SmartSched: Fine-Grained and Deterministic Parallel Execution for Smart Contracts

Hang Feng*
Zhejiang University
Hangzhou, China
h_feng@zju.edu.cn

Cong Wang
City University of Hong Kong
Hong Kong, China
congwang@cityu.edu.hk

Lei Wu
Zhejiang University
Hangzhou, China
lei_wu@zju.edu.cn

Yajin Zhou†
The Chinese University of Hong Kong
Hong Kong, China
yajin@ie.cuhk.edu.hk

Abstract

Parallel transaction execution has proven to be a promising technique for improving the throughput of blockchain systems. However, existing parallel blockchains often exhibit poor performance due to their transaction-level scheduling strategy. This coarse-grained concurrency control method suffers from high abort rates and limited parallelism under contended workloads.

To this end, we present SmartSched, a novel framework that exploits the *miner-replica* paradigm to accelerate parallel smart contract execution at replicas. SmartSched interleaves non-conflicting instructions from multiple transactions to expose fine-grained concurrency. It dynamically detects and resolves dependencies at runtime, and performs transaction scheduling at the instruction level. To further improve parallelism, transactions are executed within a multi-layer versioned state, which provides execution isolation, enables early dependency resolution, and supports commutative operations. SmartSched preserves the original transaction order semantics and remains compatible with current blockchain designs. Compared to existing parallel solutions, our approach not only avoids conflicts but also parallelizes non-conflicting instructions, thereby maximizing parallelism. We implement a prototype on Ethereum and evaluate it using Ethereum smart contracts. The evaluation results show that SmartSched peaks at a throughput of 223.4k TPS, achieving 1.6× to 11.6× improvements over baselines.

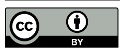
CCS Concepts

• **Computing methodologies → Concurrent algorithms**.

Keywords

Blockchain, Smart Contract, Parallel Execution, Fine-Grained, Concurrency Control.

*Part of this work was done when the author was a research assistant at CityUHK.
†Corresponding Author.



This work is licensed under a Creative Commons Attribution 4.0 International License.
Middleware '26, Tarragona, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2621-7/26/12
https://doi.org/10.1145/3801927.3810461

ACM Reference Format:
Hang Feng, Cong Wang, Lei Wu, and Yajin Zhou. 2026. SmartSched: Fine-Grained and Deterministic Parallel Execution for Smart Contracts. In *27th International Middleware Conference (Middleware '26), December 14–18, 2026, Tarragona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3801927.3810461

1 Introduction

Blockchain and smart contract technologies have witnessed rapid growth in recent years, facilitating the development of decentralized applications such as cryptocurrencies and Decentralized Finance. Users issue transactions to invoke smart contracts, which are customized programs executed on-chain. However, this rapid expansion has raised increasing concerns about the performance bottlenecks of current blockchain systems [32, 42, 48, 53, 63, 66], which have become a major obstacle to their widespread adoption.

Most blockchain systems today follow the *miner-replica* paradigm: a miner first generates a new block and broadcasts it to replica nodes across the distributed network. Each replica then independently executes the transactions to validate the block and updates its local blockchain state. In this paradigm, replica synchronization is critical to overall system performance. Recent advances in consensus protocols have significantly reduced the time needed to reach agreement. As a result, *transaction execution* at replicas is emerging as a new bottleneck for modern blockchains [21, 35].

To ensure that all nodes in the distributed network maintain consistent state, traditional blockchains adopt a single-threaded execution model, where transactions are processed sequentially in the miner-determined order (i.e., the preset order). However, this serial execution fails to take advantage of modern multi-core systems, thereby severely limiting scalability. To overcome this bottleneck, researchers have explored parallel transaction execution techniques [48, 53] to improve transaction throughput. However, to maintain a consistent global state across blockchain nodes, the result of parallel execution must be equivalent to that of serial execution under the preset order. In other words, the concurrent system must guarantee *deterministic serializability*. Early attempts [10, 18, 54] incorporate conventional deterministic concurrency control techniques [8] (e.g., validation-based schemes) to guarantee determinism and replicated execution. However, these general-purpose methods suffer from frequent conflicts and aborts due to the contended blockchain workloads [27, 38, 45, 54].

In the domain of deterministic databases [8], dependency-based concurrency control methods achieve conflict-free schedules based on inter-transaction dependencies. These methods typically require prior knowledge of transaction read/write sets. Dependencies may be analyzed before execution. BOHM[22] and PWV [23] construct dependency graphs for deterministic scheduling. Calvin [57] uses an ordered locking mechanism that implicitly enforces dependencies. Alternatively, dependencies can be detected at runtime, as in Scalable Replay [46], which waits on dependency.

Such dependency-based methods have also been incorporated into blockchain execution models. Since the miner is required to execute transactions during block generation in the *miner-replica* paradigm, recent research [9, 21, 24] has leveraged this property to improve replica-side parallelism. During block generation, the miner collects transactions' read/write sets to construct a transaction dependency graph. This graph is then shared with replicas to enable conflict-free scheduling, thereby reducing replica synchronization time to improve overall throughput.

Although the above dependency-based scheduling avoids conflicts and aborts at replicas, it still suffers from another significant limitation: the *coarse-grained scheduling strategy*. Existing works generally adopt transaction-level scheduling: a transaction can begin execution only after all its dependencies have been resolved, and these dependencies are resolved only upon the completion of the predecessor transactions. However, this coarse-grained method tends to overestimate dependencies among transactions, whereas the actual dependencies arise at the instruction level rather than at the transaction level. For example, a load instruction may depend on a specific store instruction issued by a predecessor transaction, rather than on the transaction as a whole. In practice, even transactions that conflict at the transaction level may still contain instructions that do not conflict [38]. Consequently, transaction-level scheduling may unnecessarily enforce serial execution even for non-conflicting instructions, leading to limited parallelism. We will further exemplify this inefficiency in Section 3.

To address the above limitations, we propose SmartSched, a finer-grained scheme for parallel smart contract execution at replicas. The core idea is to exploit intra-transaction parallelism by enabling non-conflicting instructions to execute in parallel, thereby improving parallelism and throughput. However, under the constraints of (i) the preset transaction order, and (ii) the complexity of smart contracts, generating such fine-grained conflict-free schedules presents two key challenges:

How to achieve correct and efficient scheduling? Unlike analyzing transaction dependencies, precomputing instruction-level dependencies is highly impractical. First, smart contracts are written in Turing-complete languages and are executed on stack-based virtual machines, making it difficult to pinpoint instruction dependencies [16, 26, 38, 40, 44, 58]. Second, even if such fine-grained dependencies could be obtained, the resulting data would be significantly larger than transaction-level dependency graph or read/write sets, introducing excessive storage and network overhead. To address this challenge, we propose a *scheduler* featuring *instruction-level scheduling* that interleaves instructions across transactions. Instead of relying on complete read/write sets to preconstruct a heavy conflict-free instruction-level schedule, the *scheduler* requires only write sets (obtained from the miner) and detects dependencies at runtime to preserve the preset order. Specifically, it preemptively detects potential dependencies before each read operation. Once an upcoming conflict is detected, the current execution is suspended, and the worker thread switches to another ready transaction. Similarly, once a write operation completes, the *scheduler* resumes the transactions waiting for that write. This fine-grained scheduling strategy parallelizes non-conflicting instructions from different transactions, while instructions within a transaction are still executed sequentially. Finally, execution results are validated and committed sequentially to ensure consistency.

How to resolve dependencies as early as possible? To avoid write conflicts, transactions are typically executed in isolated environments, where execution results remain invisible to other transactions until commit. However, executing instruction-level schedules requires resolving dependencies at the instruction granularity. Since the same state may be updated multiple times within a single transaction, simply allowing intermediate or unvalidated states to be accessed by other transactions prematurely, as employed in Block-STM [28]'s multi-version memory, accelerates dependency resolution but may lead to validation failures and re-execution. To address this challenge, we propose a *multi-layer versioned state* that maintains multiple state versions with different isolation levels. Between the isolated local environments and the committed global state, we introduce a shared pre-committed layer that holds partial execution results deemed safely accessible to subsequent transactions, even if the transaction has not completed. This design enables early and accurate dependency resolution while still preserving execution isolation. Only write–read dependencies are tracked, and each dependency is resolved once the corresponding write becomes pre-committed or finally committed. Additionally, we support commutative operations for cumulative state updates to further break up dependency chains.

SmartSched exploits instruction-level parallelism while preserving preset-order semantics. We implement SmartSched based on Go-Ethereum [3], and evaluate it against both general-purpose and blockchain-specific baselines using Ethereum [61] smart contracts. Our system achieves a peak throughput of 223.4k TPS, representing 11.6× over serial execution and 1.6×–3.7× over state-of-the-art solutions using transaction-level scheduling. Finally, we further evaluate the stand-alone performance and overhead of the instruction-level scheduling and the pre-commit mechanism.

In summary, this paper makes the following contributions:

- **Novel Execution Scheme.** We propose SmartSched, a fine-grained framework for replica-side smart contract execution that enables non-conflicting instructions to execute in parallel. To our knowledge, SmartSched is the first system to perform both dependency detection and resolution at the instruction level.
- **Scalable Scheduling Strategy.** We design a dedicated *scheduler* that dynamically detects and resolves dependencies at runtime, and schedules transactions at a fine granularity.
- **Efficient State Model.** We introduce the *multi-layer versioned state* model that achieves early and accurate dependency resolution while preserving execution isolation.
- **Comprehensive Evaluation.** We evaluate SmartSched from various aspects. The results demonstrate substantial improvements in performance and scalability.

2 Background and Related Work

2.1 Blockchain and Smart Contracts

A blockchain is an append-only ledger composed of a sequence of blocks, maintained by a distributed and trustless network. Every block contains an ordered list of transactions. Each node in the network stores a complete copy of the ledger. A blockchain node is typically divided into two layers: the consensus layer and the execution layer. The consensus layer is responsible for block synchronization and network consensus, while the execution layer processes transactions and updates the blockchain state (i.e., the world state). Our work focuses on parallel transaction execution on the execution layer. In this paper, we use Ethereum [61] as a representative platform to present our work.

Most blockchain systems adopt the *miner-replica* paradigm. At each block slot, the miner (elected by the consensus protocol) selects and executes candidate transactions to generate a new block, and broadcasts the block to the network. Replica nodes (validators) independently re-execute the transactions to validate the block and update their local blockchain states. To ensure a consistent blockchain state across all nodes, transactions within each block are executed sequentially, following the preset order determined by the miner. Lower transaction index indicates higher priority.

Ethereum adopts an account-based state model and maintains two categories of state: the account state and the storage state. Each account has an account state including its balance, nonce, and code hash (if it is a smart contract account). In addition, a smart contract account also maintains a private storage state, represented as a key-value mapping between 32 bytes. Both account and storage states are organized using Merkle Patricia Tries [4], indexed by account addresses and storage keys, respectively.

Users initiate transactions to invoke smart contract execution and update blockchain state. Smart contracts are user-defined, Turing-complete programs typically written in Solidity [7] and executed on the stack-based Ethereum Virtual Machine (EVM). A smart contract transaction may execute tens to hundreds of thousands of EVM instructions, depending on the contract complexity. Since the EVM is stack-based, instruction inputs depend on the stack and memory layout at runtime. This makes instruction dependencies implicit, especially when instructions span across basic blocks. Static analysis cannot accurately predict a transaction's execution trace due to the dynamic control and data flows of smart contracts [16, 44]. Even with dynamic analysis, capturing instruction-level dependencies incurs significant overhead (e.g., transforming instruction traces into SSA form [38, 58]).

2.2 Problem Statement

Task. Given a block of ordered transactions, a concurrent schedule must satisfy *deterministic serializability*, that is, the execution results must be identical to those produced by sequentially executing the transactions under the preset order.

Scope. This paper focuses on parallel execution of transactions *within a single block*. While some blockchain systems also explore block-level parallelism—such as DAG-based blockchains [12, 36, 37, 60] that allow multiple blocks to be produced and processed in parallel—these topics are beyond the scope of this paper.

Table 1: Comparison of SmartSched with existing works.

Execution Scheme	Method(s)	Prior Knowledge	Scheduling Granularity	VM-Compatible
OCC [5, 10, 54], 2PL [18]	N/A	N/A	TX	✓
Sei-v2 [6], Block-STM [28]	I*	N/A	TX	✓
[49], [9], [21], [64] [65], [45], [11]	I	RW	TX	✓
Fabric+(s) [51, 55] [52], [62], [31], [30]	I, II	RW	TX	✓
OCC-DA [27]	I, III	RW	TX	✗
PaVM [24]	I, III	RW	FUNC	✗
Crystality [59]	III, IV	N/A	STATE	✗
ParallelEVM [38] Forerunner [17]	IV	N/A	TX	✓
Spectrum [19]	I, IV	RW	TX	✓
SmartSched	I	W	INSTR	✓

I: Dependency-guided scheduling. II: Transaction reordering.
III: Parallel programming language support. IV: Fast re-execution.
VM-Compatible: Refers to semantic compatibility, i.e., the system produces identical results to the original VM and can therefore be integrated into existing blockchains without requiring a hard fork of the VM.
*The systems may miss dependencies, leading to aborts and re-execution.

2.3 Concurrency Control Basics

Concurrency control methods generally fall into two categories: optimistic concurrency control (OCC, [33]) and pessimistic concurrency control (PCC). OCC assumes that most transactions do not conflict and thus allows them to proceed concurrently without acquiring locks. A validation phase is performed before commit to detect any violations of serializability. In contrast, PCC (e.g., two-phase locking, 2PL [14]) adopts a conservative approach, requiring transactions to acquire locks before accessing shared data. Multiversion concurrency control (MVCC, [15]) maintains multiple versions for each data item, allowing concurrent transactions to access different data versions. MVCC can be used to implement snapshot isolation [13] and reduce write conflicts.

2.4 Related Work

Parallel transaction execution has been widely exploited to improve the throughput of blockchains. Early efforts directly apply general-purpose concurrency control algorithms under the constraint of preset order, such as OCC with serial validation and commit and 2PL. However, these systems perform poorly (see Section 6.3) in blockchain environments due to high contention and limited scheduling flexibility. Thus, recent works have proposed blockchain-specific concurrency control mechanisms featuring diverse optimization methods. These works can be broadly categorized based on the following optimization methods: dependency-guided scheduling, transaction reordering, parallel programming language support, and fast re-execution. We characterize and compare representative works with our solution in Table 1, where R/W denotes the use of read/write sets.

Most dependency-guided scheduling systems precompute dependencies from transactions' read/write sets and construct conflict-free transaction-level schedules, where a transaction begins execution only after all its predecessors have committed. Sei-v2 [6]

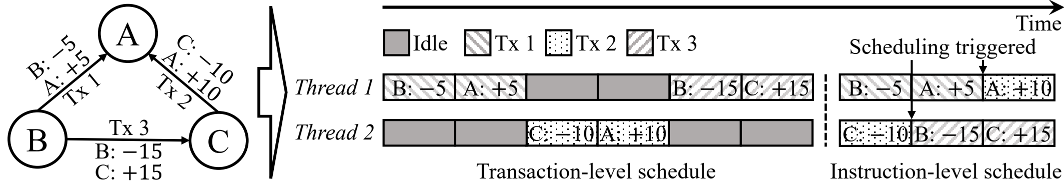


Figure 1: A money transfer example of transaction- and instruction-level scheduling.

predicts the states accessed by transactions to estimate dependencies prior to execution. DMVCC[45] employs static analysis of smart contract source code to identify writes that can be exposed early. Block-STM [28] optimistically runs execution and validation concurrently using a multi-version memory model, where execution results are visible to other transactions before validation. It analyzes dependencies at runtime without requiring prior knowledge, but may miss dependencies, leading to conflicts and aborts. When validation fails, the read/write sets observed during the previous execution can be used to infer dependencies and guide subsequent re-execution. Block-STM and Spectrum [19] improve concurrency through runtime dependency detection, where (potential) conflicting transactions may be blocked. Although reducing conflicts/aborts, the above systems still stall conflict-free operations. By comparison, SmartSched requires only write sets, yet achieves an accurate and fine-grained dependency resolution.

Another line of work allows miners to reorder transactions ahead of block generation, aiming to reconcile conflicts and improve parallelism. Following the *execute-order-validate* paradigm, Fabric's extensions [51, 55] analyze read/write sets and adjust the total order to reduce conflicts and shorten the critical path. ASMR [30] departs from the consensus-imposed total order and derives an optimal deterministic serializable schedule based on an undirected conflict graph. Jin et al. [31] and Nezha [62] enhance the reordering process through optimized graph structures and advanced sorting algorithms. OptME [52] further improves performance by optimizing the construction of dependency graph and enabling the inter-epoch reordering. Although transaction reordering helps reduce conflicts, it may conflict with the incentive mechanisms in current permissionless blockchains. For example, in Ethereum, miners prioritize transactions that offer higher gas fees [20, 41]. In contrast, SmartSched preserves the preset transaction order.

Some studies introduce parallelism into smart contract languages and design virtual machines with native support for parallel execution. Crystality [59] partitions contract storage into disjoint, parallelizable segments and enables asynchronous functions to achieve state-level parallelism. OCC-DA [27] introduces commutative opcodes for smart contracts to break conflict chains. PaVM [24] provides programming interfaces for *subfunctions* that can run in parallel, enabling function-level parallelism. However, these approaches shift the burden of concurrency control from the runtime system to developers and users. In addition, they are incompatible with existing blockchain platforms, limiting their deployability in practice. In contrast, SmartSched is user-transparent and can be seamlessly integrated into mainstream blockchain infrastructures.

Fast re-execution is employed to mitigate the overhead caused by transaction aborts. During optimistic execution, ParallelEVM [38] synthesizes *SSA operation logs* to facilitate precise conflict identification and efficient partial re-execution. Crystality re-executes only the conflicting function chains. Spectrum [19] supports partial rollback by maintaining multiple stack snapshots and a multi-versioned memory for each EVM instance. Forerunner[17] speculatively executes transactions across multiple futures and generates specialized *accelerated programs* to speed up final execution. Although originally designed for speculation, its program specialization technique can also be used to accelerate re-execution. Fast re-execution and SmartSched address conflicts from different angles—one accelerates recovery after conflicts occur, the other improves scheduling and avoids conflict occurrence.

Deterministic databases [8, 34, 47, 50, 56] have been extensively studied. Similar to blockchains, these systems enforce an agreed-upon serialization order to ensure replica consistency, although this agreed order is not necessarily pre-determined. Aria [39] does not require prior knowledge and introduces a deterministic reordering mechanism that transforms RAW dependencies into WAR dependencies. Although reducing conflicting transactions, Aria fails to satisfy the preset order constraint. Calvin [57] proposes a deterministic locking strategy that relies on pre-declared complete read/write sets. Each transaction must acquire all locks for its accessed records before execution, following the global transaction order. This implicitly enforces dependencies before execution begins. BOHM [22] decomposes transaction processing into the currency control phase and the execution phase. Dependency graphs are built prior to execution. A transaction becomes ready when all its read dependencies are resolved. PWV [23] also uses dependency graphs and introduces the concept of *early write visibility*, which allows write operations to be accessed before the end of execution. Scalable Replay[46] requires only write sets and detects dependencies at runtime. During execution, a read operation is blocked if it depends on a preceding write that has not yet been completed. DORA [43] decomposes a transaction into fine-grained actions that can be executed by different threads based on data partitioning. These systems also operate at the coarse granularity of a transaction.

3 Motivation

3.1 Fine-Grained Scheduling Opportunity

Many existing approaches (marked as I in Table 1) exploit transaction dependencies to construct conflict-free transaction-level schedules. In these designs, each transaction is treated as an indivisible scheduling unit and can begin execution only after all its dependencies have been resolved. Moreover, a `write-read` dependency is not resolved until the writer transaction completes. Despite avoiding conflicts, they fail to capture intra-transaction scheduling opportunities. However, modern blockchain transactions often exhibit

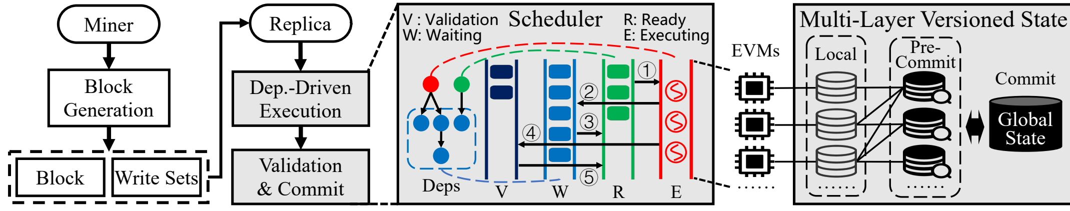


Figure 2: SmartSched architecture.

partial conflicts, where conflicts arise only among a subset of instructions [38]. Thus, deferring the entire transaction may block non-conflicting instructions, reducing parallelism and underutilizing computational resources. Figure 1 illustrates a money transfer example, which represents the most prevalent workload in current blockchains [27, 45]. Consider using two threads to execute three money transfer transactions, where each transaction increases the balance of one account and decreases that of another. Since every transaction accesses the balances of two out of three accounts, they collectively form a dependency chain. Under transaction-level scheduling, these transactions must be executed serially, leaving one thread idle and consuming a total of six time slots. However, within each transaction, the two balance updates are independent of each other. This allows for a more efficient schedule at the instruction level, where non-conflicting operations across transactions can be executed in parallel. The right side of Figure 1 illustrates an optimal schedule that interleaves the operations from the three transactions, enabling all of them to complete within just three time slots. For Tx_2, the operation $C-10$ is executed first, while $A+10$ is postponed until Tx_1 completes. Likewise, Tx_3 starts immediately after $B-5$ is executed, without having to wait for the completion of Tx_1. Motivated by this observation, we propose a finer-grained strategy that exploits instruction-level scheduling to maximize parallelism. To avoid the overhead of collecting full read/write sets and preconstructing dependency graphs, our approach relies only on write sets and detects dependencies dynamically at runtime.

3.2 Accelerating the Miner-Replica Paradigm

In the *miner-replica* paradigm, replica-side execution lies on the critical path of network synchronization. Blockchain networks typically involve a large number of replicas. Thus, optimizing transaction execution on replicas is critical to improving system throughput. Meanwhile, the miner must execute transactions to update the state and assemble a valid block, thereby naturally observing transaction write sets during block generation. By sharing this information as scheduling hints, replicas can detect dependencies at runtime and perform instruction-level scheduling. Although this incurs additional overhead for the miner, accelerating replica execution improves system throughput, which in turn increases the miner's expected rewards. Hence, miners are naturally incentivized to provide such metadata [21, 29]. In practice, sharing read/write sets to guide replica-side scheduling has been employed in previous works, as discussed in Section 2.4.

4 SmartSched Overview

SmartSched is designed to maximize parallelism and avoid conflicts while preserving the preset order at replicas. It introduces instruction-level scheduling to enable interleaved execution, thereby allowing non-conflicting instructions across transactions to execute in parallel. Figure 2 depicts the overall workflow and components of our method. While generating a block, the miner observes and records transaction write sets, which are disseminated to replicas along with the block for subsequent execution. Replicas employs a two-phase processing model: dependency-driven execution and sequential validation and commit. Transactions are executed in a *multi-layer versioned state* that supports early dependency resolution while isolating execution environments, coordinated by a *scheduler* that dynamically detects dependencies. Execution results are validated and committed sequentially.

Scheduler. The *scheduler* maintains dependencies and coordinates transaction processing at runtime. It adopts a dependency-driven speculative execution model. Initially, the *scheduler* marks all transactions as ready. Whenever an idle worker thread is available and there exists a ready transaction, the *scheduler* dispatches that transaction for execution (①). Instead of precomputing dependencies for all instructions, the *scheduler* inspects instruction execution and analyzes dependencies at runtime. Specifically, before executing each instruction, it leverages the input write sets to check whether the instruction reads a state that is expected to be written by a uncommitted predecessor transaction. Upon detecting an impending conflict, the *scheduler* suspends the execution of the affected transaction (②), preventing unnecessary aborts. To avoid idle waiting, the worker thread attempts to switch to another ready transaction when a transaction is suspended, ensuring continuous thread utilization. Whenever a dependency is resolved (through pre-commit or final commit), the *scheduler* marks the suspended transaction as ready again (③), allowing it to continue execution without restart.

The transaction enters the pending validation state after completing execution (④). However, it can be validated only after all preceding transactions have been committed. The validation verifies whether the execution's read set remains consistent with the latest global state. If the validation succeeds, the execution result is committed to the global state; otherwise, it is aborted and marked as ready for re-execution (⑤). By enforcing a sequential validation-commit pipeline, the system preserves correctness and deterministic serializability under malicious write sets.

Multi-Layer Versioned State. To avoid conflicts and improve parallelism, we introduce the *multi-layer versioned state* model, which stores multiple state versions across transactions and organizes them into three isolation levels:

1. **Local state**: Each transaction is executed within a fully isolated environment invisible to other transactions. It stores the intermediate results during the execution. In addition, it also maintains

Algorithm 1: Optimistic execution.

```
Data: A block: B = ⟨tx_0, tx_1, ..., tx_{n-1}⟩;
A list of execution contexts: C = ⟨ctx_0, ctx_1, ..., ctx_{n-1}⟩;
A priority queue of ready txs: Q_ready;
1  for i in [0, 1, ..., n-1] do
2      Q_ready.Push(tx_i); // Initialization.
3  while tx_{n-1} is not committed do
4      if Q_ready is not empty and an idle worker thread exists
5          tx_i ← Q_ready.Pop();
           // Run the following on the idle worker.
6          if tx_i has not started
7              ctx_i ← NewContext(tx_i);
8              Start execution of tx_i; // See Algorithm 2.
9          else SwitchTo(tx_i, ctx_i); // Resume execution.
```

Algorithm 2: Instruction-level scheduling.

```
Data: Execution contexts: C = ⟨ctx_0, ctx_1, ..., ctx_{n-1}⟩;
Ready queue: Q_ready;
A list of waiting sets: W = ⟨W_0, W_1, ..., W_{n-1}⟩;
Input: Transaction to be executed: tx_i;
1  while (op ← GetNextOp(ctx_i) ) ≠ STOP do
2      if op == SLOAD and IsFirstRead(tx_i, op.key)
3          prec ← CheckLastWrite(tx_i, op.key);
4          if prec ≥ 0 and not IsCommitted(tx_prec)
             and not IsPreCommitted(tx_prec, op.key)
5              Add (i, op.key) into W_prec;
6              Suspend current tx_i and yield worker thread;
7      RunOp(ctx_i, op);
8      if op == SSTORE and IsLastWrite(tx_i, op.key, id)
9          PreCommit(tx_i, op.key, op.value);
10         for (waiting, key) in W_i do
11             if key == op.key
12                 Remove (waiting, key) from W_i;
13                 Q_ready.Push(tx_waiting);
```

cumulative values to support commutative operations that break up the transaction's critical path.

2. **Pre-committed state**: Each transaction owns a semi-isolated environment that is readable by subsequent transactions. It holds selected execution results promoted from the local state, allowing subsequent transactions to read them early, even if the execution has not yet completed.
3. **Global state**: A shared layer that maintains all committed states and is accessible to all transactions.

Transaction execution first attempts to read from the local state. If unavailable, the read is redirected to the pre-committed state of the uncommitted predecessor transaction. If still not found, the read operation ultimately falls back to the global state. Write operations are initially applied to the local state. If a write is identified as the final update to the state for the transaction (according to the transaction's write set), it is optimistically updated to the pre-committed state, allowing subsequent transactions to access it early without unnecessary delays. When a transaction passes validation, its local state is committed to the global state. Once the last transaction is committed, the global state equals to the final results.

Our *multi-layer versioned state* prevents dirty reads (via the isolated local state) while allowing safely accessible states to be exposed as early as possible (via the pre-committed state). Finally, the global state provides the authoritative view of committed results.

5 Design and Implementation

During block generation, the miner executes all transactions sequentially and extracts the write set of each transaction by capturing write instructions. To uniquely identify write operations, we introduce the *instruction ID*, an incremental number assigned to each executed instruction within a transaction. Unlike the program counter (PC), which may repeat across different iterations or calls, the instruction ID uniquely labels every instruction instance in the dynamic execution trace. Thus, the write set consists of ⟨`WKey`, `WID`⟩ pairs. `WKey` is the key written by the instruction. `WID` means the instruction ID of the last instruction that writes to the corresponding `WKey` in the transaction. Write sets are disseminated to replicas for scheduling during parallel execution. `WKey` serves as the basis for dependency detection, while `WID` helps determine whether a write operation can be safely promoted to the pre-committed state.

5.1 Scheduling

As Figure 2 illustrates, SmartSched employs the *scheduler* to coordinate transaction processing. Each transaction transitions through one of the following statuses:

- `Ready`: The transaction is ready to start or resume execution.
- `Executing`: The transaction is currently executing.
- `Waiting`: The transaction is suspended due to an unresolved dependency on a write from a preceding transaction.
- `Validation`: The transaction has completed execution and is pending validation.

The *scheduler* maintains ready transactions using a priority queue Q_{ready}, The queue is keyed by transaction indices, with smaller indices indicating higher priority. This heuristic aligns with the original sequential semantics of the blockchain, where transactions with lower indices are executed earlier. Each transaction is executed within the EVM instance of a worker thread, using its own execution context, which includes the local state, pre-committed state, and runtime data (e.g., stack, memory, code, and runtime instruction ID). Meanwhile, all transactions share a global state. Algorithm 1 describes our optimistic execution strategy. During initialization, all transactions are optimistically pushed into Q_{ready} (Lines 1-2). Then, the scheduling for-loop iterates until the last transaction is committed (Lines 3-9). It continuously evaluates two conditions: (1) Whether Q_{ready} contains any ready transactions, and (2) whether an idle worker thread is available. Whenever both conditions are satisfied, a ready transaction is popped from Q_{ready}, marked as `Executing`, and dispatched to the idle worker thread: If the transaction has not started, a new execution context is initialized, and the execution starts from the beginning (Algorithm 2). Otherwise, the suspended execution resumes from the point where it was interrupted (Line 6 in Algorithm 2).

To track dependencies, the *scheduler* dynamically maintains a waiting set $W_i = \{(j, key_{i,j}) | i < j < n\}$ for each transaction tx_i. Each entry $(j, key_{i,j})$ indicates that tx_j is currently waiting for tx_i's write to the state $key_{i,j}$. Algorithm 2 illustrates how the *scheduler*

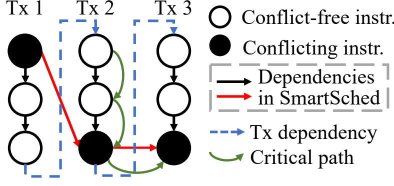


Figure 3: An example of dependencies in SmartSched.

detects dependencies and achieves instruction-level scheduling. The *scheduler* inspects every state-access instruction and analyzes their dependencies at runtime. Here, we use contract storage as an example; while other states (e.g., balance and nonce) follow a similar approach. Before executing a storage load instruction (`SLOAD`), the scheduler checks whether this is the first read of the key within the transaction (Line 2). If so, the *scheduler* uses the function `CheckLastWrite` to determine whether the target key has a write by a predecessor transaction (Line 3). `CheckLastWrite` searches `WKeys` of preceding transactions and returns the last transaction that writes to the key, or −1 if none exists. If such a predecessor transaction exists and its corresponding write has neither been committed nor pre-committed (Line 4), a dependency is detected. In this case, the current transaction transitions to the `Waiting` status. It is then added to the predecessor's waiting set, along with the target key (Line 5). Finally, the *scheduler* suspends the execution and yields the worker thread to another ready transaction (Line 6), avoiding wasted CPU resources due to idle waiting. Once the dependency is resolved, the state returned by the initial `SLOAD` is cached in the local state. Any subsequent `SLOAD` on the same key can then directly access the cached value without additional checks. After executing a storage store instruction (`SSTORE`), the scheduler uses the function `IsLastWrite` to check whether it is the transaction's final write to the key (Line 8), by comparing the current instruction ID with the key's `WID`. If so, the write is pre-committed (Line 9), allowing subsequent transactions to access this state early. The pre-commit resolves dependencies as early as possible. Then, waiting transactions depending on this `SSTORE` are moved back to Q_{ready} (Lines 10-13), enabling their execution to be resumed later (Line 9 in Algorithm 1). In SmartSched, dependency analysis is performed dynamically at runtime. The *scheduler* detects dependencies only for `Executing` transactions. Nevertheless, all transactions will eventually be executed and analyzed.

Figure 3 illustrates how SmartSched handles dependencies. Within each transaction, instructions are still executed sequentially, forming an intra-transaction dependency chain (black arrows). Across transactions, however, SmartSched maintains fine-grained instruction-level dependencies instead of coarse-grained transaction-level dependencies. The red arrows represent inter-transaction dependencies between conflicting instructions that access the same state. In contrast, the blue dashed arrows represent transaction-level dependencies, where each transaction must wait for the complete execution of the previous one. As shown, SmartSched has a dependency graph with a critical path length of 4, whereas the transaction-level scheduling yields a critical path length of 9 by serializing entire transactions unnecessarily.

Once the execution completes, the transaction transitions to the `Validation` status, pending validation before commit. Validation

Algorithm 3: Validation and commit.

```
// tx_i can be validated only after tx_{i-1} is committed.
Data: Ready queue: Q_ready;
waiting sets: W = ⟨W_0, W_1, ..., W_{n-1}⟩;
Input: Tx to be validated and committed: tx_i;
Read set of tx_i: {r}_i;
Output: Whether the validation & commit succeeds.
1  if {r}_i matches global_state then // Validation
2      Commit(tx_i);
       // post-commit scheduling.
3      for (waiting, _) in W_i do
4          Q_ready.Push(tx_waiting);
5      W_i.Clear();
6      return true; // tx_i is committed.
7  else // Dirty read is detected
8      Abort(tx_i); // Abort the execution of tx_i.
9      Q_Ready.Push(tx_i); // Schedule tx_i for re-execution.
10     return false; // Validation fails.
```

and commit are performed serially, strictly following the preset transaction order. Algorithm 3 demonstrates the pipeline of validation and commit. The validation compares the transaction's read set against the committed global state (Line 1), ensuring that the read values are consistent with the states immediately before the transaction in the serial execution model. Thus, to guarantee that the global state correctly reflects the committed states during validation, transaction tx_i can be validated only after tx_{i-1} has been committed. If the validation succeeds, the execution result is committed to the global state (Line 2), and a post-commit scheduling procedure is conducted to move all waiting transactions from W_i to Q_{ready} (Lines 3-5). Otherwise, the execution result is aborted (Line 8), and the transaction is pushed back to Q_{ready} for a restart (Line 9). Due to the serial validation-commit mechanism, each transaction may fail validation (and require re-execution) at most once. In Section 5.3, we will elaborate on the necessities of the validation and post-commit scheduling. Note that higher-priority transactions are guaranteed to eventually complete, thereby resolving dependencies and releasing threads for subsequent transactions. Thus, our scheduler inherently guarantees no starvation or deadlock.

5.2 Multi-Layer Versioned State

To achieve the isolated execution environments and pre-commit mechanism described earlier, we propose the *multi-layer versioned state* model defined as follows:

$$MLVS = \langle \{L_i\}_{i=0}^{n-1}, \{P_i\}_{i=0}^{n-1}, G \rangle$$

Here, n is the number of transactions, and $MLVS$ maintains $2n + 1$ stores, corresponding to state snapshots at different transaction versions or isolation levels. The three isolation layers are L (local), P (pre-committed), and G (global).

Each transaction owns a private L that stores the intermediate states for the current execution. All operations execute directly within this isolated environment to prevent dirty reads. For each key, L maintains two types of states: `origin_state`, which stores the initial loaded value, and `dirty_state`, which records the latest value written during execution.

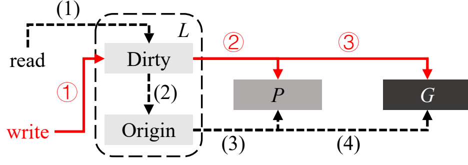


Figure 4: Dataflow in *MLVS*.

To enable pre-commit, each transaction maintains a semi-isolated P that stores a subset of `dirty_states` promoted from L. By safely exposing these states to subsequent transactions before the transaction completes, P reduces inter-transaction waiting time. It is worth noting that not all `dirty_states` are eligible for pre-commit, as a state may be written multiple times within a single transaction. Thus, the introduction of P is necessary, rather than simply making L accessible to other transactions.

All transactions share a singleton G maintaining committed states. G is read-only during execution and validation. Execution result is committed to G after validation succeeds.

Before execution, G represents the initial state of the block. Each transaction is initialized with an empty L and an empty P. Figure 4 illustrates the dataflow in *MLVS*.

During execution, a state read follows the chain of *dirty* → *origin* → *pre-committed* → *global*. (1) If a `dirty_state` exists in L, the read returns it; (2) otherwise, it attempts to load the `origin_state`. If the state is absent in L (i.e., an initial read), dependency detection is triggered (Lines 3–6 in Algorithm 2). (3) Once the dependency value is pre-committed, the read checks the predecessor transaction's P. (4) If no dependency exists, or if the predecessor has been committed, the read falls back to G. In cases (3) and (4), the initially loaded state is cached as the missing `origin_state` in L. Thus, `origin_states` constitute the read set of the execution.

The write operation first updates `dirty_state` directly (①). If a `dirty_state` is determined to be the final version after the execution, it is forwarded to P (②, Line 9 in Algorithm 2), even if the execution has not completed. During validation, all `origin_states` are verified against the latest G. If validation succeeds, all `dirty_states` are committed to G (③). Otherwise, L and P are aborted. After a transaction commits, its L and P are cleared. Once the last transaction commits, G reflects the final state under the preset order.

In real-world scenarios, almost all transactions conflict on balance and nonce states, primarily due to the transaction fee and anti-replay mechanisms:

(1) Transaction execution requires paying a fee to the miner, causing all transactions depending on the miner's balance.
(2) To prevent replay attacks, each transaction execution increments the sender's nonce. Consequently, multiple transactions from the same sender will conflict on the sender's nonce.

To address the above issues, we introduce *commutative* operations for *additive/subtractive* account-state updates: balance transfers and nonce increments. SmartSched intercepts the corresponding operations (value transfer via CALL-family instructions and nonce updates during transaction initiation) to apply commutative handling. Instead of reading the `origin_state` and then producing a `dirty_state`, L accumulates updates into a delta Δ during execution, without accessing the `origin_state`. During validation and commit, the `origin_state` is loaded to compute the final state as (`origin_state` $+ \Delta$). Commutative operations do *NOT* contribute to read set, although they still increase write set just like standard (non-commutative) update operations. Thus, transactions that update the same state via commutative operations can be executed concurrently without conflicts. The commutative operations for balance and nonce states are *fully compatible* with the current EVM, as they do not modify the instruction set. It is worth noting that commutative operations do not apply to contract storage, because storage update is performed through the `SSTORE` instruction, which overwrites a value rather than applying an incremental update.

Compared to Block-STM's multi-version structure [28], where transactions can directly read speculative (i.e., unvalidated) execution results produced by preceding transactions, our multi-layer versioned structure employs the local state to avoid dirty reads and enables results to be accessed as early as possible via the pre-committed state. Writes are never blocked. A read operation is blocked only when it targets a write that remains uncommitted in a predecessor transaction (i.e., `write-read` dependency). Finally, the global state safeguards final deterministic serializability.

5.3 Correctness and Fault Tolerance

SmartSched's dependency detection and pre-commit rely on input write sets. When the write sets are complete and accurate, `WKeys` enable precise detection, and `WIDs` ensure that early-exposed writes are consistent with their final committed values. Consequently, every read observes exactly the same state as in serial execution. No conflict arises and validation always succeeding.

Fault Tolerance. However, in decentralized and trustless blockchain networks, write sets from malicious peers may be manipulated. This may mislead the *scheduler* into missing or overestimating dependencies for subsequent transactions. Specifically, `WKey` and `WID` may be missing or incorrect, leading to the following issues:

(1) Missing `WKey`: The dependency is missed, causing subsequent transactions to run prematurely and observe the incorrect state.
(2) False `WKey`: The false-positive dependency is introduced, leading to unnecessary transaction stalls.
(3) Missing `WID`: The state update cannot be confirmed as pre-committed, delaying dependency resolution.
(4) False `WID`: The intermediate state is pre-committed, causing subsequent transactions to read the incorrect state.

Therefore, ensuring the security and correctness is crucial in untrusted scenarios. The validation phase is imposed to tackle cases (1) and (4), where inconsistent read set is detected, triggering an abort and re-execution. For cases (2) and (3), the post-commit scheduling eventually releases waiting transactions, even in the presence of false dependencies or missing pre-commits.

Correctness Proof. We prove correctness in the presence of incorrect write sets by induction on the transaction order.

Base case. For tx_0, execution is performed against the initial state of the block. Since no dependency exists, the validation succeeds and tx_0 is committed.

Inductive step. Assume that transactions $tx_0, \ldots, tx_{i-1}$ have been correctly committed. Since validation and commit are performed sequentially according to the preset transaction order, when

validating tx_i, the global state correctly reflects the committed results of $tx_0, \ldots, tx_{i-1}$. If the execution of tx_i is incorrect due to inconsistent write sets, validation detects the inconsistency in the read set and aborts the execution. tx_i is then re-executed based on the correct state after tx_{i-1}, allowing tx_i to be safely committed.

Therefore, all transactions are validated and committed in a manner equivalent to serial execution under the preset order, guaranteeing correctness and *deterministic serializability*.

5.4 Implementation

We implement SmartSched based on Geth v1.13.15, using Golang v1.22.8. Instead of manually managing threads, we leverage the Golang runtime for efficient task scheduling. At the beginning of execution, each transaction is assigned an independent goroutine. Since goroutines are lightweight and managed by the Go runtime, this approach enables efficient concurrency and resource savings without excessive OS thread creation. We use condition variables to implement the status transition between `Executing` and `Waiting`. Suspension (Line 6 in Algorithm 2) is handled via the `Wait()` function, and resumption (Line 9 in Algorithm 1) is triggered through the `Signal()` function. Finally, the goroutine scheduler handles context switching automatically and efficiently.

The ready queue Q_{ready}, pre-committed state P, and global state G are implemented using thread-safe structures (e.g., concurrent min-heap and maps). The dependency tracking (Lines 4-5 in Algorithm 2), pre-commit procedure (Lines 9-13 in Algorithm 2), and post-commit scheduling together form a critical section that is protected by a lock. The local state L implements the `StateDB`[1] interface required by the EVM for transaction execution. The global state G wraps the underlying Merkle Patricia Trie, where all reads and writes are finally applied.

We use an incremental integer `next_validation` to track the next transaction scheduled for validation. Following completion of any execution, the worker thread checks whether `next_validation` is ready for validation. If so, it proceeds with the validation. A transaction being validated implies that all preceding transactions have been committed. Thus, if validation fails, re-execution starts immediately within the same worker thread to reduce unnecessary status transition. Besides, to improve efficiency, commutative operations are disabled during re-execution, and the execution result can be directly committed without requiring an additional validation.

6 Evaluation

6.1 Environment Setup

Baselines. We compare SmartSched with both general-purpose and blockchain-specific baselines as follows: Serial, OCC, 2PL, DMVCC [45], Spectrum [19], Block-STM [28], and ParallelEVM [38]. Serial executes transactions sequentially on the global state without validation or commit. OCC performs optimistic execution without dependency-aware scheduling, followed by sequential validation and commit. For 2PL, we implement a variant of rigorous two-phase locking, where `write-write` conflicts are allowed. Each state is associated with a shared lock for read operations and an exclusive lock for write operations. Locks are released only after the transaction commits or aborts. DMVCC and Spectrum employ transaction-level scheduling based on precomputed dependencies. OCC, 2PL, Block-STM, and ParallelEVM may experience transaction aborts due to conflicts. In contrast, other methods (including ours) have a zero abort rate by design. We implement and integrate these baselines into SmartSched. Every baseline executes transactions in isolated environments without the pre-commit mechanism. To ensure a fair comparison, we also enable commutative operations for baselines (except Block-STM, as its design does not support this optimization). We do not directly compare against deterministic database systems such as Calvin [57], BOHM [22], or Scalable Replay [46], because their core techniques have already been adapted and incorporated into blockchain-oriented systems like DMVCC [45] and Spectrum [19] (see Section 2.4). Our evaluation therefore focuses on these domain-specific baselines.

Workloads. We evaluate SmartSched using two types of workloads: real-world historical transactions on the Ethereum mainnet and ERC-20 [2] token transfers. Ethereum is the largest blockchain platform that supports smart contracts. The ERC-20 smart contract implements a standard token protocol, and its transfer is the most frequent transaction type across EVM-compatible blockchains. For the history workload, we sample 10,000 blocks from block heights 19,000,000 to 20,000,000 (January-June 2024) at a fixed interval of 100. To execute these blocks, we deploy an archive node SlimArchive [25] to generate historical states. All required historical states are cached in memory before execution. For the ERC-20 workload, we first deploy an ERC-20 contract and distribute tokens to a predefined set of accounts. Then, we generate transactions that randomly transfer tokens among these accounts and pack them into blocks. Each ERC-20 transaction transfers tokens between two accounts. For each ERC-20 workload configuration (i.e., different block size and account set), we synthesize 200 blocks.

All experiments are conducted on a physical machine equipped with dual-socket Intel(R) Xeon(R) Silver 4214R CPUs (12 physical cores per socket, with Hyper-Threading enabled) and 377 GB of memory, running Ubuntu 22.04.2 LTS.

6.2 Correctness Validation

We first verify the correctness of our SmartSched implementation and the baselines. We execute historical Ethereum blocks using both SmartSched and the baseline methods. For SmartSched, we evaluate three modes: one with complete write sets, one with partial write sets, and one with incorrect write sets. After execution, we compare the execution results (e.g., gas usage, logs, account balances, and storage states) of each method against the original on-chain results. All validations pass, confirming the correctness and robustness of our implementations.

6.3 End-to-End Performance

In this section, we evaluate the end-to-end performance of block processing across different thread counts, block sizes (i.e., the number of transactions per block), and contention levels, using the throughput (TPS) and speedup over serial execution as metrics.

Thread Count. For the ERC-20 workload, we set a block size of 1000 transactions and a total of 200 accounts. Figure 5 shows the throughput under different thread counts. In general, SmartSched

[1]https://github.com/ethereum/go-ethereum/blob/master/core/vm/interface.go

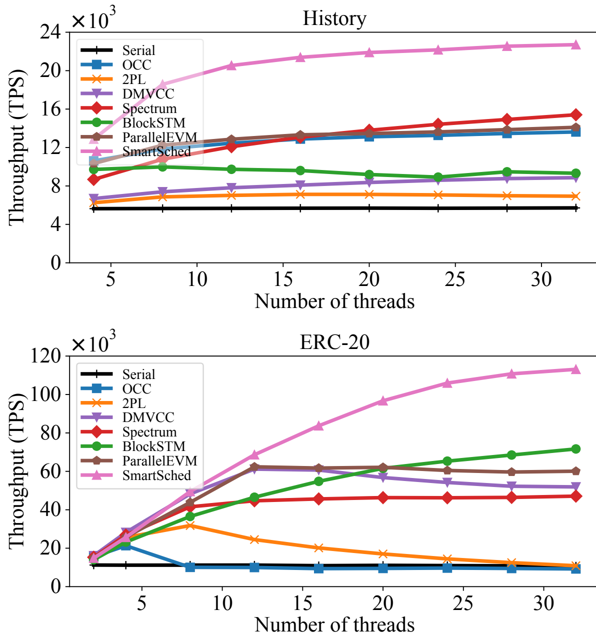


Figure 5: Speedups under different thread counts.

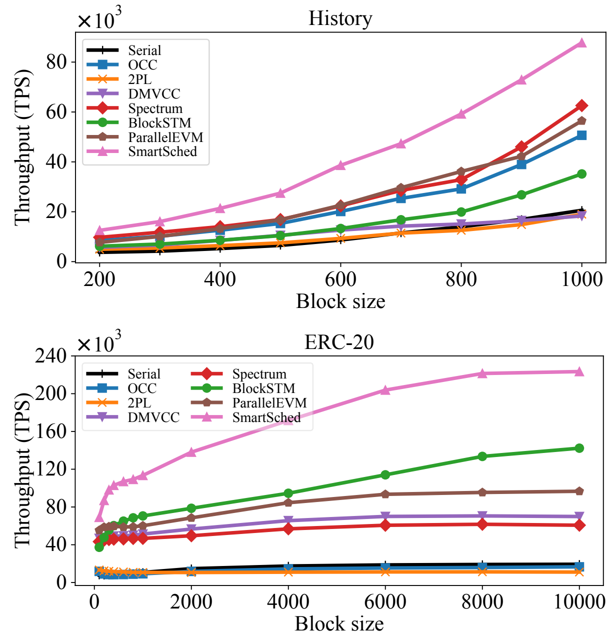


Figure 6: Speedups under different block sizes.

consistently outperforms all other schemes across all thread counts. For the history workload, SmartSched's performance starts to saturate at 16 threads, reaching 21.4k TPS, which is 3.8× over Serial and 1.6× over ParallelEVM. At 32 threads, it achieves up to 22.7k TPS, which is 4.0× over Serial and 1.5× over Spectrum. For the ERC-20 workload, SmartSched delivers even higher efficiency, scaling up well to 28 threads. In contrast, ParallelEVM, DMVCC, and Spectrum, which employ transaction-level scheduling, begin to show degraded performance starting from 16 threads due to limited scheduling granularity. At 32 threads, SmartSched achieves up to 113.1k TPS, which is 10.5× over Serial and 1.6× over Block-STM.

Block Size. In this experiment, we fix the thread count to 32 and set the total number of accounts to 200 for the ERC-20 workload. Figure 6 presents the throughput under different block sizes. For the history workload, SmartSched continues to scale as the block size increases. At block size 1,000, SmartSched achieves up to 87.8k TPS, which is 4.3× over Serial and 1.4× over Spectrum. For the ERC-20 workload, the speedup trend differs. As the block size increases below 400, SmartSched exhibits rapid throughput growth, achieving up to 102.9k TPS at a block size of 400, which is 12.7× over Serial and 1.7× over Block-STM. However, after reaching that point, the throughput continues to grow at a slower rate. Nevertheless, SmartSched still outperforms other schemes, achieving up to 223.4k TPS at block size 10,000, which corresponds to 11.6× over Serial, 3.7× over Spectrum, and 1.6× over Block-STM.

Contention Level. Next, we evaluate performance under varying contention levels using the ERC-20 workload with 32 threads and a block size of 1,000. To control contention levels, we adjust the total number of accounts, which is inversely proportional to the contention level. Figure 7 illustrates the relationship between contention level and throughput. Similar to the trend observed in block size vs. throughput, SmartSched's throughput increases rapidly as the number of accounts grows, reaching 108.4k TPS at 64 accounts, which is 10.1× over Serial and 2.0× over ParallelEVM. After 64 accounts, the speedup stabilizes, peaking at 135.8k TPS at 1024 accounts, which is 11.5× over Serial and 1.5× over ParallelEVM. SmartSched outperforms other schemes in most cases, except when the number of accounts is less than 16. In this scenario, ParallelEVM achieves higher throughput (≥68.0k TPS), while SmartSched achieves up to 45.5k TPS.

Speedup Distribution. Finally, we present the cumulative distribution of per-block speedups over Serial execution for the history workload in Figure 8. SmartSched delivers highest speedups across nearly all percentiles. At the 25th, 50th (median), and 75 percentiles, SmartSched achieves speedups of 3.4×, 4.4×, and 5.5×, respectively. In comparison, Spectrum achieves 2.3×, 2.9×, and 3.7×, and ParallelEVM achieves 2.1×, 2.8×, and 3.5×. Notably, 10% of blocks are accelerated exceeding 6.7× with SmartSched.

Discussion. In general, SmartSched outperforms all baselines in both performance and scalability. An exception arises in extremely high-contention scenarios (≤16 accounts in Figure 7), where ParallelEVM achieves higher performance due to its partial, fast re-execution mechanism, which is tailored for high-contention workloads. Thus, ParallelEVM gains a relative advantage in such cases. For most schemes, the ERC-20 workload yields greater speedup than the history workload. The main reason is that the ERC-20 workload exhibit a uniform access pattern, whereas the history workload is skewed with hotspot states [38], which limits parallelism. Overall, general-purpose concurrency control methods (OCC, 2PL) exhibit poor performance. However, OCC shows some improvements in the history workload. This is mainly because many transactions in the history workload involve only native token transfers without smart contract execution. In this scenario, our commutative

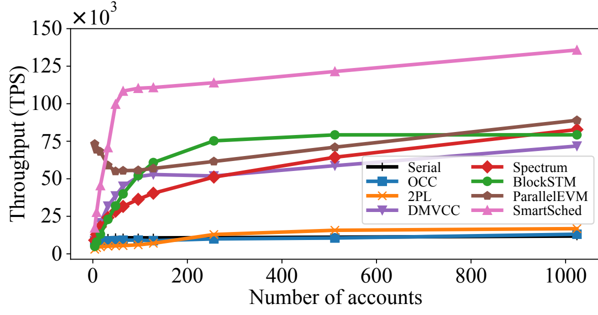


Figure 7: Speedups under different contention levels.

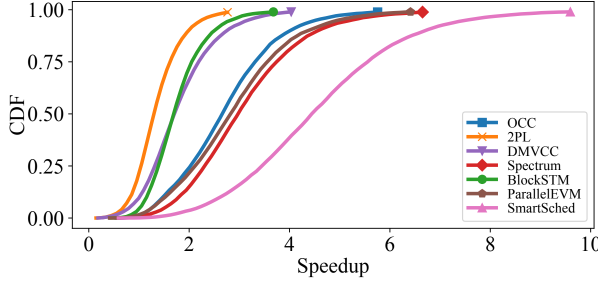


Figure 8: Cumulative distribution of per-block speedups.

operations ensure that such transactions do not generate conflicts, further improving OCC's performance.

6.4 Stand-Alone Performance

We further conduct ablation studies to evaluate the stand-alone performance of our instruction-level scheduling and pre-commit mechanism using the ERC-20 workload (block size: 1,000). Specifically, we examine two additional variants: SmartSched† and SmartSched*. SmartSched† enables instruction-level scheduling but disables pre-commit, meaning that waiting transactions can only be released after commit of predecessor transactions. SmartSched* implements only the pre-commit mechanism but stalls the thread instead of switching to another ready transaction upon detecting dependencies. To quantify the relative improvements, we define the relative speedup ($R^{\dagger}$ or R^{*}) as the ratio of SmartSched's speedup to that of each corresponding variant. As Figure 9 illustrates, both stand-alone and overall performance improve monotonically as the number of accounts increases. However, the relative improvements (R values) initially increase and subsequently decrease, peaking when the number of accounts approaches the number of threads. At 8 threads–8 accounts, SmartSched†, SmartSched*, and SmartSched achieve speedups of 1.5×, 1.2×, and 3.7×, respectively, resulting in an $R^{\dagger}$ of 2.5× and an R^{*} of 3.2×. At 32 threads–48 accounts, The speedups are 2.8×, 2.6×, and 9.4×, yielding 3.3× $R^{\dagger}$ and 3.6× R^{*}. In boundary cases, the relative improvements are less significant. For example, At 8 threads–128 accounts, $R^{\dagger}$ and R^{*} are 1.0× and 1.3×; at 32 threads–4 accounts, $R^{\dagger}$ and R^{*} are 1.3× and 1.4×.

Discussion. The combined solution provides greater performance benefits than applying each optimization independently, particularly when the number of accounts aligns with the number of threads, achieving the "1 + 1 > 2" effect. However, the relative improvement is lower when the number of accounts is either very

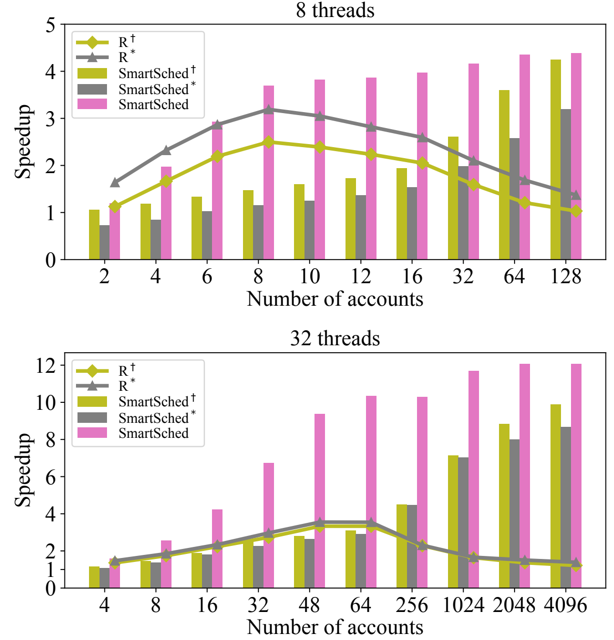


Figure 9: Stand-alone performance of SmartSched.

small or very large. This performance degradation is primarily due to the following reasons: The extremely small account set creates a high-contention scenario where nearly all transactions together form a sequential dependency chain. Under such conditions, pre-committed writes seldom help resolve dependencies earlier.. Furthermore, there are few ready transactions, limiting the effectiveness of instruction-level scheduling. In contrast, an extremely large account set provides a low-contention scenario with minimal dependencies, allowing most transactions to execute in conflict-free environments without requiring scheduling and pre-commit.

6.5 Overhead Analysis

A historical block has an average size of 184.5 KB, while the aggregated write sets across all its transactions occupy 39.3 KB. As shown in Table 2, a historical transaction has an average size of 563.6 bytes and performs 3.1 write operations, whose average write-set size is 121.3 bytes. Similarly, each ERC-20 transaction has an average size of 277.0 bytes and performs 2.0 write operations, with an average write-set size of 92.0 bytes. Consequently, recording write sets incurs an additional overhead of 21.5% for the historical workload and 33.2% for the ERC-20 workload in terms of storage and network cost. Moreover, the Number of Reads indicates that a replica performs dependency checks 2.6 times per historical transaction and 2.0 times per ERC-20 transaction during execution.

Then, we conduct a breakdown analysis to quantify the replica-side runtime overhead. Except for transaction execution (Exec.) , SmartSched introduces four additional stages: validation (Val.), commit (Comm.), dependency management (Dep.), and context switching (CtxSw). Dependency management includes both detection and resolution (Lines 2–6, 8–13 in Algorithm 2), The cost of context switching is measured as the time between when the *scheduler* starts to select a ready transaction (Line 5 in Algorithm 1)

Table 2: Statistics of read/write sets per transaction.

Workload	# of Reads	# of Writes	Transaction Size*	Write Set Size*
History	2.6	3.1	563.6 Bytes	121.3 Bytes (21.5%)
ERC-20	2.0	2.0	277.0 Bytes	92.0 Bytes (33.2%)

*Sizes refer to bytes after Recursive Length Prefix (RLP) serialization.

and when the chosen transaction begins or resumes execution. The total overhead is thus given by $Val. + Comm. + Dep. + CtxSw$.

We measure the time spent in each stage for every transaction and aggregate the results by stage. Table 3 presents the average cost breakdown for processing a block of 1,000 ERC-20 transactions under different configurations. Compared to the total cost, the execution time remains the dominant factor, ranging from 183.5 ms (86.5%) to 268.3 ms (65.6%) across all settings, while the total overhead accounts for 28.6 ms (13.5%) to 140.7 ms (34.4%). Validation and commit together contribute only a minor portion (6.4 ms on average, less than 3%). Thus, despite their sequential nature, their impact on end-to-end performance is negligible. Similarly, dependency management incurs an average overhead of 5.8 ms (<3%). In contrast, context switching constitutes the largest portion of the overhead, ranging from 62.6% (17.9/28.6) to 90.6% (127.5/140.7).

Discussion. For the history workload, the read set is smaller than the write set. This is because many historical transactions involve commutative operations (e.g., native-token transfers), which contribute only to the write set without increasing the read set. In contrast, ERC-20 transactions access contract storage exclusively through `SLOAD` and `SSTORE`, resulting in read and write sets of equal size. Context switching dominates the runtime overhead at replicas, particularly with many threads or in high-contention scenarios. The underlying reasons are as follows: SmartSched relies on Golang's goroutine scheduler to perform context switching. As the number of threads increases or contention intensifies, the goroutine scheduler becomes increasingly busy, leading to higher switching costs. Thus, future work may address this limitation by designing a more efficient context switching mechanism tailored to high-frequency scheduling, such as manually managing context switching instead of relying on the language runtime or OS to further enhance SmartSched's performance.

7 Limitations and Future Work

SmartSched faces two limitations. First, it requires write sets as prior knowledge, which introduces certain overhead. However, we argue that this overhead is acceptable. Compared to previous dependency-guided approaches that rely on complete read/write sets, SmartSched only requires write sets, effectively reducing storage and network overhead. Moreover, these write sets are naturally collected by miner nodes during block construction and can be shared with minimal additional effort. As shown in Section 6.5, a historical block produces only 39.3 KB of write-set data. This additional cost is modest relative to the performance benefits and remains fully acceptable for modern network and storage systems.

In practice, real-world blockchain systems have already incorporated transaction access information into blocks to enable various optimizations (not limited to parallel execution). For example, Ethereum's Access List Transaction (EIP-2930 [1]) requires each

Table 3: Processing time breakdown (ms) of SmartSched.

(Threads, Accounts)	Exec.	Val.	Comm.	Dep.	CtxSw	Overhead
(8, 4)	208.2	3.1	2.3	6.3	63.3	75.0 (26.5%)
(8, 16)	189.0	3.5	2.7	5.1	21.6	32.9 (14.8%)
(8, 64)	183.5	3.6	2.9	4.2	17.9	28.6 (13.5%)
(32, 16)	268.3	3.5	2.7	6.9	127.5	140.7 (34.4%)
(32, 64)	264.5	3.8	3.2	6.7	44.0	57.8 (17.9%)
(32, 256)	248.0	3.8	3.2	5.6	27.2	39.8 (13.8%)

transaction to explicitly declare the keys of the states it intends to access (corresponding to the `WKeys` in our method), enabling clients to prefetch data from disk before execution. Solana [64] requires transactions to explicitly declare their read/write sets, enabling the runtime to execute non-conflicting transactions in parallel. This trend demonstrates that exposing read/write intentions at the protocol level is feasible in production environments and aligns with real-world deployments. SmartSched can be naturally built on top of such existing systems rather than starting from scratch.

Second, replica nodes rely on the miner to share accurate write sets. If a malicious miner deliberately provides incorrect or incomplete write sets (as discussed in Section 5.3), the performance of replica nodes may be compromised. In the worst case, validation failures may cascade across transactions, forcing sequential re-executions and resulting in performance even worse than serial execution. Nevertheless, we argue that such misbehavior is economically disincentivized in practice. As discussed in Section 3.2, the throughput of a blockchain system is constrained by the slowest participating nodes, including replicas. Thus, attacking or slowing down replicas leads to fewer transactions being processed, which in turn reducing the miner's own rewards.

To further mitigate these limitations, future work may explore two directions: (1) designing more efficient techniques for write set extraction and dissemination with low overhead and strong correctness guarantees; and (2) establishing practical incentive and punishment mechanisms to encourage miners to provide accurate write sets, thereby aligning individual and network-wide benefits.

8 Conclusion

This paper presents SmartSched, a fine-grained framework for parallel smart contract execution at replicas. By introducing the *instruction-level scheduling* and the *multi-layer versioned state*, our approach parallelizes non-conflicting instructions across transactions. thereby enhancing parallelism and execution efficiency. Experimental results demonstrate that SmartSched achieves substantial throughput improvements (1.6× to 11.6×) over baselines.

Acknowledgments

We thank all reviewers and the shepherd for their invaluable comments and support. This work is partially supported by the National Natural Science Foundation of China (NSFC) under Grant 62172360, HK Innovation and Technology Commission (ITC) under Scheme MHP/135/23, and NSFC under Grant U21A20467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

[1] [n. d.]. EIP-2930: Optional access lists. https://eips.ethereum.org/EIPS/eip-2930. [Online; accessed April-2026].

[2] [n. d.]. ERC-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20. [Online; accessed April-2026].

[3] [n. d.]. Go-Ethereum (Geth) client. https://github.com/ethereum/go-ethereum/. [Online; accessed April-2026].

[4] [n. d.]. Merkle Patricia Trie. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/. [Online; accessed April-2026].

[5] [n. d.]. Monad: Parallel Execution. https://docs.monad.xyz/monad-arch/execution/parallel-execution. [Online; accessed April-2026].

[6] [n. d.]. Sei v2 - The First Parallelized EVM Blockchain. https://blog.sei.io/sei-v2-the-first-parallelized-evm/. [Online; accessed April-2026].

[7] [n. d.]. The Solidity Programming Language. https://soliditylang.org/. [Online; accessed April-2026].

[8] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM* 61, 9 (2018), 78–88.

[9] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[10] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[11] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2024. Optsmart: a space efficient opt imistic concurrent execution of smart contracts. *Distributed and Parallel Databases* 42, 2 (2024), 245–297.

[12] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2025. Mysticeti: Reaching the Latency Limits with Uncertified DAGs.. In *NDSS*.

[13] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[14] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 185–221.

[15] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[16] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2189–2207.

[17] 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 570–587.

[18] Zhihao Chen, Xiaodong Qi, Xiaofan Du, Zhao Zhang, and Cheqing Jin. 2021. Peep: A parallel execution engine for permissioned blockchain systems. In *Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part III 26*. Springer, 341–357.

[19] Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. Spectrum: Speedy and Strictly-Deterministic Smart Contract Transactions for Blockchain Ledgers. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2541–2554.

[20] José Eduardo de Azevedo Sousa, Vinícius Oliveira, Júlia Valadares, Glauber Dias Goncalves, Saulo Moraes Villela, Heder Soares Bernardino, and Alex Borges Vieira. 2021. An analysis of the fees and pending time correlation in Ethereum. *International Journal of Network Management* 31, 3 (2021), e2113.

[21] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 303–312.

[22] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201. doi:10.14778/2809974.2809981

[23] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High performance transactions via early write visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624. doi:10.14778/3055540.3055553

[24] Yaozheng Fang, Zhiyuan Zhou, Surong Dai, Jinni Yang, Hui Zhang, and Ye Lu. 2023. Pavm: A parallel virtual machine for smart contract execution and validation. *IEEE Transactions on Parallel and Distributed Systems* 35, 1 (2023), 186–202.

[25] Hang Feng, Yufeng Hu, Yinghan Kou, Runhuai Li, Jianfeng Zhu, Lei Wu, and Yajin Zhou. 2024. {SlimArchive}: A Lightweight Architecture for Ethereum Archive Nodes. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 1257–1272.

[26] Cuifeng Gao, Ao Chen, Chengze Wu, Wenzhang Yang, Jiaming Ye, and Yinxing Xue. 2025. Are Static Analysis Tools Still Working during the Evolution of Smart Contracts? A Comprehensive Empirical Study. *ACM Transactions on Software Engineering and Methodology* (2025).

[27] 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 *Proceedings of the 44th International Conference on Software Engineering*. 2315–2326.

[28] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 232–244.

[29] Rong Han, Zheng Yan, Xueqin Liang, and Laurence T Yang. 2022. How can incentive mechanisms and blockchain benefit with each other? a survey. *Comput. Surveys* 55, 7 (2022), 1–38.

[30] Yaron Hay and Roy Friedman. 2024. Batch-schedule-execute: on optimizing concurrent deterministic scheduling for blockchains. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 163–174.

[31] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. 2021. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering* 34, 11 (2021), 5070–5083.

[32] Dodo Khan, Low Tang Jung, and Manzoor Ahmed Hashmani. 2021. Systematic literature review of challenges in blockchain scalability. *Applied Sciences* 11, 20 (2021), 9372.

[33] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[34] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When private blockchain meets deterministic database. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–28.

[35] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. 2024. LVMT: An efficient authenticated storage for blockchain. *ACM Transactions on Storage* 20, 3 (2024), 1–34.

[36] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 2018. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870* (2018).

[37] Chenxin 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 *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 515–528.

[38] Haoran Lin, Hang Feng, Yajin Zhou, and Lei Wu. 2025. ParallelEVM: Operation-Level Concurrent Transaction Execution for EVM-Compatible Blockchains. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys '25)*.

[39] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proc. VLDB Endow.* 13, 12 (July 2020), 2047–2060. doi:10.14778/3407790.3407808

[40] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[41] Bernhard K Meister and Henry CW Price. 2024. Gas fees on the Ethereum blockchain: from foundations to derivative valuations. *Frontiers in Blockchain* 7 (2024), 1462666.

[42] Ahmed Afif Monrat, Olov Schelén, and Karl Andersson. 2019. A survey of blockchain from the perspectives of applications, challenges, and opportunities. *Ieee Access* 7 (2019), 117134–117151.

[43] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented transaction execution. *Proceedings of the VLDB Endowment* 3, 1 (2010), 928–939.

[44] Michele Pasqua, Andrea Benini, Filippo Contro, Marco Crosara, Mila Dalla Preda, and Mariano Ceccato. 2023. Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode. *Journal of Systems and Software* 200 (2023), 111653.

[45] Xiaodong Qi, Jiao Jiao, and Yi Li. 2023. Smart contract parallel execution with fine-grained state accesses. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 841–852.

[46] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2017. Scalable replay-based replication for fast databases. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2025–2036.

[47] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 180–194.

[48] Iqra Sadia Rao, ML Mat Kiah, M Muzaffar Hameed, and Zain Anwer Memon. 2024. Scalability of blockchain: a comprehensive review and future research direction. *Cluster Computing* 27, 5 (2024), 5547–5570.

[49] Daniël Reijsbergen and Tien Tuan Anh Dinh. 2020. On exploiting transaction concurrency to speed up blockchains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1044–1054.

[50] Kun Ren, Alexander Thomson, and Daniel J Abadi. 2014. An evaluation of the advantages and disadvantages of deterministic database systems. *Proceedings of the VLDB Endowment* 7, 10 (2014), 821–832.

[51] 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 *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 543–557.

[52] Donghyeon Ryu and Chanik Park. 2024. Toward High-Performance Blockchain System by Blurring the Line between Ordering and Execution. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[53] Abdurrashid Ibrahim Sanka and Ray CC Cheung. 2021. A systematic review of blockchain scalability: Issues, solutions, analysis and future research. *Journal of Network and Computer Applications* 195 (2021), 103232.

[54] Vikram Saraph and Maurice Herlihy. 2019. An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:1901.01376* (2019).

[55] 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 *Proceedings of the 2019 International Conference on Management of Data*. 105–122.

[56] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 70–80.

[57] 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 *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[58] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 67–82.

[59] Hao Wang, Minghao Pan, and Jiaping Wang. 2025. Crystality: A Programming Model for Smart Contracts on Parallel EVMs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 412–425.

[60] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. 2023. Sok: Dag-based blockchain systems. *Comput. Surveys* 55, 12 (2023), 1–38.

[61] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[62] Jiang Xiao, Shijie Zhang, Zhiwei Zhang, Bo Li, Xiaohai Dai, and Hai Jin. 2022. Nezha: Exploiting concurrency for transaction processing in dag-based blockchains. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 269–279.

[63] Junfeng Xie, F Richard Yu, Tao Huang, Renchao Xie, Jiang Liu, and Yunjie Liu. 2019. A survey on the scalability of blockchain systems. *IEEE network* 33, 5 (2019), 166–173.

[64] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain v0. 8.13.

[65] Haowen Zhang, Jing Li, He Zhao, Tong Zhou, Nianzu Sheng, and Hengyu Pan. 2023. BlockPilot: A Proposer-Validator Parallel Execution Framework for Blockchain. In *Proceedings of the 52nd International Conference on Parallel Processing*. 193–202.

[66] Qiheng Zhou, Huawei Huang, Zibin Zheng, and Jing Bian. 2020. Solutions to scalability of blockchain: A survey. *Ieee Access* 8 (2020), 16440–16455.