined data dependencies. This introduces potential non-determinism issues within existing DCC schemes, resolving which violates the agreed-upon determinism. If an execution scheme fails to preserve ordering fairness, the fairness ensured by consensus becomes ineffective. More severely, it may allow adversaries to gain an unfair advantage in order-sensitive applications [18, 34, 37, 50], leading to inequitable outcomes.

For blockchain ledgers, the execution scheme prioritizes preserving ordering fairness, while also having performance concerns. To ensure the same serial order, a transaction must observe all modifications of preceding conflicting transactions. Sequential execution pessimistically holds this guarantee, but with poor efficiency. To enhance execution efficiency, a practical scheme is to concurrently execute transactions while aborting and re-executing order-violating ones. However, as conflicts are hard to pre-determine, this scheme faces multi-core scaling bottlenecks in contended workloads, typical of on-chain applications during demand spikes (e.g., flash minting/sale scenarios) [26]. In such scenarios, numerous and costly transaction aborts can occur due to unforeseen conflicts that violate the agreed-upon order, severely degrading overall performance. Hence, how to design an execution scheme that ensures both agreed-upon serial order and high performance remains an urgent issue to be solved for blockchain ledgers.

This paper addresses the following research questions: Can we design an execution scheme that preserves ordering fairness by ensuring strictly-deterministic serializability (SDS, as formally defined in §3.1)? And can this scheme simultaneously maintain high performance under conflicts? Our proposed execution scheme, Spectrum, addresses both questions concretely. Spectrum is a concurrent and strictly-deterministic execution scheme for blockchain ledgers, designed to preserve consensus-established ordering fairness with high performance. The main technical challenges that Spectrum addresses are as follows: (1) how to achieve SDS in the concurrent execution of smart contract transactions; (2) how to minimize re-execution overhead caused by transaction aborts, especially in the complex stack-based execution runtime; (3) how to integrate applicable and efficient scheduling techniques to reduce conflicts without requiring complete a-priori information.

Spectrum addresses the first challenge by leveraging a speculative deterministic concurrency control that does not require complete a-priori knowledge of transactions. With respect to an agreed-upon ordering, Spectrum runs transactions out-of-orderly, while aborting and re-executing ones contradicting that order during runtime. This implies the transaction commit order is always equivalent to the agreed-upon order. We prove that this speculative processing ensures SDS even for smart contract transactions with mutable read/write sets.

Then, Spectrum introduces an innovative partial rollback mechanism to tackle the second one. The main intuition is that when a transaction is aborted due to mis-speculations, instead of complete rollback and re-execution, specific executed logics unrelated to the conflicts can be preserved and reused through re-execution. This implies partial dependencies can be retained to quickly align with the pre-ordering. Spectrum proposes a novel persistent structure to realize this mechanism in the stack-based EVM [57] execution runtime, enabling operation-level transaction re-execution.

For the last challenge, Spectrum proposes a predictive transaction scheduling method that uses merely partial pre-acquired transaction knowledge to schedule potential conflicting transactions at runtime. If the prediction works, it helps reduce the number of mis-speculations and their induced aborts, thus maintaining superior performance under contention.

To the best of our knowledge, Spectrum is the first DCC scheme to ensure both strict determinism and high performance across diverse workloads. Leveraging Spectrum empowers blockchain ledgers to realize fair smart contract execution and support various throughput-centric scenarios.

To summarize, we present the following main contributions:

- We propose Spectrum, a deterministic execution scheme that achieves both strict determinism and high performance. It leverages speculative processing to realize fair and speedy smart contract execution for blockchain ledgers and addresses the scaling bottleneck with two key optimizations, enabling effective scaling under diverse workloads.
- We propose a partial rollback mechanism and implement an efficient, operation-level rollback approach in stack-based EVM smart contract execution, significantly reducing the overhead per mis-speculation.
- We design a predictive transaction scheduling method that uses merely partial a-priori transaction knowledge to pre-schedule potential conflicting transactions, effectively minimizing the number of mis-speculations.
- We evaluate Spectrum by running Ethereum-style smart contracts on popular benchmarks. The experimental results on YCSB, SmallBank and TPC-C show that Spectrum outperforms competitive schemes by 1.4x to 4.1x.

The rest of the paper is organized as follows. We present the background in §2, problem statement in §3, Spectrum design in §4, theoretical analysis in §5, implementation in §6, experimental evaluation in §7, related work in §8, and conclusion in §9.

2 BACKGROUND

2.1 Parallelizing Transactional Smart Contracts

Blockchain ledgers implement smart contracts as self-executing programs invoked by users as transactions. The current de facto standard implementation is Ethereum-style smart contracts. Their operations are specified in bytecode and run on a stack-based, 256-bit word-size Ethereum Virtual Machine (EVM). EVM currently supports over 100 types of operations to facilitate rich semantic applications. It executes operations to manipulate the stack and trigger on-chain state transitions. Notably, SLOAD/SSTORE operations are used to read/write smart contract state, where each state is mapped to a key/value pair and managed by state storage [59]. Ethereum-style smart contracts written in Solidity [7] are quasi-Turing-complete [57]. This implies that their access patterns, i.e., read/write sets (keys), are determined at runtime, making it difficult to predict accurate ones before execution.

Instead of having all replicas execute smart contract transactions sequentially for determinism, existing works attempt to exploit transaction parallelism while still ensuring determinism. These works can be broadly classified into two categories: *primary-follower parallelism* and *equally-replicated parallelism*.

Primary-Follower parallelism. In the order-execute (OE) paradigm, several works [11, 19, 29] explore transaction parallelism through differential execution designs on the primary and its followers. Given that transactions could be first packed and executed by the primary replica (i.e., the consensus leader) and then replayed and validated by other replicas, they propose two-phase concurrent deterministic execution schemes: the primary executes smart contracts concurrently using optimistic concurrency control (OCC) and records scheduling information in the block structure, facilitating the followers to replay consistent results with high concurrency. But these primary-follower approaches are vulnerable and only offer blocked parallelism. A malicious primary can inhibit the execution efficiency of followers with misleading schedule information. Additionally, followers must wait for the primary to finish its execution before replaying, resulting in low and blocked parallelism.

Equally-Replicated parallelism. Unlike primary-follower parallelism, equally-replicated parallelism makes no distinction between primary replicas and followers. All replicas are treated identically and utilize the same execution scheme to process ordered smart contracts, thus providing non-blocked parallelism. Under the OE paradigm, various works try to modify and integrate concurrency control schemes used in databases into blockchain ledgers. Nathan et al. [39] and OCC-DA [24] each incorporate deterministic abort rules to non-deterministic concurrency controls, specifically serializable snapshot isolation (SSI) and OCC, to ensure deterministic execution. However, although they eliminate non-determinism, the resulting deterministic serial order can differ from sequential execution of the pre-determined ordering in cases of conflicts.

More recently, several works have achieved transaction parallelism by assuming that complete read/write sets of smart contracts can be determined prior to execution using static analysis [6] or simulated execution. ParBlockchain [9] leverages this pre-acquired information to identify conflicts among transactions. It designs the OXII protocol, which constructs conflict graphs for each replica to execute non-conflicting transactions in parallel. PEEP [15] first incorporates deterministic concurrency control from the deterministic database Calvin [55] into blockchain ledgers. Based on the prior assumption, PEEP enables each replica to achieve concurrent smart contract execution with a determinism guarantee through an ordered lock scheme. However, these works heavily rely on the strong prior assumption. For complicated smart contract transactions (e.g., those with data dependencies, branchings, or delegate calls), accurately acquiring this information through static analysis or simulation is not feasible, making these schemes less practical. To address such cases, Harmony [33] builds upon the recent state-of-the-art deterministic database, Aria [38], which does not require the early acquisition of read/write sets. However, the adoption of these DCC schemes cannot maintain both strict determinism and high performance for blockchain ledgers.

Beyond the OE paradigm, Hyperledger Fabric [10] and its variants [25, 47, 49] introduce parallelism with an execute-order-validate (EOV) paradigm. NeuChain [40] makes ordering implicit through deterministic execution based on an execute-validate (EV) paradigm. But these schemes require a redesign of the execution paradigm, making them incompatible with the widely-used OE paradigm.

Table 1 tabulates a comparison of Spectrum with current deterministic execution schemes.

Table 1: Comparison with deterministic execution schemes

Deterministic Execution Schemes	Execution Paradigm	No Complete R/W Sets	Strict Determinism	Scaling in Contention
▲ HyperLedger Fabric [10]	EOV	✓	✗	✗
▲ Fabric(++, #) [47, 49]	EOV	✓	✗	✓
▲ NeuChain [40]	EV	✓	✗	✗
● Calvin [55], PWV [22]	OE	✗	✗	✓
● Bohm [23], Caracal [45]	OE	✗	✗	✓
● QueCC [43]	OE	✗	✗	✓
● Aria, AriaFB [38]	OE	✓	✗	✓
● Sparkle [36]	OE	✓	✓	✗
▲ Primary-Follower [29]	OE	✓	✗	✓
▲ SSI [39], OCC-DA [24]	OE	✓	✗	✗
▲ OXII [9], PEEP [15]	OE	✗	✗	✓
▲ Harmony [33]	OE	✓	✗	✓
▲ **Spectrum (Ours)**	OE	✓	✓	✓

● Scope of Databases ▲ Scope of Blockchains

3 PROBLEM STATEMENT

3.1 Violation of the Agreed-Upon Serial Order

Since the ordering established by consensus upholds fairness for all replicas, the execution scheme must preserve ordering fairness by ensuring the same serial order. We first provide a formal definition of strictly-deterministic serializability (SDS) in Definition 3.1. Briefly, for a specific agreed-upon ordering of transactions, an execution scheme holding this property ensures that its execution effects remain consistent with those of sequential execution. In the context of blockchain ledgers, if an execution scheme ensures SDS, it guarantees that the execution results maintain ordering fairness.

Definition 3.1 (Strictly-Deterministic Serializability, SDS). Given an agreed ordering of transactions, O:$\langle T_1, \ldots, T_n \rangle$, an execution schedule of transactions S satisfies **strictly-deterministic serializability** iff its effect is equivalent to the sequential execution of O, which adheres to the transactions commit order, $\langle T_1, \ldots, T_n \rangle$.

```
pragma solidity >=0.8.2 <0.9.0;
contract MutableRW {
  mapping(uint256 => uint256) public store;
  function add(uint256 key0, uint256 key1) public {
    uint256 bal0 = store[key0];
    uint256 bal1 = store[key1];
    if (bal0 < bal1) store[key0] += bal1;
    else store[key1] += bal0;
  }
}
```

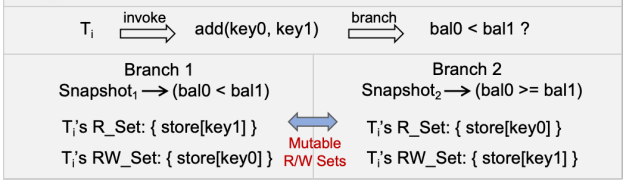


Figure 2: A Transaction T_i with Mutable Read/Write Sets

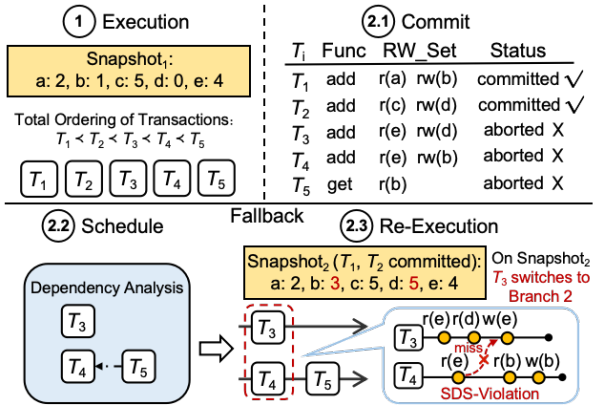


Figure 3: An Example of SDS-Violation in AriaFB Processing Transactions with Mutable Read/Write Sets

While most DCC schemes ensure a deterministic serial order, they do not guarantee that this order is equivalent to the agreed-upon order when processing smart contract transactions with runtime-determined accesses. The runtime-determined nature leads to mutable read/write sets. More generally, we define a transaction with mutable read/write sets as one whose read/write sets can vary across different snapshots due to runtime-determined logic, such as branching, dependent transactions, etc. Fig. 2 gives an example. In this figure, the *MutableRW* smart contract includes an *add* function with a conditional statement. When a transaction T_i invokes the add function, the read/write sets of T_i vary according to the snapshots on which T_i is executed, as shown at the bottom.

Early DCC schemes, such as Calvin [55] and its follow-ups [15, 22, 23, 43, 45], require knowing complete transaction read/write sets in advance to achieve deterministic scheduled execution. Even though these schemes can utilize the OLLP method [55] to pre-acquire complete one-shot read/write sets through simulated read-only execution on a snapshot before scheduling, the smart contract transaction can incur mutable read/write sets during runtime, leading to a mismatch with the pre-acquired ones. In such cases, Calvin merely aborts the transaction to avoid non-determinism, which entails a different deterministic serial order that contradicts the agreed-upon one, thus clearly violating SDS.

The recent state-of-the-art deterministic database Aria [38], proposes a DCC scheme without requiring a-priori knowledge. But Aria employs deterministic reordering techniques by transforming RAW dependencies to WAR ones, yielding a deterministic result that differs from that of the agreed-upon order, thus violating SDS. Further, even in the absence of reordering, Aria still fails to ensure strict determinism. We consider AriaFB as the variant of Aria without reordering, and exemplify its failed attempt to hold SDS.

Aria processes a batch of pre-ordered transactions in two phases: execution and commit. During the execution phase, all transactions are executed concurrently on a consistent snapshot, with their writes maintained locally. Once all transactions have been executed, the commit phase uses their execution results to abort conflicting transactions and commit the others. AriaFB offers an efficient fallback strategy to handle these aborted transactions at the end of the commit phase: the aborted ones are deterministically re-executed using Calvin's ordered lock scheme [55], based on their read/write sets obtained in the first phase. However, AriaFB still violates SDS when processing smart contract transactions with mutable read/write sets, as exemplified in Fig. 3. In the first phase, five transactions $\langle T_1, T_2, T_3, T_4, T_5 \rangle$ execute on $Snapshot_1$ resulting in T_1 and T_2 commit while T_3, T_4 and T_5 abort in the commit phase. Aborted transactions are then scheduled based on their obtained read/write sets, to re-execute in parallel. However, during re-execution, the read/write sets of T_3 change according to $Snapshot_2$, causing T_4 to miss T_3's write on e, thus violating SDS. Since the correctness of the fallback strategy relies on an accurate schedule, if the schedule mismatches the runtime situation due to mutable read/write sets, the mismatched transaction has to abort to prevent potential concurrency anomalies, resulting in SDS-violation.

To conclude, despite being the state-of-the-art DCC scheme, Aria and all its variants (including Aria-based Harmony [33]) fail to guarantee SDS, even when not employing the reordering optimization. In addition to the illustrated example, the rationale is that during two-phase processing of the Aria scheme, a transaction T_j can be committed before T_i's commit, where $T_i \prec T_j$ in the agreed-upon order. If T_i happens to have mutable read/write sets that conflict with those of T_j, it causes T_j to miss T_i's update, leading to a different deterministic serial order that violates the agreed-upon one. In such cases, the execution results of Aria lose expected ordering fairness ensured by fair consensus in blockchain ledgers.

3.2 Scaling Bottleneck under Contention

To guarantee SDS, an execution scheme should let a transaction T_i observe all its preceding conflicting transactions. Sequential execution meets this requirement by letting T_i wait for all its preceding transactions to commit before T_i can execute, but with poor efficiency. In the database scope, Sparkle [36] fulfills SDS by utilizing an optimistic approach with a speculative commit strategy. It executes transactions in parallel, optimistically committing subsequent transactions while repeatedly aborting and re-executing those transactions that are conflicting with the agreed-upon serial order during runtime. Applying this intuition helps ensure SDS in concurrent smart contract execution. However, while Sparkle guarantees SDS, its performance falls short of handling contended workloads, as its optimistic processing leads to two severe issues that cause significant performance degradation.

First, since the optimistic processing impose no constraints on the transaction processing order, the contended workloads increase the probability of conflicts occurring, leading to additional transaction rollback and re-execution overhead. This overhead becomes even more costly when processing smart contract transactions due to their complex execution runtimes, further impacting overall performance. Second, the increased contention in workloads brings a higher number of conflicts and aborts. Due to the side effects of optimism, certain transactions may be aborted repeatedly. And with an increasing number of threads, the number of aborts also increases sharply. This causes a considerable waste of CPU resources and restricts the scheme's multi-core scalability. Although Sparkle implements a preemptive locking method to reduce conflicts, this method introduces additional blocking and remains inefficient. We will further elaborate on these limitations in §4.3 and §4.4.

4 SPECTRUM DESIGN

4.1 Design Overview

The design goal of Spectrum is to ensure both strict determinism and high performance in concurrent smart contract execution. Currently, none of the existing DCC schemes can fulfill this goal. Also, we find it difficult to adapt the current state-of-the-art DCC scheme Aria for SDS guarantee, owing to its design limitations. Instead, Spectrum harnesses the power of speculation to ensure SDS for concurrent smart contract transactions (§4.2). Subsequently, it remains challenging to maintain high performance under contended workloads. To address this issue, Spectrum introduces two key optimizations based on speculative processing: (i) Spectrum proposes and implements an innovative partial transaction rollback mechanism to mitigate the high overhead per mis-speculation (§4.3). (ii) Spectrum designs a predictive transaction scheduling method that does not require complete a-priori knowledge, yet effectively reduces the number of mis-speculations under contention (§4.4).

4.2 Speculative Transaction Execution

Speculative processing. Spectrum incorporates a multi-versioned, speculative deterministic concurrency control that allows for out-of-order speculation. The intra-block transactions processed by Spectrum are pre-ordered through consensus, each assigned a globally unique and auto-incremented sequence number, which reflects the total order of all transactions. To achieve SDS, Spectrum ensures that the effect of concurrent execution is equivalent to the result of serial execution ordered by the sequence number. This is enforced by allowing a transaction T to finally commit *iff* all of its preceding transactions have finished their final commit and T captures all related modifications. Since transactions are executed in speculation, mis-speculations can cause conflicts that contradict the agreed-upon total order. Spectrum detects these conflicts at runtime, aborts the conflicting transactions, and then re-executes them with the same sequence number to ensure strict determinism across all replicas. To support this processing, Spectrum leverages a multi-versioned shared storage and maintains a version list for each state, wherein versions are arranged in ascending order of sequence numbers. Additionally, each version keeps a read dependency list to track transactions that have read it.

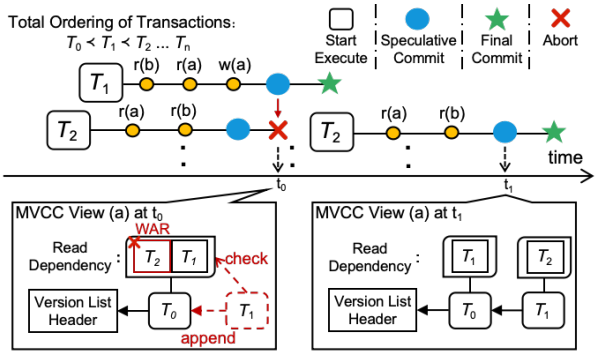


Figure 4: Speculative Transaction Execution

The lifetime of a transaction T includes the following phases: (1) Execution, (2) Speculative Commit, and (3) Final Commit.

(1) Execution: During its execution, the transaction T may execute multiple read and write operations on different states. For a read operation, T first attempts to read the state from its local writeset and readset. If the state is not found, T continues its read in the shared storage by scanning the state's version list and retrieving the version with the largest sequence number that is smaller than its own sequence number. Due to speculative processing, this retrieved version may be incorrect, as some other transactions that precede T could later yield more recent versions. Therefore, T needs to record its sequence number in the read dependency list of the accessed state version, which enables aborting T if a conflict is detected later. For a write operation, T merely inserts the update into its local writeset. Spectrum employs an early abort mechanism: Prior to performing any read or write operation, T checks whether it has been notified to abort. If so, T requires a rollback and re-execution with the same sequence number.

(2) Speculative Commit: Once T finishes its execution, it enters the speculative commit phase. In this phase, its local writes are made visible to other transactions: for each state within its writeset, T applies it to shared storage and checks if there are any transactions that might have missed reading this version. Specifically, T inserts a new state version into the state's version list at the position indicated by T's sequence number, and checks the read dependencies tracked by the largest preceding state version to abort transactions with larger sequence numbers, as they missed T's update on this state. After applying all states in T's writeset, T is considered to be speculatively committed.

(3) Final Commit: Following the speculative commit phase, T checks whether it could be finally committed. If so, it must satisfy both of the following conditions: (i) all transactions that precede T have already been finally committed, and (ii) T remains unaborted. If either of these conditions is not met, the check fails, and T would be suspended or aborted accordingly. Otherwise, T is considered to be finally committed. Since the writes of T have been applied in the (2) phase, the final commit merely increments the counter that tracks the sequence number of the latest final commit transaction. It also removes redundant write versions that precede T and cleans up T's inserted read dependencies, as they are no longer needed. Note that before T is finally committed, T remains possible to be aborted due to mis-speculations, and may undergo (1) and (2) phases several times until reaching the final commit, as illustrated in Fig. 4.

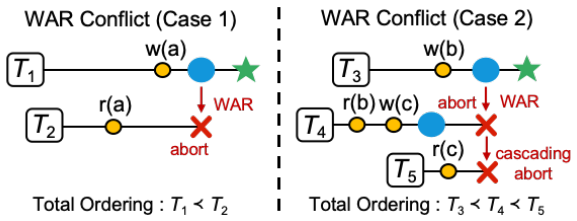


Figure 5: Exemplifying the WAR Conflict

Spectrum lets each thread independently execute the pre-ordered transactions in speculation, thus exhibiting high inter-thread concurrency. By leveraging speculative transaction execution, the resulting transaction commit order from concurrent execution is guaranteed to strictly conform to the agreed-upon serial order, thereby holding the SDS guarantee. In sharp contrast to Sparkle, Spectrum tracks read dependencies on each state version for accurate conflict detection, supports non-blocking writes, and utilizes proposed optimizations to maintain high performance under contention.

Conflicts in mis-speculations. As transactions are executed out-of-orderly during speculation, mis-speculations can occur when runtime results conflict with the agreed-upon serial order. In such cases, the conflicting transaction needs to be aborted, removing all its written versions (also triggering cascading aborts of transactions that read these versions) and related read dependencies, clearing the execution contexts, and subsequently re-executing with the same sequence number. We now introduce and discuss the types of conflicts. During speculation processing, only one type of conflict occurs due to mis-speculations: the *write-after-read* (WAR) conflict, as defined in Definition 4.1. Note that by adopting the multi-versioned design, multiple written versions can coexist during execution to prevent the *write-after-write* (WAW) conflict. Meanwhile, each transaction only reads the written version of preceding transactions, thus avoiding the *read-after-write* (RAW) conflict (i.e., T_i reads the version written by T_j, where $i < j$).

Definition 4.1 (Write-After-Read (WAR) Conflict). Transaction T_i incurs a WAR conflict with T_j if any T_i's write W_{T_i} (W_{T_i} is visible after T_i's speculative commit) is missed by T_j's read R_{T_j} where W_{T_i} conflicts with R_{T_j} and $i < j$.

Fig. 5 illustrates two cases of WAR conflicts. In Case 1, during T_2's execution, its read operation on state a missed T_1's write. Therefore, when T_1 reaches the speculative commit phase, it detects a WAR conflict with T_2 and aborts T_2. In Case 2, after T_4 has speculatively committed, T_5 reads T_4's write on state c. Later, T_3 encounters a WAR conflict with T_4 on state b, thus aborting T_4 and causing T_4 to cascade abort T_5.

4.3 Fine-Grained Transaction Rollback

Notion of partial transaction rollback. When a WAR conflict occurs, the aborted transaction triggers a revert process to rollback all its execution contexts and modifications. Traditionally, the rollback-to state is the initial state of the transaction, and subsequently, the transaction is reallocated to a free worker thread for re-executing all of its operations. Though this transaction-level revert obviously ensures correctness, it could be expensive and wasteful if a transaction faces conflicts that impact only a small portion of its execution, yet has to re-execute all of its operations. Furthermore, smart contract transactions are more complicated than simple transfer transactions, involving intricate business logic and complex stack-based execution runtimes. Consequently, the transaction-level revert not only leads to a significant waste of CPU resources but also imposes costly re-execution overhead.

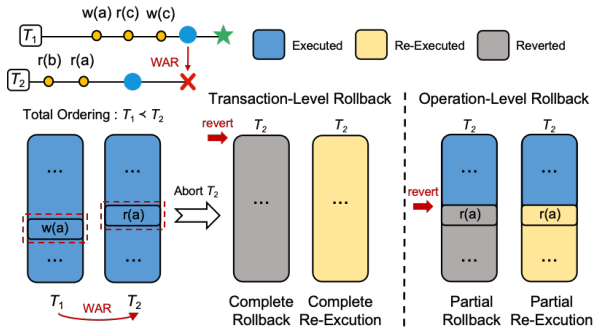


Figure 6: Operation-Level Transaction Rollback

To optimize the costly re-execution overhead, our approach allows aborted transactions to re-execute only the specific operations impacted by conflicts. Meanwhile, other executed operations that are unrelated to conflicts can be preserved and reused in subsequent re-execution, thus saving overhead. The insight is that the partial order implied by these non-conflicting operations conforms to the agreed-upon serial order, and preserving them helps quickly align with the established order, thereby achieving SDS. To achieve this, we propose and incorporate a partial rollback mechanism into speculative execution. Generally speaking, when a transaction T should abort due to a conflict, T only needs to rollback to a checkpointed state *right before* the conflict occurs, rather than rolling back to the initial state. During re-execution process, T can resume and continue its execution using the execution context recorded by the checkpoint, and perform read/write operations for new versions to proceed with its execution, as exemplified in Fig. 6.

Implementing partial rollback in stack-based EVM. EVM is currently the dominant standard, and most smart contract execution runtimes are EVM-compatible. Each smart contract transaction runs an EVM instance, which consists of a program counter (*PC*), a runtime *Stack*, and a temporary *Memory* table. EVM executes operations indicated by the PC to manipulate the runtime stack. When performing an operation-level rollback, the PC and the written Memory can be stored and recovered easily. However, recovering previous stack states is non-trivial.

Compared to converting stack-based operations into register-based representations [14], having stack-native support for partial rollback is a more general and ideal approach. But unlike the register-based VM that stores operands in registers and allows for easy tracking, the stack-based VM manages operands within a stack, without explicitly knowing the operand addresses. In order to support partial rollback, it is necessary to journal all potential stack states that might be reverted to in the future. One strawman approach is to take a complete snapshot of all the elements residing in the stack, copying and storing them when a checkpoint needs to be created. However, this approach is highly inefficient, incurring significant memory and maintenance costs, as validated in §7.3.

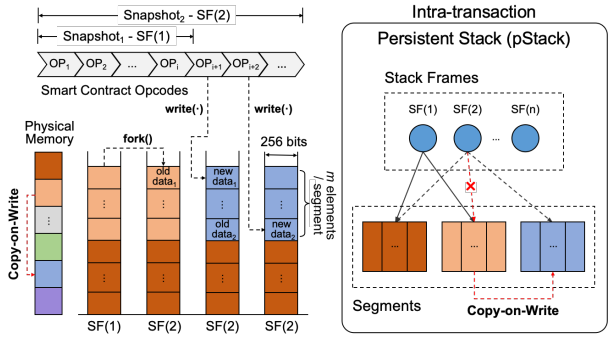


Figure 7: An Overview of pStack-Based Stack Frame Management

To overcome this limitation, we design *pStack*, a persistent data structure that efficiently restores and operates on multiple stack snapshots, as illustrated in Fig. 7. Conceptually, pStack functions like a standard stack but with enhanced capabilities. It can additionally access all elements by indexing while tracking previously recorded stack snapshots. The pStack partitions the stack into several memory segments, each tracking m consecutive stack elements. This enables efficient snapshotting by sharing parts of the stack state (unmodified segments) across multiple snapshots instead of copying the entire stack.

A snapshot of the stack state is indicated by a stack frame (SF). Each SF consists of three parts: a pointer list to memory segments, a stack height, and a list of boolean *ref-markers*. The ref-markers track whether current SF shares the segment with a previous SF, facilitating the *Copy-on-Write* (CoW) mechanism at the segment level. The pStack exposes three primitive methods:

- ***write(elementIndex, value)***: It updates current SF while preserving previous data. The SF first locates a target segment to be written with *elementIndex*. If its ref-marker is false, the *value* is directly written to the target segment. Otherwise, the operation clones that segment into a newly allocated memory segment, and writes *value* on the new memory segment. After that, the ref-marker is set to false, indicating that the new segment is solely owned by the current SF. Standard stack operations, such as push and pop can be implemented trivially using this method.
- ***fork()***: It creates a new stack frame SF as a snapshot of the current stack, and returns a unique *snapshotID*. When a new SF is created, it copies the stack height and pointers in the previous SF, and marks all segments with true ref-markers, meaning all segments are shared.
- ***rollback(snapshotID)***: It restores the stack state to the previous SF indicated by the *snapshotID*. This operation can be simply implemented as removing all SFs created after snapshotID, and deallocating segments in the SF if the corresponding ref-marker is false in that SF.

To implement partial rollback in the EVM, Spectrum replaces the runtime Stack with pStack and makes the Memory table multi-versioned. Then, we present how to integrate the partial rollback implementation with speculative execution in Algorithm 1. In general, we need to add checkpointing during execution, modify the speculative commit, and redesign the abort process to support operation-level rollback and re-execution.

Algorithm 1: Operation-Level Transaction Rollback

```
   /* Add-on EVM Primitives to Support Rollback                    */
1  function revertStateTo(EVM, RKey key):
2      PC, Memory, snapshotID=EVM.getCheckpoint(key);
3      EVM.setPCandMemory(PC, Memory);
4      EVM.pStack.rollback(snapshotID);
5  function makeCheckpoint(EVM, RKey key):
6      snapshotID=EVM.pStack.fork();
7      PC, Memory=EVM.recordPCandMemory();
8      EVM.addCheckpoint(key, PC, Memory, snapshotID)

   /* Transaction Primitives                                       */
9  function drainWrittenEntriesAfter(conflictKey);
10 function drainReadEntriesAfter(conflictKey);
11 function findEarliestConflictKey(keySet);

   /* Abort T with Received Conflict Keys.                         */
12 function localAbort(T):
13     keySet = atomicTake(T.conflictKeySet);
14     confictKey = T.findEarliestConflictKey(keySet);
15     revertStateTo(T.EVM, confictKey);
16     readEntries = T.drainReadEntriesAfter(conflictKey);
17     for ⟨key, value⟩ in readEntries do
18         MVCCTable.removeReadDependency(T.id, key);
19     writtenEntries =
         T.drainWrittenEntriesAfter(conflictKey);
20     for ⟨key, value, alreadyPublic⟩ in writtenEntries do
21         if alreadyPublic then
22             MVCCTable.removeVersion(T.id, key);

   /* Speculatively Commit a Transaction                           */
23 function speculativeCommit(T):
24     for ⟨key, value, alreadyPublic⟩ in T.writtenEntries do
25         if not alreadyPublic then
26             MVCCTable.insertVersion(T.id, key, value);
27             alreadyPublic = True;
```

Lines 1-11 describe the additional primitives for the EVM and the transaction. In Spectrum's design, the transaction makes a checkpoint before reading an external state. Since the EVM performs read through SLOAD operations, a transaction T calls the *makeCheckpoint(·)* primitive before executing any SLOAD operation satisfying that the state to be read is missed from its local read/write set. When a WAR conflict occurs, the conflict key (the key causing the abort) is recorded in the aborted transaction's *conflictKeySet*. If T detects to be aborted, it invokes the *localAbort(·)* primitive. T first atomically retrieves conflict keys and finds the latest checkpoint prior to any of these conflicts. Then, T invokes the *revertStateTo(·)* primitive to partially rollback to this checkpoint (lines 12-15). Afterwards, T cleans up the affected read dependencies and written versions (lines 16-22). Note that after the rollback, T can re-execute based on the execution context recorded by the checkpoint and proceed with recorded PC to perform new operations. Since the partial rollback has preserved unconflicted versions, the process of speculative commit merely needs to insert versions that are newly-written after the latest abort (lines 23-27).

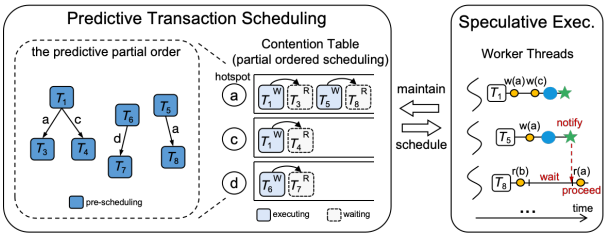


Figure 8: An Overview of Predictive Transaction Scheduling

4.4 Predictive Transaction Scheduling

The fine-grained rollback effectively reduces the overhead per mis-speculation. Nevertheless, under highly-contended workloads, the significantly increased number of mis-speculations cause numerous aborts, consuming a majority of CPU resources, while yielding rare commits. The main reason for such aborts stems from that the out-of-order speculative processing enforces no partial order of conflicting transactions due to assuming no prior knowledge, then the arbitrary scheduling leads to more frequent mis-speculations that limit the scheme's multi-core scalability under contention.

Sparkle proposes a preemptive locking method to partially reduce conflicts: before writing a state, a transaction T locks the state, preventing T's following transactions from accessing it until T finishes execution. However, this method fails to eliminate all cases of mis-speculations and also blocks write-after-write, resulting in significantly degraded execution parallelism.

Instead, Spectrum proposes a predictive scheduling method using pre-acquired information to effectively reduce mis-speculations. We observe that although obtaining accurate and complete a-priori read/write sets for smart contract transactions is not feasible, acquiring partial ones is indeed practical. This is because *some* states within the smart contract are accessed in a deterministic manner, i.e., their accesses are not changed by runtime-determined data dependencies. Therefore, the access keys can be determined and acquired before execution through static analysis or simulated execution. By leveraging this partial information, the predictive scheduling pre-schedules conflicting transactions to achieve superior multi-core scalability under contention, as outlined in Fig. 8. To clarify, our method assumes only partial prior knowledge, and even if this information is incorrect, it introduces only inefficiency but does not compromise the SDS guarantee of speculative execution.

Pre-Scheduling. The pre-scheduling takes partial pre-acquired read/write sets to schedule potentially conflicting transactions at runtime. For instance, if T_i is found to cause a WAR conflict with T_j on state a, the pre-scheduling blocks T_j's read on a until T_i reaches the final commit, thus pessimistically avoiding potential runtime WAR conflicts. Spectrum realizes partial ordered scheduling with a contention table (CT). For a transaction T, CT maintains T's pre-acquired knowledge to find WAR conflicts, and pre-schedules T in the topological order based on WAR conflicts in prediction. If the prediction is correct, this helps reduce runtime mis-speculations.

Algorithm. We apply the predictive scheduling only to highly-conflicted keys, which can be identified by setting an abort rate threshold. Algorithm 2 describes the detailed scheduling procedure. CT maintains an ordered list for each state to record read/write attempts. Before executing a transaction T, the worker thread inserts

Algorithm 2: Predictive Transaction Scheduling

```
   /* Runtime Pre-Scheduling                                    */
1  function schedule(T):
2      CT.insertInfo(T.preScheInfo);
3      wait until: last_scheduled + 1 == T.id;
4      last_scheduled = last_scheduled + 1;
5  function execution(T):
6      before read a scheduled key:
7      wait until: T.shouldWait(key) ≤ last_finalized; ...
8  function finalCommit(T):
9      last_finalized = last_finalized + 1;
10     CT.clear(T.preScheInfo); ...
```

T's pre-acquired read/write sets (only hotspot) into CT. During the inter-thread concurrent process, CT continually updates the *shouldWait* transaction on each T's read key. By ensuring CT manages all transactions preceding T, the worker thread could identify potential WAR conflicts related to T (lines 1-4). When executing T, if T attempts to read a scheduled key, it must wait until preceding writes on this key finalize (lines 5-7). When T is finally committed, it clears all its inserted attempts from CT and sets the last finalized transaction to itself. This implicitly notifies those waiting transactions (lines 8-10). Compared to Calvin's ordered lock pre-scheduling, our method schedules only WAR conflicts and partial transaction keys, leveraging concurrent maintenance and sequential granting to avoid using the limited single-threaded lock manager.

Independent predictive scheduling across replicas. The predictive scheduling is tailored for blockchain ledgers with multiple mutually-distrusting replicas. Each replica can independently generate a predictive schedule. Despite different predictions, speculative execution enforces the same agreed-upon serial order to maintain consistency across replicas. Blockchain ledgers commonly use Byzantine consensus protocols to negotiate the total order, assuming at most f malicious replicas out of n ($n \geq 3f + 1$). Even if f malicious replicas intentionally provide inefficient scheduling, the system performance is unaffected because the majority execute independently in speculation to obtain consistent results.

5 ANALYSIS AND DISCUSSIONS

SDS analysis. We present a theoretical analysis of the SDS guarantee of our proposed execution scheme.

Theorem 5.1. *The speculative transaction execution of an agreed-upon block of smart contract transactions enforces SDS.*

Proof. (By Contradiction.) The block B_h contains a set of pre-ordered transactions $\langle T_1, ..., T_n \rangle$ satisfying $T_1 \prec T_2 \prec \cdots \prec T_n$. The speculative transaction execution imposes a constraint on the commit order of transactions: transaction T_i is finally committed iff all its preceding transactions are finally committed. This implies that the execution finally committed by T_i is based on the snapshot taken after all preceding transactions $\langle T_1, \cdots, T_{i-1} \rangle$ were committed. Specifically, even if T_i has mutable read/write sets, its finally committed read/write sets are executed on the correct snapshot aligned with sequential semantics, thus ensuring SDS.

Assume there is an SDS schedule S that does not impose the above constraint on the transaction commit order. Then, in a possible schedule, $FC_{T_j} \prec FC_{T_i}$ and $i < j$ (we use FC_{T_k} to indicate T_k's final commit). If T_i WAR conflicts with T_j, the execution finally committed by T_j misses T_i's writes, however, sequential execution always commits T_i before T_j as $i < j$. Consequently, after executing B_h, the serial order of S is not equivalent to that of sequential execution, thus contradicting the SDS guarantee of S. □

Additionally, we discuss the phantom reads [21], which may occur in range scans. Notably, smart contract runtimes do not natively support range scans. Even if implemented manually (which is highly inefficient), Spectrum can capture WAR conflicts on the metadata (e.g., key set size) to prevent SDS-violation.

Analysis of partial rollback. The correctness of the partial rollback is evident, as it preserves unconflicted logics aligned with the total order and relies on speculative execution to finally keep consistent with the agreed-upon order. Its efficiency is tied to the expected operation savings in re-execution overhead, contingent upon the positions of mis-speculated read operations. We give a quantified efficiency analysis: Suppose we have a table with k keys. A smart contract transaction has n operations denoted as $T = \langle OP_i \mid 1 \le i \le n \rangle$. Among these, there are t read/write operations accessing unique keys, $\mathcal{S} = \langle OP_{rw(i)} \mid 1 \le i \le t, 1 \le rw(i) \le n \rangle$. And out of these t operations, r are read operations, denoted as $\mathcal{S}' = \langle OP_{rw(h_j)} \in \mathcal{S} \mid 1 \le j \le r, 1 \le h_j \le t \rangle$. We assume that key access follows a uniform distribution.

The probability of a WAR conflict is calculated as follows: Since each read operation involves a unique key, the probability of the conflict occurring in the transaction is $\frac{r}{k}$. By adopting the early abort mechanism, we assume that the conflict could be detected prior to the next read/write operation. Thus, the number of rollbacked operations due to conflict at position $rw(h_j)$ for complete rollback and partial rollback are respectively $rw(h_j{+}1)$ and $rw(h_j{+}1) - rw(h_j)$ (if $h_j = t, rw(h_j + 1) = n$). Consequently, the expected number of rollbacked operations for complete rollback is $E(OP_{c\_rollback}) = \frac{r}{k} \cdot \sum_{j=1}^{r}(\frac{1}{r} \cdot (rw(h_j + 1)))$, whereas for partial rollback, it is $E(OP_{p\_rollback}) = \frac{r}{k} \cdot \sum_{j=1}^{r}(\frac{1}{r} \cdot (rw(h_j{+}1) - rw(h_j)))$. The expected number of operations saved per conflict indicates CPU resource savings from re-execution, and can be represented as

$$\begin{aligned} E(OP_{saved}) &= E(OP_{c\_rollback}) - E(OP_{p\_rollback}) \\ &= \frac{1}{k} \sum_{j=1}^{r} (rw(h_j)) \end{aligned}$$

6 IMPLEMENTATION

To ensure a fair comparison, we implement Spectrum and all baselines in the same codebase following the C++20 standard. We incorporate evmone[1], a C++ implementation of EVM, for smart contract execution environment, and write all smart contracts in Solidity. We assign each transaction a globally unique 64-bit TID (Transaction ID) to indicate the agreed-upon total order of transactions. In a real deployment, transactions would be ordered and assigned TIDs by consensus protocols. Given that ordering is an orthogonal issue to the execution scheme, we simply assign TIDs when generating transactions. States are managed by an in-memory concurrent hash table, maintaining key-value pairs where each value represents either a single state version or a list of versions (if multi-version is adopted). In Spectrum, for each transaction, we maintain a pStack-integrated EVM, called EVMCoW, and empirically set the pStack segment size $m = 2$. For predictive scheduling, we let threads handling scheduled reads wait until they receive notifications.

[1] https://github.com/ethereum/evmone

7 EVALUATION

7.1 Experimental Setup

Testbed. Our experiments are conducted on a physical machine equipped with 2 Intel Xeon Gold 6330 CPU @ 2.00GHz processors. Each processor has 28 physical cores with 84MB L3 cache and resides within one NUMA zone, for a total of two NUMA zones. Each NUMA zone has 128 GB DRAM, for a total of 256GB DRAM. The machine runs Ubuntu 20.04 LTS with 5.15.0 Linux Kernel.

Baselines. We compare Spectrum with the following deterministic execution schemes. Among them, Calvin, Aria, and AriaFB fail to ensure SDS and are evaluated only for performance comparison.

Serial: Serial executes and commits transactions sequentially using one single worker thread.

Calvin: Calvin [55] uses an ordered lock scheme for deterministic scheduled execution. We implement this scheme with optimized concurrent maintenance and sequential granting features. We use simulated execution to obtain the pre-acquired read/write sets.

Aria (with reordering): Aria [38] uses a deterministic batch processing technique to ensure determinism without the need for pre-acquisition of read/write sets. Our implementation of Aria incorporates both reordering and fallback optimizations.

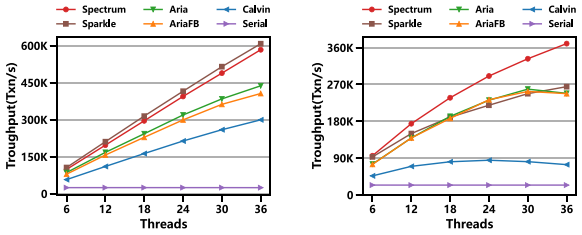
AriaFB (without reordering): AriaFB employs the fallback strategy but does not adopt reordering optimization. Note that AriaFB still fails to promise SDS when processing mutable read/write sets.

Sparkle: Sparkle [36] has two variants in its paper: single-node and distributed. We implement the single-node one (with early abort enabled) as this paper does not discuss distributed transactions.

Benchmarks. We select three popular benchmarks and implement them using smart contracts to conduct extensive experiments.

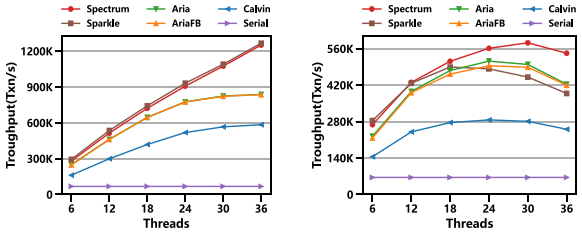
YCSB: The YCSB benchmark [17] is a benchmark suite designed to evaluate the performance of data management systems. We implement a YCSB-like smart contract with a global mapping to represent a key-value store, where both key and value are 256-bit integers. Our evaluation incorporates a transactional YCSB benchmark, where we group 10 read/write operations into a function to be invoked by a transaction. Each operation accesses a single unique key, and the read/write ratio is set to 50%:50%.

SmallBank: The Smallbank benchmark [8] simulates common operations in banking applications. We implement a SmallBank-like smart contract with global mappings to represent the savingStore and checkingStore, whose keys indicate accounts and the values record their balance, both are 256-bit integers. We realize six functions: getBalance(2r), depositChecking(1r1w), transactSaving(1r1w), writeCheck(2r1w), amalgamate(2r2w), and sendPayment(2r2w). These functions are invoked with equal probability.



(a) Throughput (Uniform) (b) Throughput (Skewed)

Figure 9: YCSB Throughput of Varying Threads



(a) Throughput (Uniform) (b) Throughput (Skewed)

Figure 10: SmallBank Throughput of Varying Threads

Note that the read/write sets of transactions that invoke sendPayment are mutable due to the presence of branching logic.

TPC-C: The TPC-C benchmark [1] models an industry-grade order processing application. We implement a TPC-C alike smart contract that supports the NewOrder, Payment, and Delivery transaction. These three transactions comprise 92% of the standard mix TPC-C workload. Since smart contract runtimes do not natively support range scans, we omit the other two transactions requiring range queries. In the experiment, for higher contention, we evaluate under a single-warehouse setting [22, 45] and invoke the NewOrder, Payment, and Delivery transactions in a ratio of 11:11:1.

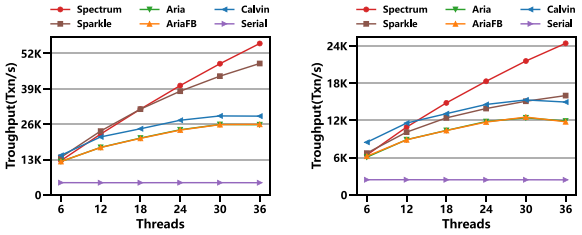
Workload. We generate two workloads for YCSB and SmallBank: uniform workload (**Uniform**) and skewed workload (**Skewed**). For the former, keys accessed by a transaction are uniformly chosen. For the latter, keys are selected via a Zipfian distribution, controlled by a *Zipf* parameter. A higher *Zipf* leads to more skewed and contended workload. For TPC-C, we vary the number of orderlines per order (each orderline incurs about 4 reads and 5 writes) to increase the contention degree. In all workloads, the default block/batch size is 100 and the keys are chosen from a set of 1 million keys.

7.2 Scheme Comparison

In this comparison, we evaluate Spectrum that uses solely operation-level rollback without applying predictive scheduling. Detailed evaluation of predictive scheduling is presented in §7.4 and §7.5.

Throughput. We compare the throughput by varying the number of worker threads under both uniform and skewed workloads.

YCSB Results: Fig. 9 shows YCSB results. Under the uniform workload shown in Fig. 9(a), both Spectrum (583 KTxn/s) and Sparkle (607 KTxn/s) scale well with 36 worker threads, and outperform that of Aria (437 KTxn/s), AriaFB (405 KTxn/s), Calvin (299 KTxn/s) and Serial (24 KTxn/s). This is because Spectrum and Sparkle realize superior inter-thread concurrency. Spectrum incurs



(a) Throughput (10 orderlines / order) (b) Throughput (20 orderlines / order)

Figure 11: TPC-C Throughput of Varying Threads

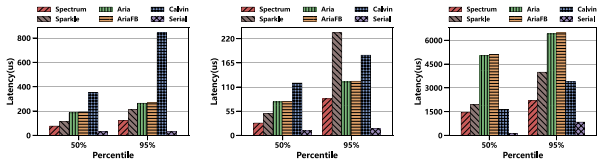
checkpointing overhead, which slightly lowers its throughput relative to Sparkle. In contrast, Aria and AriaFB have extra overhead of phase synchronization, and Calvin is constrained by the overhead of managing locks. Fig. 9(b) depicts the results under the skewed workload with *Zipf* = 0.9. At 36 threads, Spectrum achieves 1.4x, 1.5x, 1.5x, 5.0x and 15.4x higher throughput than Sparkle, Aria, AriaFB, Calvin and Serial, respectively. As the number of threads increases, Sparkle suffers from numerous costly aborts that limit its scalability, while Spectrum effectively reduces the overhead per abort with its fine-grained rollback mechanism, thus performing the best. Aria has similar performance to AriaFB since very few RAW dependencies could be transformed through the reordering optimization in the evaluation. Meanwhile, the weaker isolation requirement of Aria cannot compensate for the phase synchronization overhead, resulting in lower throughput than Spectrum.

SmallBank Results: Fig. 10 shows SmallBank results. Fig. 10(a) shows similar findings under the uniform workload: The throughput of Spectrum is comparable to Sparkle and surpasses Aria, AriaFB, Calvin and Serial by 1.5x, 1.5x, 2.1x and 18.7x respectively. Since SmallBank entails fewer read/write operations than YCSB, the increased proportion of phase synchronization overhead impacts the scalability of Aria and AriaFB, while Spectrum and Sparkle do not introduce such overhead. Fig. 10(b) chooses a higher *Zipf* = 1.1 and depicts the results. As the number of threads increases, the throughput of Sparkle reaches its peak (489 KTxn/s) at 18 threads and gradually decreases due to costly overhead of increased misspeculations. In contrast, Spectrum reduces the overhead per misspeculation, reaching a peak at 30 threads (582 KTxn/s). Overall, with 36 threads, Spectrum achieves 542 KTxn/s, outperforming Sparkle by 39%, Aria variants by 29%, and Calvin by 216%.

TPC-C Results: Fig. 11 reports TPC-C results. We choose 10 and 20 orderlines per order to evaluate longer transaction scenarios than YCSB/SmallBank. Fig. 11(a) and Fig. 11(b) show the corresponding results: Spectrum achieves the highest throughput within both experiments, benefiting from its high concurrency and partial rollback optimization. As the number of read/write operations per transaction increases, the advantage of partial rollback proposed by Spectrum becomes more evident. The throughput of Spectrum surges from 1.15x to 1.52x compared to Sparkle. Besides, Spectrum achieves up to 2.16x, 2.15x, and 1.92x higher throughput than Aria, AriaFB, and Calvin, respectively.

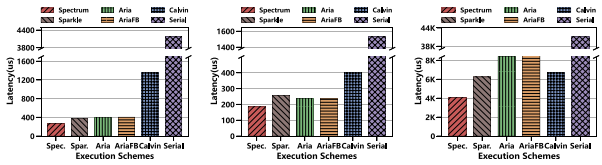
Transaction & block latency. We compare the p50 and p95 transaction latency, and block latency of all schemes.

YCSB Results: We evaluate YCSB transaction latency using 36 threads under the skewed workload with *Zipf* = 0.9. In Fig. 12(a),



(a) YCSB (b) SmallBank (c) TPC-C

Figure 12: Transaction Latency



(a) YCSB (b) SmallBank (c) TPC-C

Figure 13: Block Latency

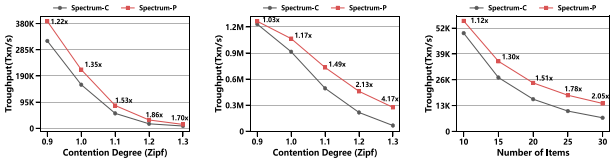
Serial (32 us ~ 34 us) achieves the lowest latency as it avoids coordination and aborts. Spectrum (76 us/p50, 123 us/p95) exhibits low re-execution overhead and thus much lower latency than Sparkle (112 us/p50, 211 us/p95). Aria variants have higher latency (184 us ~ 270 us) since commits within a batch must wait for the entire batch to complete. Moreover, Calvin incurs the highest latency due to locking overhead. Fig. 13(a) displays the average block latency: Spectrum has the lowest (270 us), followed by Sparkle (377 us), Aria (402 us), AriaFB (404 us), Calvin (1352 us), and Serial (4183 us).

SmallBank Results: The latency results (36 threads, *Zipf* = 1.1) of SmallBank are depicted in Fig. 12(b) and Fig. 13(b). Despite shorter transactions, Spectrum (83 us/p95) saves p95 latency by up to 64% than Sparkle (233 us/p95, whose p95 spikes are caused by repeated and costly aborts), 32% than Aria and AriaFB, and 54% than Calvin. Besides, as Spectrum achieves the highest throughput, it also reduces the block latency by 28.7% (Sparkle), 22.0% (Aria), 22.6% (AriaFB), 53.9% (Calvin) and 88.0% (Serial).

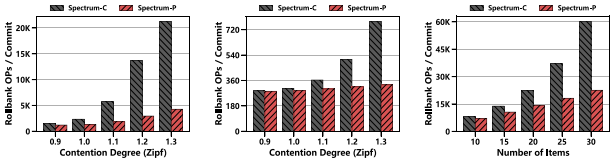
TPC-C Results: Fig. 12(c) shows the transaction latency results of TPC-C with 20 orderlines per order. Among concurrent schemes, Spectrum remains the lowest p50 (1351 us) and p95 (1876 us) latency, saving latency by 14.3% to 72.3% compared to others. Calvin avoids aborts by scheduling, where the benefits outweigh its overhead for longer transactions, resulting in lower latency than Sparkle, Aria and AriaFB, but still higher than Spectrum. Fig. 13(c) presents the block latency results. Compared to other concurrent schemes, Spectrum reduces the block latency by a large margin (39% ~ 53%) .

7.3 Evaluation of Partial Rollback

Reduction of re-execution overhead. We test the re-execution overhead savings through partial rollback (denoted as Spectrum-P) compared to complete rollback (denoted as Spectrum-C), measured by throughput and number of rollback operations per final commit transaction. Fig. 14 shows the results of all benchmarks using 36 threads with varying the contention degree. As depicted in Fig. 14(a), 14(b), and 14(c), the partial rollback mechanism improves the scheme throughput by up to 1.86x (YCSB), 4.17x (SmallBank), and 2.05x (TPC-C) compared to complete rollback. This improvement stems from that the partial rollback effectively reduces the number of operations needed to be rollbacked and re-executed. More

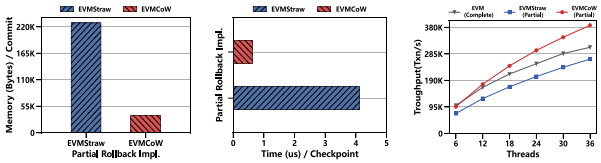


(a) YCSB Throughput (b) SmallBank Throughput (c) TPC-C Throughput



(d) YCSB Saved OPs (e) SmallBank Saved OPs (f) TPC-C Saved OPs

Figure 14: Reduction of Re-Execution Overhead



(a) Memory Cost (b) Checkpoint Time Cost (c) YCSB Throughput

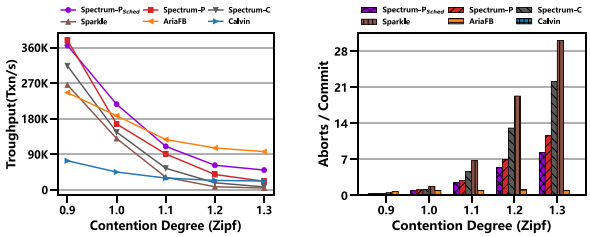
Figure 15: Detailed Evaluation of Partial Rollback

precisely, Spectrum-P saves up to 79.8% (YCSB), 57.6% (SmallBank) and 62.4% (TPC-C) rollback operations per committed transaction compared to Spectrum-C, as reported in Fig. 14(c), 14(d), and 14(e). Note that under high contention, the sharply increased number of aborts adversely affect overall throughput, as analyzed in §4.4.

Efficiency of pStack. We compare pStack with the strawman approach that copies all stack elements when checkpointing, measuring their memory cost, time cost, and throughput impact. Their integrated EVMs are EVMCoW and EVMStraw. Our evaluation uses YCSB with *Zipf* = 0.9 and 36 threads. Fig. 15(a) shows the memory cost of checkpoints created by each final commit transaction. By sharing memory of unmodified segments, EVMCoW incurs only 34 KB memory cost per commit, reducing the cost by 84.8% compared to EVMStraw (223 KB/Commit). Fig. 15(b) reports the average time cost of creating a checkpoint. EVMCoW is highly efficient in managing snapshots, requiring only 0.6 us to make a checkpoint, whereas EVMStraw consumes over 4 us. Fig. 15(c) presents a throughput comparison, including complete rollback for clearer contrast. EVMCoW efficiently implements checkpointing with the Copy-on-Write mechanism, its partial rollback exhibits the highest throughput, up to 25.5% higher than complete rollback. In contrast, EVMStraw incurs significant memory and maintenance costs, making it slow and cache-unfriendly, resulting in even 13.6% lower throughput than complete rollback, and 31.2% lower than EVMCoW.

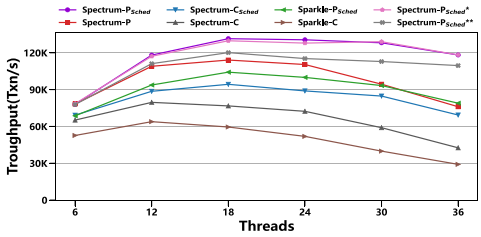
7.4 Evaluation of Predictive Scheduling

In the following evaluation, Spectrum-P$_{Sched}$ implements both predictive scheduling and partial rollback, Spectrum-C$_{Sched}$ and Spectrum-P respectively only apply the former and the latter, and Spectrum-C employs neither of these optimizations.



(a) Throughput of Varying Zipf (b) Aborts/Commit in Varying Zipf

Figure 16: Detailed Evaluation of Predictive Scheduling



(a) Throughput of Varying Threads

Figure 17: Ablation Studies on Integrated Optimizations

We compare scheme performance under various contended workloads. Fig. 16(a) shows 36 threads YCSB throughput in varying $Zipf$ (0.9 ~ 1.3). Spectrum-P$_{Sched}$ predictively schedules hotspot keys (top 10th to 20th, 7.2% of total accesses). With a higher $Zipf$, the throughput of all schemes decreases due to increased conflicts and degraded parallelism. At $Zipf$ = 0.9, Spectrum-P$_{Sched}$ underperforms Spectrum-P by 3.7% due to scheduling overhead outweighing benefits. As $Zipf$ increases, predictive scheduling shows clear benefits: from $Zipf$ 1.0 to 1.3, Spectrum-P$_{Sched}$ outperforms Spectrum-P by 1.3x, 1.2x, 1.6x, 2.3x and yields 1.5x, 2.0x, 3.5x, 6.8x higher throughput than Spectrum-C by further reducing mis-speculations and induced overhead. In contrast, AriaFB re-executes the aborted transaction at most once, outperforming Spectrum-P$_{Sched}$, but failing to ensure SDS. Despite Calvin avoiding aborts, it lags behind Spectrum-P$_{Sched}$ due to large locking overhead. Fig. 16(b) reports the number of aborts per final commit. By leveraging scheduling method, Spectrum-P$_{Sched}$ reduces the number of aborts by up to 28.2% and 62.3% compared to Spectrum-P and Spectrum-C.

7.5 Evaluation of Integration

For an in-depth analysis, we conduct ablation studies on Spectrum by integrating one optimization at a time. We also apply proposed optimizations to Sparkle for overall comparison. Fig. 17 reports the results on YCSB by setting $Zipf$ = 1.1 with varying the number of threads. Clearly, Spectrum-C and Sparkle-C scale poorly under contention due to a significant increase in costly aborts. On the contrary, Spectrum-P$_{Sched}$ scales the best within 36 threads, benefiting from both partial rollback and predictive scheduling. At 36 threads, Spectrum-P$_{Sched}$ achieves the highest throughput (118 KTxn/s), 1.6x than Spectrum-P, 2.8x than Spectrum-C, 1.6x than Sparkle-P$_{Sched}$ and 4.1x than Sparkle-C. More precisely, the improvement of using only pre-scheduling (Spectrum-C$_{Sched}$, 1.6x than Spectrum-C) is less than that of using only partial rollback (Spectrum-P, 1.8x than Spectrum-C), but leveraging both yields the best performance (Spectrum-P$_{Sched}$, 2.8x than Spectrum-C).

Then, we adjust the number of scheduled keys to evaluate the trade-offs between scheduling and abort overhead, identified as Spectrum-P$_{Sched}$* (top 10th to 40th, 13.2% of total accesses), and Spectrum-P$_{Sched}$** (top 5th to 10th, 7.6% of total accesses). Though Spectrum-P$_{Sched}$* further reduces aborts by scheduling more keys, the increased scheduling overhead causes throughput merely comparable to Spectrum-P$_{Sched}$. Besides, Spectrum-P$_{Sched}$** schedules hotter keys, notably increasing the number and overhead of blocked executions awaiting notifications, making its throughput 7.3% lower than Spectrum-P$_{Sched}$, yet still 44.1% higher than Spectrum-P.

8 RELATED WORK

Deterministic concurrency control. DCC schemes are able to concurrently execute transactions while ensuring a deterministic serial order. Their potential to optimize the overhead of replication protocols has gained great interest from both academia [22, 23, 30, 36, 42, 43, 45, 54, 55, 62] and industry [4] for over a past decade. Early works assume a-priori transaction knowledge, they are either single versioned, e.g., Calvin [55], PWV [22], and QueCC [43], or multi-versioned, e.g., Bohm [23], Orthrus [46], and Caracal [45]. More recently, Aria [38] and Sparkle [36] break through the pre-acquisition limitations. However, they fail to maintain both ordering fairness and high performance for blockchain ledgers. Furthermore, DCC schemes hold promise for simplifying the distributed commit protocol, with several works [36, 42, 55, 62] proposing related solutions tailored for multi-partition transactions.

Optimized on-chain transaction processing. Recent works aim to optimize on-chain transaction processing in terms of consensus, execution, and storage. [12, 27, 35, 58] propose performant Byzantine consensus protocols to scale ordering performance. Besides exploiting execution concurrency, [16, 53] leverage pipelined execution to improve performance. Regarding storage, [41, 56, 60] design optimized state stores to alleviate read/write amplifications.

9 CONCLUSION

We present Spectrum, a concurrent and strictly-deterministic execution scheme that preserves consensus-established ordering fairness with high performance for blockchain ledgers. Spectrum produces the same agreed-upon serial order through speculative transaction execution and introduces two novel optimizations to maintain high performance under contention. The operation-level rollback optimization reduces the overhead per mis-speculation with minimal costs. The predictive scheduling optimization incorporates partial ordered scheduling to further minimize the number of mis-speculations. Future work includes enhancing Spectrum for intra-transaction parallelism and data-partitioned sharding settings.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by National Key Research and Development Program of China (No. 2021YFB2700100), Shanghai "Science and Technology Innovation Action Plan" Project (No. 23511100700), and Program of Shanghai Academic/Technology Research Leader (No. 23XD1401100).

REFERENCES

[1] 2010. TPC Benchmark C. https://www.tpc.org/tpcc/
[2] 2024. AntChain. https://antchain.antgroup.com/
[3] 2024. Diem. https://www.diem.com/
[4] 2024. Fauna. https://fauna.com/
[5] 2024. Quorum. https://consensys.net/quorum/
[6] 2024. Slither. https://github.com/crytic/slither
[7] 2024. Solidity Programming Language. https://soliditylang.org
[8] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. IEEE, 576–585.
[9] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems. In *ICDCS*. IEEE, 1337–1347.
[10] Elli Androulaki, Artem Barger, Vita Bortnikov, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *EuroSys*. ACM, 30:1–30:15.
[11] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2022. OptSmart: a space efficient optimistic concurrent execution of smart contracts. *Distributed and Parallel Databases* (2022), 1–53.
[12] Yehonatan Buchnik and Roy Friedman. 2020. FireLedger: A High Throughput Blockchain Consensus Protocol. *Proc. VLDB Endow.* 13, 9 (2020), 1525–1539.
[13] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *OSDI*. USENIX Association, 173–186.
[14] Yang Chen, Zhongxin Guo, Runhuai Li, Shuo Chen, Lidong Zhou, Yajin Zhou, and Xian Zhang. 2021. Forerunner: Constraint-based Speculative Transaction Execution for Ethereum. In *SOSP*. ACM, 570–587.
[15] Zhihao Chen, Xiaodong Qi, Xiaofan Du, Zhao Zhang, and Cheqing Jin. 2021. PEEP: A Parallel Execution Engine for Permissioned Blockchain Systems. In *DASFAA (3) (Lecture Notes in Computer Science)*, Vol. 12683. Springer, 341–357.
[16] Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Xiaodong Qi, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2021. SChain: A Scalable Consortium Blockchain Exploiting Intra- and Inter-Block Concurrency. *Proc. VLDB Endow.* 14, 12 (2021), 2799–2802.
[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.
[18] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *SP*. IEEE, 910–927.
[19] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2020. Adding concurrency to smart contracts. *Distributed Computing* 33, 3 (2020), 209–225.
[20] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB - A Shared Database on Blockchains. *Proc. VLDB Endow.* 12, 11 (2019), 1597–1609.
[21] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.
[22] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (2017), 613–624.
[23] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201.
[24] Péter Garamvölgyi, Yuxi Liu, Dong Zhou, Fan Long, and Ming Wu. 2022. Utilizing Parallelism in Smart Contracts on Decentralized Blockchains by Taming Application-Inherent Conflicts. In *ICSE*. ACM, 2315–2326.
[25] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2019. FastFabric: Scaling Hyperledger Fabric to 20, 000 Transactions per Second. In *ICBC*. IEEE, 455–463.
[26] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. 2023. Diablo: A Benchmark Suite for Blockchains. In *EuroSys*. ACM, 540–556.
[27] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *ICDE*. IEEE, 1392–1403.
[28] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084.
[29] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. 2022. A High Performance Concurrency Protocol for Smart Contracts of Permissioned Blockchain. *IEEE Trans. Knowl. Data Eng.* 34, 11 (2022), 5070–5083.
[30] Manos Kapritsos, Yang Wang, Vivien Quéma, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*. USENIX Association, 237–250.
[31] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. 2023. Themis: Fast, Strong Order-Fairness in Byzantine Consensus. In *CCS*. ACM, 475–489.
[32] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-Fairness for Byzantine Consensus. In *CRYPTO (3) (Lecture Notes in Computer Science)*, Vol. 12172. Springer, 451–480.
[33] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When Private Blockchain Meets Deterministic Database. *Proc. ACM Manag. Data* 1, 1 (2023), 98:1–98:28.
[34] Michael Lewis. 2014. *Flash boys: a Wall Street revolt.* WW Norton & Company.
[35] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A Decentralized Blockchain with High Throughput and Fast Confirmation. In *USENIX Annual Technical Conference*. USENIX Association, 515–528.
[36] Zhongmiao Li, Paolo Romano, and Peter Van Roy. 2019. Sparkle: Speculative Deterministic Concurrency Control for Partially Replicated Transactional Stores. In *DSN*. IEEE, 164–175.
[37] Yushi Liu, Liwei Yuan, Zhihao Chen, Yekai Yu, Zhao Zhang, Cheqing Jin, and Ying Yan. 2023. ChainDash: An Ad-Hoc Blockchain Data Analytics System. *Proc. VLDB Endow.* 16, 12 (2023), 4022–4025.
[38] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 11 (2020), 2047–2060.
[39] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proc. VLDB Endow.* 12, 11 (2019), 1539–1552.
[40] Zeshun Peng, Yanfeng Zhang, Qian Xu, Haixu Liu, Yuxiao Gao, Xiaohua Li, and Ge Yu. 2022. NeuChain: A Fast Permissioned Blockchain System with Deterministic Ordering. *Proc. VLDB Endow.* 15, 11 (2022), 2585–2598.
[41] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. 2021. RainBlock: Faster Transaction Processing in Public Blockchains. In *USENIX Annual Technical Conference*. USENIX Association, 333–347.
[42] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *EDBT*. OpenProceedings.org, 73–84.
[43] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *Middleware*. ACM, 13–25.
[44] Xiaodong Qi, Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2023. SChain: Scalable Concurrency over Flexible Permissioned Blockchain. In *ICDE*. IEEE, 1901–1913.
[45] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *SOSP*. ACM, 180–194.
[46] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *SIGMOD*. ACM, 1583–1598.
[47] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A Transactional Perspective on Execute-order-validate Blockchains. In *SIGMOD*. ACM, 543–557.
[48] Mark Russinovich, Edward Ashton, Christine Avanessians, et al. 2019. CCF: A framework for building confidential verifiable replicated services. *Technical report, Microsoft Research and Microsoft Azure* (2019), 1–17.
[49] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. In *SIGMOD*. ACM, 105–122.
[50] Zeshun Shi, Cees de Laat, Paola Grosso, and Zhiming Zhao. 2023. Integration of Blockchain and Auction Models: A Survey, Some Applications, and Challenges. *IEEE Commun. Surv. Tutorials* 25, 1 (2023), 497–537.
[51] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *SIGMOD*. ACM, 340–355.
[52] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997).
[53] Parth Thakkar and Senthilnathan Natarajan. 2021. Scaling Blockchains Using Pipelined Execution and Sparse Peers. In *SoCC*. ACM, 489–502.
[54] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1 (2010), 70–80.
[55] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*. ACM, 1–12.
[56] Shikun Tian, Zhonghao Lu, Haizhen Zhuo, et al. 2024. LETUS: A Log-Structured Efficient Trusted Universal BlockChain Storage. In *SIGMOD*. ACM, 161–174.
[57] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151 (2014), 1–32.
[58] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *PODC*. ACM, 347–356.
[59] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. 2020. Analysis of Indexing Structures for Immutable Data. In *SIGMOD*. ACM, 925–935.

[60] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. 2024. COLE: A Column-based Learned Storage for Blockchain Systems. In *FAST*. USENIX Association, 329–345.
[61] Yunhao Zhang, Srinath T. V. Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *OSDI*. USENIX Association, 633–649.
[62] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: Scalable Multi-Partition Transactions on Single-Threaded Partitioned Databases. *Proc. VLDB Endow.* 15, 11 (2022), 2939–2952.