Seer: Accelerating Blockchain Transaction Execution by Fine-Grained Branch Prediction

Shijie Zhang
HUST[†]
shijiezhang@hust.edu.cn

Ru Cheng
HUST[†]
ruc@hust.edu.cn

Xinpeng Liu
HUST[†]
xpliu@hust.edu.cn

Jiang Xiao[*]
HUST[†]
jiangxiao@hust.edu.cn

Hai Jin
HUST[†]
hjin@hust.edu.cn

Bo Li
HKUST[‡]
bli@cse.ust.hk

ABSTRACT

Increasingly popular *decentralized applications* (dApps) with complex application logic incur significant overhead for executing smart contract transactions, which greatly limits public blockchain performance. Pre-executing transactions off the critical path can mitigate substantial I/O and computation costs during execution. However, pre-execution does not yield any state transitions, rendering the system state inconsistent with actual execution. This inconsistency can lead to deviations in pre-execution paths when processing smart contracts with multiple state-related branches, thus diminishing pre-execution effectiveness. In this paper, we develop Seer, a novel public blockchain execution engine that incorporates fine-grained branch prediction to fully exploit pre-execution effectiveness. Seer predicts state-related branches using a two-level prediction approach, reducing inconsistent execution paths more efficiently than executing all possible branches. To enable effective reuse of pre-execution results, Seer employs checkpoint-based fast-path execution, enhancing transaction execution for both successful and unsuccessful predictions. Evaluations with realistic blockchain workloads demonstrate that Seer delivers an average of 27.7× transaction-level speedup and an overall 20.6× speedup in the execution phase over vanilla Ethereum, outperforming existing blockchain execution acceleration solutions.

PVLDB Reference Format:
Shijie Zhang, Ru Cheng, Xinpeng Liu, Jiang Xiao, Hai Jin, and Bo Li. Seer: Accelerating Blockchain Transaction Execution by Fine-Grained Branch Prediction. PVLDB, 18(3): 822 - 835, 2024.
doi:10.14778/3712221.3712245

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CGCL-codes/SeerEVM.

[*]Corresponding author.
[†]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.
[‡]Department of Computer Science and Engineering, the Hong Kong University of Science and Technology, Hong Kong SAR.
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. 18, No. 3 ISSN 2150-8097.
doi:10.14778/3712221.3712245

1 INTRODUCTION

Modern blockchains have evolved from Bitcoin's [31] focus on peer-to-peer currency transactions to utilizing smart contracts written in Turing-complete languages to express more complex application logic [7, 40, 51, 52]. This shift has catalyzed innovation in *decentralized applications* (dApps), including *decentralized finance* (DeFi) [43], *non-fungible tokens* (NFTs) [42], and *decentralized identity* (DID) [29].

Ethereum [44] exemplifies an advanced public blockchain platform supporting a wide range of dApps, with smart contract transactions comprising nearly 70% of its load [32]. Ethereum processes transactions using the *Dissemination-Consensus-Execution* (DiCE) model [8], where transactions are first disseminated through P2P networks, then added to newly generated blocks via consensus among nodes (consensus phase), and finally executed by each node using the *Ethereum Virtual Machine* (EVM) [44] to achieve a consistent state (execution phase). The next consensus round cannot start until all state transitions in the current execution phase are completed. Therefore, the throughput depends heavily on the block generation interval and the number of transactions processed within it [32, 49]. Recent EVM-compatible DiCE blockchains try to improve throughput by either increasing on-chain transaction volume (e.g., larger blocks [4], DAG structures [3, 27, 47, 48], sharding schemes [19, 20, 35, 50]) or shortening block generation intervals [4, 38]. However, increasing transaction volume significantly raises EVM execution latency, which in turn lengthens block generation intervals. Consequently, performance remains limited by EVM execution, especially for smart contracts. Addressing this bottleneck would enable higher transaction volumes without prolonging block generation intervals, improving overall throughput.

Pre-execution is a promising technique to accelerate blockchain transaction execution [2, 8, 32]. By leveraging the time window during the dissemination and consensus phases, nodes can pre-execute transactions to offload I/O and computation costs from the critical path and fetch the read-write set to facilitate concurrency control when enabling concurrent execution. However, when dealing with smart contract transactions featuring increasingly complex logic, the effectiveness of pre-execution is not fully realized. This limitation arises from numerous state variable-related branch conditions within smart contracts. Since pre-execution is based on the state snapshot committed in the previous block generation interval without any state transitions, the state variable versions fetched during pre-execution might differ from those during actual execution, leading to divergent execution paths. For instance, in an auction dApp,

a state variable *highestBid* tracks the leading bid. At the end of each bidding round, if *highestBid* exceeds the reserve price, the auction is finalized; otherwise, the bidding period is extended, resulting in different dApp logics to execute. As a result, cached execution results on pre-execution paths inconsistent with actual execution are difficult to reuse. Additionally, the read-write set obtained during pre-execution may change during actual execution, making it difficult to accurately detect conflicts in concurrency control.

Recent efforts to optimize blockchain transaction execution performance have overlooked the limitations of pre-execution, particularly under the multi-branch characteristics of modern smart contracts. Forerunner [8] and MTPU [32] represent state-of-the-art pre-execution approaches aimed at enhancing the utility of cached pre-execution results. Forerunner speculates multiple potential execution contexts for each transaction and caches pre-execution results for all possible paths caused by branches. However, given the complexity of modern smart contracts, pre-executing all possible paths and caching their execution results result in excessive execution and storage costs. Conversely, MTPU pre-executes only constant operations in contracts, leaving state variable-related operations to be executed on the critical path. Existing concurrency control schemes [15, 16, 34] typically use pre-computed or statically analyzed read-write sets for conflict detection. However, these read-write sets may not align with the actual read-write operations, leading to false negatives in conflict detection, thereby increasing the transaction abort rate (as detailed in Section 2.3).

Processing modern smart contract transactions necessitates an advanced pre-execution technique that can fully harness its potential. In this paper, we propose Seer, a novel execution engine for EVM-compatible blockchains that enhances the effectiveness of pre-execution. The core of Seer is to predict branches that lead to different execution paths, thereby increasing the proportion of reusable pre-execution results and improving the accuracy of read-write sets. The initial step involves identifying branch conditions related to contract state variables. Since the contract source code is unavailable on-chain, nodes rely on the bytecode stored in the blockchain state for decompilation analysis. However, decompilation inaccuracies pose a challenge to accurate branch identification. Seer addresses this issue through a stack tracing-based approach, which labels all opcodes relevant to state variables occurring in the native EVM execution stack. When an opcode comparing a state variable's value is encountered, the corresponding branch condition information is stored in a branch table maintained by Seer.

To ensure accurate branch prediction, Seer employs a two-level branch prediction mechanism during pre-execution. The idea is to simulate the transaction execution order expected in actual execution. Since pre-execution does not modify the system state, we utilize a multi-version cache to record the write versions of state variables. When encountering state variable-related branch conditions, relatively accurate state variable versions can be acquired to determine branch directions. To mitigate the overhead of caching and fetching multi-version states, we preserve recent branch direction histories in the branch table. If a branch's direction history shows a regular pattern, we can skip branch condition evaluation and use an optimized perceptron model [22] for lightweight prediction. For branches with irregular direction histories, we continue using the multi-version state-based prediction. This two-level prediction mechanism achieves efficient and accurate branch prediction.

To maximize the reuse of pre-execution results, Seer creates a checkpoint snapshot before each state variable-related branch condition to cache the pre-execution results prior to the condition. During actual execution, Seer re-evaluates each branch condition to decide whether to reuse checkpoint snapshots. If the predicted branch direction is accurate, the cached snapshot can be used for fast-path execution. In cases where the prediction is inaccurate, execution can resume from the checkpoint rather than starting over. Checkpoint snapshots also facilitate efficient pre-execution repair and re-execution of aborted transactions. Additionally, Seer outputs relatively accurate read-write sets post-pre-execution, compatible with existing concurrency control schemes.

We have implemented Seer on top of Geth [14], the official Go implementation of Ethereum, and compared it with state-of-the-art blockchain pre-execution [30, 32] and concurrency control [15] approaches. Evaluations with realistic Ethereum workloads show that Seer achieves an overall 20.6× speedup in the execution phase over vanilla Ethereum, surpassing all compared approaches.

The main contributions of our work are summarized as follows.

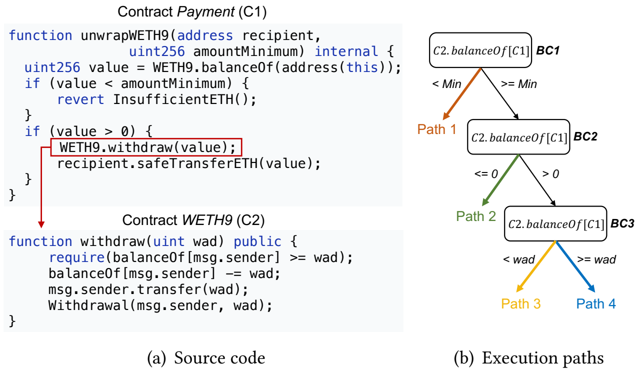
- We investigate the limitations of transaction pre-execution and concurrent execution under the multi-branch characteristics of modern smart contracts, and propose Seer, a novel public blockchain execution engine incorporating fine-grained branch prediction for effective pre-execution.
- We propose a stack tracing-based branch identification approach to accurately identify state variable-related branch conditions and a two-level branch prediction mechanism to ensure efficient and accurate branch prediction.
- We develop a checkpoint-based fast-path execution mechanism to maximize the utilization of cached pre-execution results, reducing the overhead of actual execution.
- We implement a prototype of Seer and evaluate it, demonstrating that it outperforms state-of-the-art approaches in speedup performance under realistic benchmarks.

2 BACKGROUND AND MOTIVATION

2.1 Blockchain State

In public blockchain systems that support smart contracts, like Ethereum, the current states of all accounts on the blockchain constitute the *world state* [25, 28], which is a mapping between account addresses and account states [44] and stored as key-value pairs in a *Merkle Patricia Trie* (MPT) [13]. For the contract account state, in addition to storing balance and nonce information, it also contains the contract bytecode hash and the root hash of an MPT that encodes the *contract storage*, permanently storing state variables within the contract. The *contract storage* allocates a specific slot for each state variable to store its value, with each slot capable of storing up to a 256-bit variable. When variable sizes are less than 256 bits, storage compaction occurs, i.e., multiple variables are stored together in the same slot until the 256-bit storage space is filled [10].

The *world state* transition is triggered by the *Ethereum Virtual Machine* (EVM) [44], a quasi-Turing complete machine instance held by each node. When a transaction invokes a smart contract, the EVM loads the contract bytecode and divides it into multiple



(a) Source code (b) Execution paths

Figure 1: An example of a multi-branch smart contract

opcodes and operands that the EVM can interpret for execution. The EVM uses a stack-based architecture for executing opcodes and storing operands, during which it can interact with the *contract storage* and the *memory* storing temporary data. Each state variable is accessed using specific opcodes like SLOAD (read the value from the *contract storage*) and SSTORE (write the updated value to the *contract storage*). To prevent the abuse of computational resources, a fee is charged to the caller for each contract execution, denominated in *gas*, which depends on the complexity of the contract [12].

2.2 Multi-Branch Smart Contracts

Modern smart contracts often contain numerous branches that create various execution paths to accommodate different application logics. To illustrate the multi-branch characteristics of smart contracts, we examine the widely used *Uniswap: Universal Router*, one of the top three most active contracts on the Ethereum platform. Specifically, we analyze the *unwrapETH9* function within the *Payment* sub-contract (C1), which aims to swap all of the contract's WETH tokens for ETH. As shown in Figure 1(a), this function internally calls another contract *WETH9* (C2) to perform the withdrawal operation. During the execution of *unwrapETH9*, there are three *branch conditions* (BCs) that determine which branches to execute. The first two BCs appear within the local function, while the third one appears in the internally called contract C2. Figure 1(b) depicts the four possible execution paths (with black lines indicating the shared paths) resulting from the three BCs. All three BCs evaluate the balance variable value (i.e., *balanceOf(C1)*) stored in the C2, which is used to determine whether the C1 has enough WETH for the swap. Depending on the variable values, the evaluation results of these state variable-related BCs (abbreviated as *SV-conditions*) may vary, leading to different branches (abbreviated as *SV-branches*) and altering the execution paths.

Notably, only *SV-conditions* can lead to varying execution paths on *SV-branches* under different *world states*. The remaining BCs are related to constants, e.g., checking transaction input values, which are independent of the *world state*. To verify the presence of *SV-conditions* in a broader range of contract transactions, we replayed realistic Ethereum transaction data in 1,000 consecutive blocks starting from block height 14M, consistent with the data used in Section 5, to record *SV-conditions* during replay. As shown in Table 1, although the proportion of *SV-conditions* is relatively low

Table 1: Ethereum branch statistics across 1,000 consecutive block heights

Block height offset	200	400	600	800	1000
Ratio of *SV-conditions*	12%	11%	12%	11%	15%
# of *SV-conditions* per TX	4.3	3.8	4.2	4	4.2
Ratio of regular *SV-branches*	13%	21%	23%	23%	11%

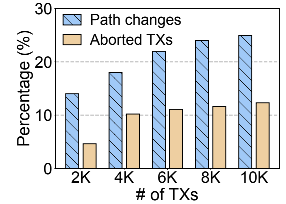


Figure 2: Percentage of varying execution paths and aborted transactions

(ranging from 11% to 15%) across five block height intervals, each contract transaction still contains an average of 4.1 *SV-conditions*. This indicates that executing the same contract transaction under different *world states* may yield inconsistent results.

2.3 Challenges of Pre-Execution

Pre-execution schemes in public blockchains typically run in parallel with consensus [8, 32], where nodes pre-execute pending transactions from the network to cache their execution results and read-write sets. During pre-execution, transactions are executed based on the same committed *world state* due to the absence of a determined execution order, differing from the sequential *world state* updates in an agreed-upon order during actual execution. This discrepancy can cause pre-execution and actual execution to take divergent branches when processing multi-branch smart contract transactions, leading to two issues that undermine the effectiveness of pre-execution. We conducted experiments to quantify these impacts, using the same dataset mentioned above (Section 2.2). To further simulate the high consensus throughput of current EVM-compatible blockchains, we pre-execute transactions with increasingly larger scales.

1) Impact the proportion of reusable pre-execution results. Continuing with the smart contract shown in Figure 1 as an example, suppose the fetched value of *balanceOf(C1)* is V_1 during pre-execution, which is less than $amountMinimum$, leading to the execution of *path 1*. However, the fetched value of *balanceOf(C1)* is V_2 during actual execution, triggering the execution of *path 4* instead. This renders the pre-executed path unusable for the subsequent execution. Figure 2 presents the proportion of transactions whose execution paths change between pre-execution and actual execution, which increases as the number of pre-executed transactions rises since more transactions read stale state versions to determine *SV-branch* directions during pre-execution.

2) Impact the accuracy of conflict detection in concurrency control. In addition to execution paths, the read-write sets obtained during pre-execution may also change, leading to false negatives and positives in conflict detection. For instance, two transactions calling the function *unwrapETH9* may read the same value of *balanceOf(C1)* during pre-execution, both taking the *path 1* without generating write sets, thus not conflicting. However, if the value of *balanceOf(C1)* changes during concurrent execution, both transactions might follow the *path 4*, causing a write-write conflict (i.e., false negative) and necessitating one transaction's abortion for serializability. We investigate the abort rate of OCC-DA [15], one of the state-of-the-art blockchain MVCC schemes, which only aborts non-serializable transactions caused by false negatives. Figure 2

shows that the abort rate grows with the increased number of pre-executed transactions due to the rise in path changes. Conversely, the false positive case does not cause transaction aborts yet misidentifies non-conflicting transactions as conflicting, affecting execution efficiency by enforcing unnecessary serialization.

2.4 Our Approach

The key to resolving the above two issues is to increase the alignment between pre-execution paths and actual execution paths. By leveraging the characteristic that inconsistent execution paths only occur on *SV-branches*, we innovatively propose *fine-grained branch prediction*, which predicts *SV-branches* to be executed rather than executing all possible branches at a high cost. The fundamental principle is to predict the values of state variables in *SV-conditions* to decide which *SV-branches* to execute.

Accurate identification of all *SV-conditions* is a prerequisite for predicting *SV-branches*. We utilize the native EVM execution stack to trace relevant opcodes and operands during pre-execution, avoiding reliance on contract source code or decompilation tools. To fetch accurate values of state variables in *SV-conditions*, we simulate the total order of transactions in the upcoming block to guide the pre-execution order. Moreover, we devise a *multi-version cache* to store all updated versions of state variables in *SV-conditions* during pre-execution, making them visible to each transaction.

In realistic contract transactions, some *SV-branches* follow regular directional patterns over time, as shown in Table 1. Such *SV-branches* can be predicted using lightweight learning algorithms without querying state variable values from the *multi-version cache*. We employ and optimize the *perceptron* model, commonly used in CPU branch prediction [22], to predict regular *SV-branches*, mitigating the read and write overhead towards the *multi-version cache*. By integrating both prediction methods, we design a *two-level prediction* mechanism that can maximize their combined strengths.

To ensure that accurately predicted *SV-branches* can be reused, we establish a *checkpoint snapshot* of the current execution state before each *SV-condition*. If all predicted *SV-branches* of a transaction are satisfied, the cached pre-execution results in the *checkpoint snapshot* can be used to skip the native EVM execution. Furthermore, *checkpoint snapshots* can also enable transactions with failed predictions to resume their executions without restarting from scratch. Additionally, by leveraging read-write sets from accurate pre-execution paths, our approach can help reduce transaction abort rates for mainstream blockchain concurrency control schemes.

3 SEER DESIGN

3.1 The Architecture of Seer

Seer is built as a modular execution engine that can be integrated into the current Ethereum system. Figure 3 depicts the architecture of Seer, which comprises three major components: the *transaction ordering simulator* and the *branch predictor* for the pre-execution phase, and the *execution scheduler* for the actual execution phase.

During the pre-execution phase, the *transaction ordering simulator* monitors the transaction pool storing network-received transactions to extract those likely to be included in the next block and simulate their execution order. Then, Seer takes the ordered

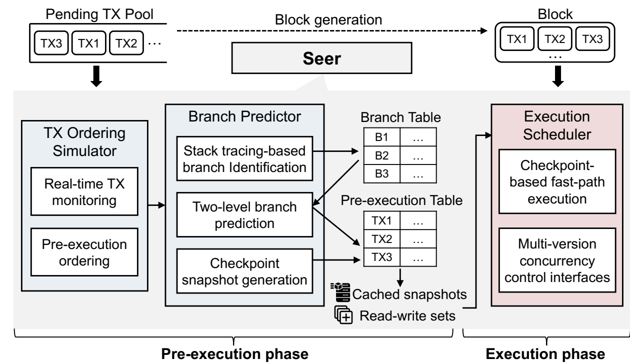


Figure 3: Seer architecture

transactions as input and employs the *branch predictor* for transaction pre-execution. Specifically, it employs the stack tracing-based approach to identify all *SV-conditions* and the two-level branch prediction mechanism to accurately predict branch directions. The relevant *SV-condition* information is stored into the *branch table*, and the pre-execution result of each transaction is cached into the *pre-execution table*. Before branch prediction, the *branch predictor* also generates a checkpoint snapshot of execution results prior to each *SV-condition* for subsequent fast-path execution.

During the actual execution phase, the *execution scheduler* takes transactions from the newly generated block, along with cached checkpoint snapshots and read-write sets from the *pre-execution table* as input, to perform fast-path execution. To enhance the parallelism of execution, the *execution scheduler* also incorporates interfaces for modern *multi-version concurrency control* (MVCC) schemes, ensuring the serializability of concurrent transactions.

3.2 Transaction Ordering Simulation

The order in which transactions are received by nodes through network dissemination often differs from their actual execution order. To obtain relatively accurate values of state variables in *SV-branches* during pre-execution, it is crucial to first simulate the actual transaction execution order as closely as possible.

In Ethereum, the execution order of transactions in a block is primarily influenced by the gas fee [1] that each transaction initiator offers to pay [44], i.e., prioritizing transactions with higher gas fees over those with lower fees. Notably, many EVM-compatible blockchains follow similar transaction ordering rules [4, 27, 38]. Hence, we utilize this ordering rule to guide the pre-execution order of transactions. Specifically, during block creation, each node uses a *transaction ordering simulator* thread to sort transactions in the transaction pool by their gas fees. Then, based on the block capacity limit (i.e., gas limit), it sends the transactions likely to be included in the block to the *branch predictor* thread for pre-execution. Since each node makes its own transaction inclusion choice based on its view of the transaction pool, the transactions pre-executed by each node will differ, as discussed in Section 3.9.

During pre-execution, the transaction pool will receive some newly arrived transactions with high gas fees. In native Ethereum

[1] *Ethereum Improvement Proposal* (EIP)-1559, proposed in 2019, puts forward a new gas fee calculation model, where the total gas fee equals the sum of the base fee adjusted with block space demand and the priority fee paid extra by the transaction initiator.

block generation, if the gas fee of a newly arrived transaction meets the criteria for inclusion in the block, the block generation process needs to be restarted to prioritize it for inclusion in the current block. To better simulate the set of transactions and execution order of the actual generated block, the *transaction ordering simulator* will continuously monitor the transaction pool to detect newly arrived transactions with higher gas fees. If the gas fee of a newly arrived transaction is higher than that of some included transactions, the *transaction ordering simulator* will derive its appropriate position within the transaction set and inform the *branch predictor*, so that this transaction can be pre-executed preferentially.

3.3 Stack Tracing-Based Branch Identification

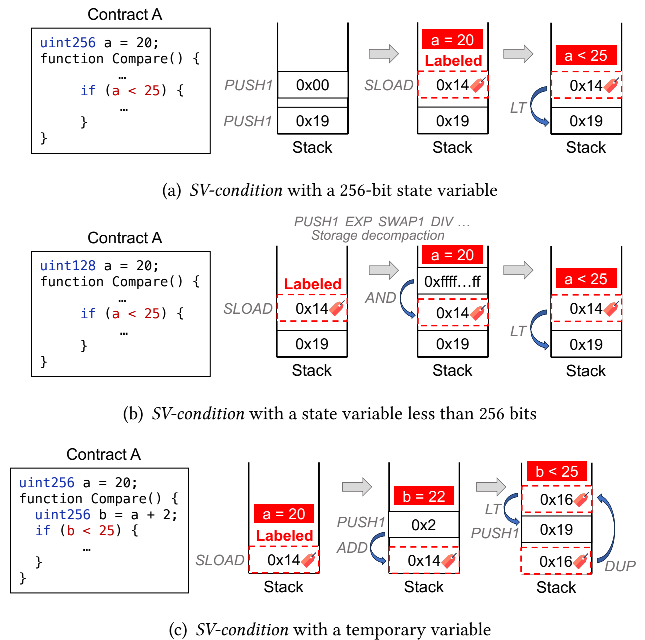
As the contract source code is not stored on-chain, decompiling the stored bytecode offers a way to derive useful execution details [5, 24] for identifying *SV-conditions*, yet it is costly and not entirely accurate. Instead, the native execution stack can be leveraged to trace opcodes related to *SV-conditions* during pre-execution.

During bytecode execution, specific types of branch statements, such as conditional or loop statements, cannot be directly identified from the stack. However, a common characteristic of *SV-conditions* within these statements is the presence of comparison opcodes applied to state variables. Our approach centers on labeling state variables fetched via the SLOAD opcode, which allows us to identify them when comparison opcodes are encountered. We utilize a dedicated structure to represent the labels of state variables, ensuring they remain identifiable even as computations or copy and move operations occur on the stack. Importantly, our method only performs essential tracing operations without interrupting stack execution and is privacy-safe, as stack operands reflect only the storage addresses and values of variables, not their names [10].

When a branch condition occurs, we can identify whether the involved elements are labeled to determine if it constitutes an *SV-condition*. However, branch identification is a non-trivial task, as variables in an *SV-condition* can manifest in three basic forms, as depicted in Figure 4. Below, we detail how our approach copes with these basic cases, as well as more complex scenarios.

❶ ***SV-condition* with a 256-bit state variable.** The opcodes of evaluating *SV-conditions* related to 256-bit state variables exhibit a fixed pattern on the stack, as shown in Figure 4(a). First, the top stack element, 0x00, representing the storage slot, is popped for the SLOAD operation, i.e., retrieving the value of the state variable *a* from the slot 0x00 and pushing 0x14 (20 in decimal) to the stack. Then, the comparison opcode LT is executed to evaluate if the top element value is smaller than the second element value. By labeling the element of 0x14 obtained by SLOAD, we can easily identify the *SV-condition* of $a < 25$ during the execution of LT.

❷ ***SV-condition* with a state variable less than 256 bits.** Unlike the first case, the opcodes of evaluating *SV-conditions* with state variables less than 256 bits require an additional storage decompaction step to derive the value of a specific variable. As presented in Figure 4(b), SLOAD is performed first to retrieve the compacted value 0x14 at the slot 0x00. Then, storage decompaction with a series of fixed opcodes occurs. At the end of storage decompaction, the value of *a* can be obtained through an AND operation with the mask code 0xff...f. By labeling the element of 0x14 at the beginning, we



(a) *SV-condition* with a 256-bit state variable

(b) *SV-condition* with a state variable less than 256 bits

(c) *SV-condition* with a temporary variable

Figure 4: Stack tracing-based branch identification under three different cases

can still recognize it as derived from a state variable after storage decompaction. This allows us to identify the *SV-condition* of $a < 25$ during the execution of the opcode LT. Besides, we can obtain the storage offset and bit-length of *a* during storage decompaction.

❸ ***SV-condition* with a temporary variable.** For branch conditions involving temporary variables derived from state variables, such as the temporary variable *b* shown in Figure 4(c), they also constitute *SV-conditions* and require identification, as the temporary variable value depends on the state variable value. We first label the element 0x14 obtained by SLOAD. When the labeled element undergoes a series of computations, its value changes while its label remains, indicating it has evolved from the state variable *a*. We also preserve a trace of all computations (i.e., opcodes and operands) operated on each labeled element to facilitate correct state transitions in subsequent fast-path execution, as detailed in Section 3.6. Finally, the element of *b*, together with its label, is copied to the top of the stack for comparison with $0x19$, allowing us to identify that the *SV-condition* of $b < 25$ is related to *a* through the label.

The three cases outlined above cover basic scenarios for evaluating a single state variable within an *SV-condition*, serving as the basis for identifying more complex *SV-conditions*. In complex *SV-conditions* containing multiple sub-conditions connected by AND or OR operations, each sub-condition can be treated as a separate *SV-condition* and identified individually. This is feasible because the evaluation result of each sub-condition will be returned to the stack, with the entire *SV-condition* then evaluated through logical operations on the stack. To store information about each identified *SV-condition* for reuse in future pre-execution, Seer maintains a *branch table* in memory, using the state variable's storage information as the key. A branch ID is used to exclusively identify *SV-conditions* under each key. Specifically, if the compared element

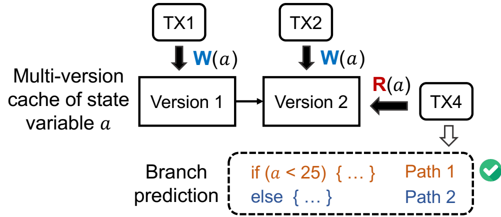


Figure 5: Multi-version state-based prediction during the pre-execution of $TX4$

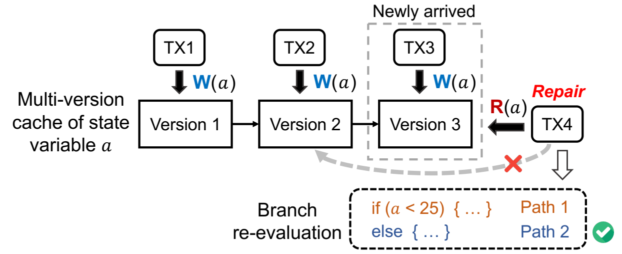


Figure 6: Pre-execution repair of $TX4$

is a constant, the branch ID is derived from the called function signature and the constant value. If both elements are state variables, only one variable's storage information is used as the key to avoid storage duplication, while the branch ID is computed based on the function signature and the other variable's storage slot and offset.

3.4 Two-Level Branch Prediction

Multi-version state-based prediction. The *transaction ordering simulator* ensures that transactions can be pre-executed in order according to the native transaction ordering rule. However, the pre-execution phase does not actually modify the *world state*, making state updates generated by sequential pre-execution invisible. To address this, we devise a *multi-version cache* for state variables to cache their different visible write versions during pre-execution. In the *multi-version cache* of a state variable, different write versions are linked by pointers, enabling flexible deletion and insertion of write versions. Each write version contains the newly written value along with the transaction details that perform the write operation. After the branch identification, nodes can fetch the latest write version of the state variable involved in the identified *SV-condition* for a relatively accurate evaluation. Since only state variables involved in *SV-conditions* are useful for predictions, the *multi-version cache* can store only necessary write versions to optimize memory usage.

Figure 5 continues with the example of the branch condition presented in Figure 4(a). During the pre-execution of $TX4$, the *branch predictor* first identifies the *SV-condition* of $a < 25$ and fetches the latest write version of a from its *multi-version cache*, i.e., the *Version 2* written by $TX2$. The *branch predictor* then evaluates whether the latest value of a is less than 25 and returns the evaluation result to the stack. Meanwhile, the evaluation results of all *SV-conditions* encountered by each transaction will be stored in an in-memory *pre-execution table* maintained by Seer, which records the pre-execution results of transactions for subsequent fast-path execution.

Lightweight pre-execution repair. As mentioned before, the *transaction ordering simulator* will occasionally send transactions with high gas fees and their positions to the *branch predictor*, which determines when to pre-execute them based on their positions. Specifically, if a newly arrived transaction ranks after all pre-executed ones, it is directly inserted into the queue of transactions to be pre-executed based on its position. If it ranks before some pre-executed transactions, it requires immediate pre-execution based on the state version at its position. Any new state version written by this transaction can render the state versions read by lower-ranked, previously pre-executed transactions stale, affecting prediction accuracy. Restarting pre-execution for these transactions from scratch can enhance prediction accuracy, yet it is costly.

We devise a lightweight pre-execution repair approach that repairs only inaccurately predicted *SV-branches*. The *branch predictor* will first perform a repair check that verifies whether the previous branch prediction results of these transactions remain accurate under the latest state versions. We continue with the previous example presented in Figure 5 to illustrate the details of repair. As depicted in Figure 6, the *branch predictor* receives a newly arrived transaction $TX3$ with a higher gas fee than the pre-executed transaction $TX4$ and then pre-executes $TX3$ immediately. Since $TX3$ writes a new version of a, the *Version 2* read by $TX4$ is stale, triggering a repair check of $TX4$. During the repair check, the *branch predictor* retrieves the branch prediction results of $TX4$ from the *pre-execution table* and fetches the latest version of a from the *multi-version cache*, i.e., the *Version 3* written by $TX3$. It then re-evaluates whether the *SV-condition* of $a < 25$ is met based on the latest version of a. If the evaluation result matches the prediction one, no repair is needed; otherwise, the pre-execution path needs to be repaired, and pre-execution should be restarted from the current *SV-condition*.

Perceptron-based prediction. As shown in Table 1, a proportion of *SV-branches* in Ethereum exhibits regular directional patterns. We can harness a learning algorithm to rapidly capture such regularities without retrieving variable values from the *multi-version cache*, mitigating the overhead of the multi-version state-based prediction. Given the limited resources of blockchain nodes, complex training on them is impractical. Inspired by CPU branch prediction, we employ the *perceptron* model [21, 22], a single-layer linear classifier that makes interpretable predictions based on a weighted sum of input features. The *perceptron* model does not require vast computational resources and can be trained on the fly, which is suitable for blockchain nodes. We thus introduce an optimized *perceptron+* model tailored for branch predictions of contract transactions.

In the *branch table*, we create a *perceptron+* instance for each *SV-condition* and store the actual branch direction (represented by the integer, i.e., '1' for taken, '−1' for not taken) after execution. We set a standard length n of branch direction histories that can be used by each *perceptron+* instance. Once the direction history length of a branch exceeds n, its oldest direction history will be deleted. The conventional *perceptron* takes a vector of direction histories $(h_1, ..., h_n)$ as input to compute a dot product with the corresponding weight vector $(w_0, ..., w_n)$, where w_0 is a bias weight. However, multiple transactions may evaluate the same *SV-condition* during pre-execution, resulting in identical predictions based on the same input vector, which affects prediction accuracy. To address this, our *perceptron+* model incorporates all predictions $(p_1, ..., p_m)$ in the current round of pre-execution as part of the direction histories while maintaining the input vector length at n. This allows the

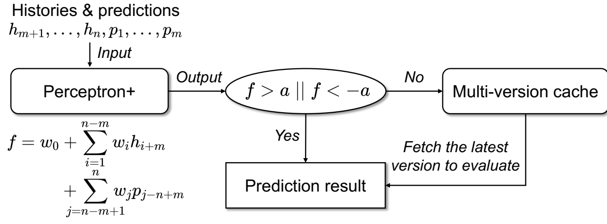


Figure 7: The overall process of two-level branch prediction

current prediction to fully utilize the previous prediction results from the same round. Hence, the prediction output f is computed as $f = w_0 + \sum_{i=1}^{n-m} w_i h_{i+m} + \sum_{j=n-m+1}^{n} w_j p_{j-n+m}$. Let α denote the threshold that can determine the confidence level of f. The predicted direction is considered taken if $f > \alpha$ and not taken if $f < -\alpha$. If f falls between $-\alpha$ and α, the prediction result is uncertain. After obtaining the actual branch directions, we conduct real-time weight updates for each *perceptron+* instance.

Two-level prediction process. We aim to coordinate the above two prediction approaches to ensure accurate and low-overhead branch prediction. The main design principle is to employ the *perceptron+* model for low-overhead prediction of *SV-branches* with regular historical directions, while *SV-branches* with irregular directional patterns are predicted using the multi-version state-based approach to ensure prediction accuracy.

When the direction history length is less than n, the *branch predictor* employs the multi-version state-based prediction. Once the length reaches n, the two-level branch prediction is activated, as depicted in Figure 7. The *branch predictor* first employs the *perceptron+* model to derive the prediction output f based on the direction histories and previous predictions if they exist. If f falls within the confidence interval ($f > \alpha$ or $f < -\alpha$), the prediction result can be directly used since the *SV-branch* has regular historical directions. If f does not fall within the confidence interval ($-\alpha \leq f \leq \alpha$), it is necessary to fetch the latest state version from the *multi-version cache* and evaluate the *SV-condition* to obtain the final prediction result. For some *SV-brances* whose direction histories consistently lack regularity, there is no need to continuously employ the *perceptron+* model for regularity checks. We set a check period and conduct only one regularity check using the *perceptron+* model in each period to further enhance prediction efficiency.

3.5 Checkpoint Snapshot

To maximize pre-execution effectiveness, we also need to cache pre-execution results at appropriate times so that the pre-execution results on accurately predicted paths can be reused during actual execution (i.e., fast-path execution). Even with inaccurate predictions, a proportion of cached results can still be reused, rather than executing transactions entirely from scratch.

During pre-execution, we generate a checkpoint snapshot at each *SV-condition* to cache execution information up to that point and store it in the *pre-execution table*. This enables the reuse of pre-execution results along the path leading up to the *SV-condition*, regardless of the branch direction. The structure of a checkpoint snapshot is shown in Figure 8, consisting of three parts: the read-write set, the branch information, and the execution state. The

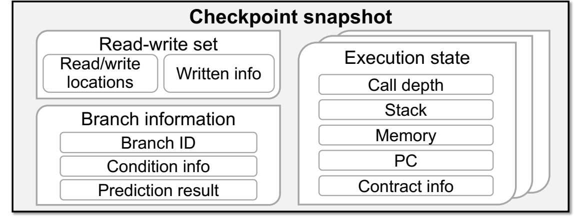


Figure 8: The structure of checkpoint snapshot

read-write set records all accessed storage slots and the written information prior to the *SV-condition*. The written information caches the written values, and if derived from a state variable, retains the relevant variable details along with the computation trace. The combination of read/write locations in each checkpoint snapshot forms the transaction's complete read-write set, used for subsequent concurrency control. The branch information includes the branch ID, the *SV-condition* details, and the prediction result, facilitating re-evaluation of *SV-conditions*. The execution state stores a snapshot of the stack and memory before the *SV-condition*, along with the PC value pointing to the next opcode, used to resume execution from the checkpoint when the predicted direction deviates from the actual one. For transactions with internal contract calls that trigger a multi-depth call stack, the execution state prior to the internal call at each depth should be stored, including the current call depth and the called contract information, e.g., contract address and consumed gas. When resuming the execution of transactions with internal contract calls, execution can start from the cached execution state at each call depth, further reducing execution overhead.

In addition to enabling fast-path execution, checkpoint snapshots can also be utilized in any case that requires efficient re-execution by reusing cached execution results, e.g., pre-execution repair and re-execution of aborted transactions. In the following subsection, we take fast-path execution as an example to elaborate on how to apply checkpoint snapshots for efficient execution.

3.6 Checkpoint-Based Fast-Path Execution

Upon receiving a newly generated block, each node employs an *execution scheduler* to concurrently execute transactions within the block for state transitions. For each transaction, a thread assigned by the *execution scheduler* retrieves their cached checkpoint snapshots from the *pre-execution table* for fast-path execution.

Fast-path execution. The core of fast-path execution lies in the thread re-evaluating the *SV-condition* in each checkpoint snapshot to verify whether the pre-execution results of a transaction can be reused, thus avoiding re-execution from scratch. During each round of re-evaluation, the thread checks if the actual branch direction matches the predicted one. If it matches, the state transitions generated on the path from the current *SV-branch* to the next checkpoint can be reused to bypass the normal EVM execution and proceed to re-evaluate the next *SV-condition*. If it does not match, the thread uses the cached execution state to resume the normal EVM execution. Such fast-path execution allows a full bypass of opcode-based execution in the EVM if all *SV-branches* are accurately predicted, while still enabling partial reuse of cached results when predictions are inaccurate, significantly reducing execution overhead.

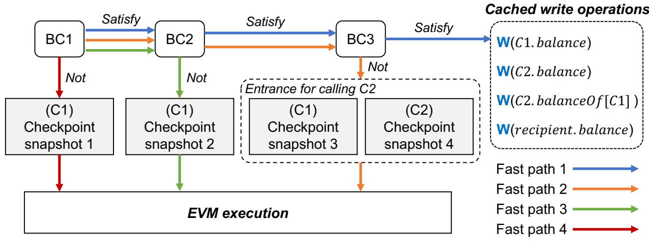


Figure 9: An example of checkpoint-based fast-path execution

Correct state transitions. Before re-evaluating each *SV-condition*, the thread needs to perform all state transitions generated along the path leading up to the current checkpoint. Specifically, it executes the `SSTORE` opcode to store the cached written values from the written information into the corresponding slots. However, if the cached written value is derived from computations involving state variables rather than constants, it needs to be recomputed for correct state transitions, as those variables' values may change during execution. As mentioned in Section 3.3, we have tracked all the computations operated on each state variable on the stack. Rather than performing recomputations on the stack, the thread first executes the `SLOAD` opcode to retrieve the current values of involved state variables, performs a quick recomputation using the cached computation trace, and stores the updated written value.

Example. We continue with the example of the *Payment* contract (C1) presented in Section 2.2 to illustrate the fast-path execution process. We cache four checkpoint snapshots for three BCs in the function of *unwrapWETH9*, where the checkpoint snapshot 3 provides an entrance for internally calling C2. Assuming the predicted branch directions for the three BCs are all taken during pre-execution, the following four fast paths would occur during actual execution. ❶ **Fast path 1:** The actual branch directions of the three BCs satisfy their predictions. The opcode-based execution in the EVM is bypassed, and the correct state transitions are performed using the cached four write operations. ❷ **Fast path 2:** The actual branch directions of BC1 and BC2 match their predictions, yet BC3's actual direction does not. The execution path before BC3 is skipped, and execution can be resumed using the checkpoint snapshots 3 and 4. ❸ **Fast path 3:** Only BC1's actual direction matches the prediction. Hence, the execution path before BC2 is skipped, and execution can be resumed by using the checkpoint snapshot 2. ❹ **Fast path 4:** Although none of the actual directions match predictions, some constant operations before BC1 are skipped, and execution can be resumed by using the checkpoint snapshot 1.

3.7 Adaption to MVCC

Concurrency control is essential for effective conflict resolution during concurrent fast-path execution. Current approaches to accelerate blockchain transaction execution typically adopt MVCC schemes, which can reduce transaction aborts and ensure serializability by allowing concurrent transactions to read their dependent state versions instead of a unified version. To adapt to current blockchain MVCC schemes, Seer abstracts key functions of MVCC and provides interfaces for pluggable use.

Algorithm 1: Concurrent execution using MVCC interfaces

Input: Smart contract transactions $CTXs$ to be executed, checkpoint snapshots CSs, transaction read-write sets $RWSets$

1 $Deps \leftarrow$ a mapping structure to store transaction dependencies;
2 $ValTasks \leftarrow$ an empty queue of transactions to be validated;
3 $CommitID \leftarrow 0$;
4 $G \leftarrow MVCC.Construct(RWSets)$;
5 **for** $ctx \in CTXs$ **do**
6 $\quad dep_{max} \leftarrow -1$;
7 $\quad$ **for** $dep \in G.edges[ctx.id]$ **do**
8 $\quad\quad dep_{max} \leftarrow max(dep_{max}, dep)$;
9 $\quad Deps[ctx.id] \leftarrow dep_{max}$;
10 **while** $CommitID < CTXs.length$ **do**
11 $\quad ctx_{next}, idleT_i \leftarrow MVCC.Schedule(CTXs, Deps, CommitID)$;
12 $\quad idleT_i.FastPath(CSs[ctx_{next}.id])$; // run in $idleT_i$
13 $\quad ValTasks.add(ctx_{next})$; // run in $idleT_i$
14 $\quad$ **while** $ValTasks.length > 0$ **do**
15 $\quad\quad ctx_{val} \leftarrow ValTasks.poll()$;
16 $\quad\quad$ **if** $ctx_{val}.id \neq CommitID$ **then**
17 $\quad\quad\quad ValTasks.add(ctx_{val})$;
18 $\quad\quad\quad$ **break**
19 $\quad\quad abort \leftarrow MVCC.Validate(ctx_{val}, Deps[ctx_{val}.id])$;
20 $\quad\quad$ **if** $abort$ **then**
21 $\quad\quad\quad MVCC.Abort(ctx_{val})$;
22 $\quad\quad$ **else**
23 $\quad\quad\quad MVCC.Commit(ctx_{val})$;
24 $\quad\quad\quad CommitID \leftarrow CommitID + 1$;
25 **return**

We select OCC-DA [15] as an example to illustrate the process of concurrent execution using MVCC interfaces, as presented in Algorithm 1. The *execution scheduler* first calls the *MVCC.Construct* interface to build a dependency graph based on the read-write sets $RWSets$ generated by the combined read/write locations from checkpoint snapshots, and identifies the dependent transaction with the highest ID of each transaction within the graph (lines 4-9). Then, the *execution scheduler* calls the *MVCC.Schedule* interface to fetch the idle thread and the next executable transaction ctx_{next} whose dependent transactions have been committed to ensure serializability. The idle thread performs fast-path execution using the checkpoint snapshots of ctx_{next} and adds ctx_{next} to the validation task queue $ValTasks$ after fast-path execution (lines 10-13).

During transaction validation, the *execution scheduler* retrieves the transaction ctx_{val} from $ValTasks$ and calls the $MVCC.Validate$ interface to verify if its read state versions have been modified by other concurrent transactions (lines 14-19). If the read versions of ctx_{val} are stale, the *MVCC.Abort* interface is called to abort ctx_{val}, i.e., revert its write versions and push it back into the execution queue to await fast-path re-execution (line 21). If its read versions remain unmodified, the *MVCC.Commit* interface is called to commit ctx_{val} and persist its state updates (lines 23-24). Benefiting from our branch prediction, transaction dependencies are relatively accurate, reducing the ratio of aborted transactions during validation.

3.8 Optimizations

Identify system state variables without storage. There is a category of variables related to the system state that are not stored in the *contract storage*, e.g., those indicating the current block height and

timestamp introduced by the opcodes of NUMBER and TIMESTAMP. Although these variables may only exist in a few contracts, not identifying them would result in their reuse as cached fixed results. In fast-path execution, if a transaction is located in a different block than during pre-execution, its cached fixed results may change, compromising execution correctness. Thus, branch conditions involving such variables should be treated as *SV-conditions*. As they do not involve storage compaction, we can identify their relevant *SV-conditions* by labeling them using specific opcodes, and track their related computations, similar to cases ❶ and ❸ in Section 3.3.

Memory management. Since we employ three new in-memory storage structures on top of the native EVM for pre-execution and fast-path execution, it is essential to manage them to avoid imposing a burden on memory costs. In the *branch table*, we only retain frequently used *SV-conditions* and delete entries of infrequently used ones. Specifically, we establish a lifecycle for each *SV-condition*. After a fixed time interval, we calculate its activity level, i.e., the access frequency divided by its lifecycle. Based on the activity levels, we perform a table cleanup. For the *multi-version cache*, all inserted write versions in the table will be deleted after each round of pre-execution. For the *pre-execution table*, we delete the pre-execution results of transactions that have been included in a block and executed afterward. The results of remaining transactions not yet included in a block will be retained for reuse in subsequent pre-execution rounds, avoiding the need to restart from scratch.

3.9 Correctness

We next discuss the correctness of Seer mainly in terms of how it ensures the serializability of execution and the consistency of state transitions between honest nodes.

Theorem 1. *Given a block $B = \{tx_1, \ldots, tx_n\}$, the execution of transactions in B using Seer can yield the same state transition ζ as serial execution, and ζ can be produced on each honest node side.*

Proof. We first prove that the fast-path execution result of a transaction tx_i using Seer is identical to its normal execution in the EVM. For contract transactions, branch prediction accuracy only impacts their actual execution efficiency without compromising execution correctness. For accurate predictions, the actual execution path matches the predicted one. As discussed in Section 3.6, any discrepancies in cached state transitions on the predicted path can be corrected by recomputing written values using the latest variable states. For inaccurate predictions, checkpoint snapshots allow execution to resume in the EVM. This ensures that any divergent execution results of tx_i can be corrected by the normal EVM execution, yielding correct state transitions. For common transfer transactions, Seer defaults to the native EVM execution.

Due to network latencies, nodes may pre-execute different transactions, resulting in divergent pre-execution results. However, since these pre-execution results impact only execution efficiency, they do not compromise consistency among nodes. Moreover, MVCC guarantees the serializability and consistency of concurrent executions by deterministically producing identical state transitions for the same transaction inputs, adhering to a serial execution schedule. Even with Byzantine adversaries, the native consensus protocol maintains a consistent blockchain view among honest nodes, and their independent transaction executions remain unaffected. To sum up, once an agreed-upon block B is proposed, each honest node can obtain the same transaction input and produce a unified state transition ζ, equivalent to that of serial execution, through Seer's concurrent fast-path execution. □

4 IMPLEMENTATION

We have implemented Seer on top of Geth v1.11.5, primarily introducing a new *virtual machine* (VM) module that can replace the native one. To support stack tracing during pre-execution, we establish a new set of on-stack instructions in the VM module, named *pre-instructions*. The label structure captures information about labeled state variables and maintains a trace of their relevant computations on the stack. The *multi-version cache* is implemented using a double-linked list to efficiently manage write versions. Besides, we maintain a list of read versions to track transactions accessing the *multi-version cache*. When a transaction writes a new version, the cached read versions can help identify lower-ranked transactions that have missed this version, triggering repair checks if needed. We implement a *perceptron+* model and integrate it into the *pre-instructions* for on-stack invocation. Besides, we implement the *branch table* and the *pre-execution table* using mapping storage structures for efficient retrieval. To support concurrent transaction execution, we implement a series of MVCC interfaces and optimize the native in-memory state database (i.e., *StateDB*) to maintain multiple write versions of each state item.

5 EVALUATION

We evaluate Seer under realistic blockchain workloads, aiming to answer the following questions: (i) What are the effectiveness and overhead of Seer's prediction-based pre-execution? (ii) What is the overall performance improvement brought by Seer in the execution phase? (iii) What are the performance gains and resource utilization resulting from Seer's core design components?

5.1 Experimental Setup

Testbeds. Our testbed consists of Amazon EC2 c7i.8xlarge instances, each with a 3.2-GHz Intel® Xeon® Platinum 8488C processor (32 vCPUs, 64 GB memory), running Ubuntu 22.04 LTS. We run all experiments on two identical VMs to observe if they output consistent state updates. Each reported evaluation result averages five independent runs on the two VMs.

Workloads. We select real transaction data from the Ethereum network as the workloads to evaluate the performance of Seer in an actual blockchain environment. The evaluation workloads range from a block height of 14,000,000 to 14,750,000 (January 2022 to May 2022), during which the daily transaction activity on Ethereum is relatively high. We employ two modes to replay Ethereum transactions. The first mode sequentially replays Ethereum blocks with realistic sizes (*realistic-replay*), primarily to evaluate branch prediction accuracy and the distribution of speedups across transactions. The second mode replays large Ethereum blocks of varying sizes (*synthetic-replay*), primarily simulating large-scale workloads within each block generation interval to evaluate the performance of pre-execution and concurrent execution.

To evaluate contract transactions with varying complexity, we classify them in the evaluation workloads into complex (C-TXs) and

Table 2: Average *SV-condition* count across contract transactions with varying gas usage in the workloads

Gas usage	[21K, 60K)	[60K, 100K)	[100K, ∞)
Percentage (%)	46%	17%	37%
# of *SV-conditions*	1.9	3.5	7.1

* In Ethereum, the gas consumption for common transfer transactions is fixed at 21,000, while contract transactions typically consume 21,000 gas or more.

normal (N-TXs) categories based on gas usage, as adopted in prior works [8, 12] to measure the computational cost during contract execution. Furthermore, we analyze the relationship between gas usage and the number of *SV-conditions*, as shown in Table 2. As expected, higher gas usage corresponds to a higher average number of *SV-conditions.* Transactions consuming less than 100,000 gas have an average number of *SV-conditions* below the overall average of 4.1 (Table 1), while those consuming 100,000 gas or more far exceed this average. Thus, we use 100,000 gas as the threshold to distinguish C-TXs (37% share) from N-TXs (63% share) for later evaluation.

Baselines. We compare Seer against four baseline approaches: vanilla Ethereum [44], Forerunner [8], MTPU [32], and OCC-DA [15]. Vanilla Ethereum uses the native execution engine to execute each transaction in a block sequentially. Forerunner and MTPU employ different pre-execution solutions to accelerate actual execution. OCC-DA is an enhanced protocol of MVCC using pre-obtained read-write sets. Regarding baseline implementations, we directly utilize the official open-source implementations [14, 30] of vanilla Ethereum and Forerunner. In contrast, MTPU and OCC-DA have not yet released their implementations as open source. We thus implement the pre-execution scheme of MTPU and the MVCC protocol of OCC-DA based on the descriptions in their respective papers. Moreover, we implement OCC-DA's native method of obtaining read-write sets (a.k.a. R/W), which only fetches read-write sets during pre-execution without caching any pre-execution results.

5.2 Pre-Execution Effectiveness

In this series of experiments, we investigate the effectiveness and overhead of Seer's pre-execution. Each experiment is conducted under both C-TXs and N-TXs.

Prediction accuracy. We employ *realistic-replay* to execute transactions in 1,000 consecutive blocks and observe the average branch prediction accuracy within each 200-block height interval. We use two metrics to evaluate branch prediction accuracy: the ratio of transactions with all *SV-branches* accurately predicted (i.e., perfect predictions) and the ratio of accurately predicted branches (i.e., hit ratio). Figure 10 shows the results, where the error bars indicate the range between maximum and minimum values. The branch prediction accuracy remains stable across different block heights, with N-TXs performing better than C-TXs. However, even for C-TXs, approximately 80% achieve perfect predictions, and the hit ratio is only slightly lower than that of N-TXs. This indicates that a large proportion of the pre-execution results can be reused.

Transaction-level speedup. Next, we quantify how much Seer can accelerate the execution of each contract transaction over vanilla Ethereum and compare its performance with MTPU and Forerunner. We use the *realistic-replay* mode to execute transactions from

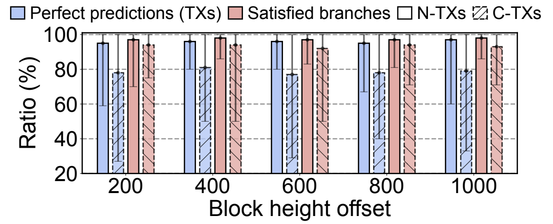


Figure 10: Average branch prediction accuracy

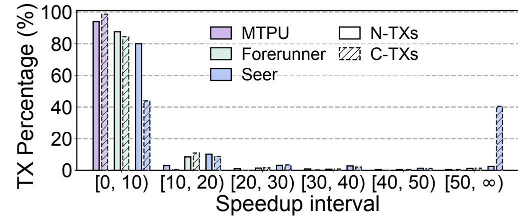


Figure 11: Comparison of transaction-level speedup

Table 3: Comparison of the ratio of aborted transactions

# of TXs	2K	4K	6K	8K	10K
R/W +	3.1% (c)	5.6% (c)	5.9% (c)	6.1% (c)	6.4% (c)
OCC-DA	1.5% (n)	4.6% (n)	5.2% (n)	5.5% (n)	5.9% (n)
Seer +	0% (c)	0.05% (c)	0.17% (c)	0.23% (c)	0.22% (c)
OCC-DA	0.2% (n)	0.15% (n)	0.33% (n)	0.47% (n)	0.48% (n)

* 'c' denotes the ratio of aborted C-TXs, and 'n' denotes the ratio of aborted N-TXs.

10,000 blocks for stable results. Figure 11 presents that, for both MTPU and Forerunner, most transactions achieve a speedup within 10×. MTPU performs slightly better on N-TXs than C-TXs. Forerunner shows the opposite trend but remains limited, with only 1.4% of C-TXs reaching a speedup of 50× or more. In contrast, Seer significantly outperforms on C-TXs, with 40.6% achieving a speedup of 50× or higher. The average speedup of all C-TXs reaches 50.2×. Due to the smaller number of *SV-conditions* in N-TXs, the speedup effect is somewhat limited, yet the average speedup still reaches 9×, outperforming the two baselines. This speedup improvement is driven by Seer's accurate branch prediction and efficient fast-path execution, demonstrating Seer's capability in handling increasingly complex transactions in the future.

Transaction abort rate. In this experiment, we compare the transaction abort rates of OCC-DA using Seer and the native R/W method to investigate conflict detection accuracy. We employ the *synthetic-replay* mode to simulate varying transaction scales. As reported in Table 3, the abort rate of OCC-DA using R/W is much higher than that of OCC-DA with Seer and grows as the number of concurrent transactions increases, with C-TXs being especially prone to aborts. This is because R/W statically derives read-write sets based on a single state snapshot, which may result in changes to these sets during actual execution (Section 2.3). In contrast, Seer, with its branch prediction functionality, can capture accurate read-write sets for conflict detection. This greatly reduces abort rates for both C-TXs and N-TXs, maintaining the overall abort rate below 1%.

Pre-execution latency. Apart from pre-execution benefits, we compare the pre-execution latency of Seer, MTPU, and Forerunner

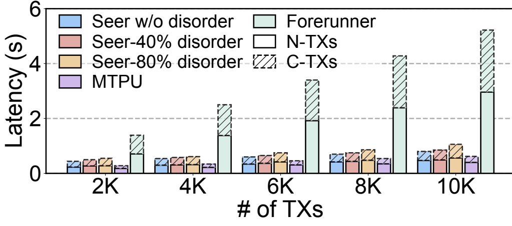


Figure 12: Comparison of pre-execution latency

using *synthetic-replay* under varying transaction scales. To investigate Seer's pre-execution repair overhead, we simulate realistic pre-execution disorder by inserting different proportions of out-of-order transactions (i.e., disorder ratio) into each node's transaction input queue. As shown in Figure 12, MTPU exhibits the lowest pre-execution latency across all scales. Forerunner, due to its brute-force approach, incurs significantly higher latency, with C-TXs (37% share) contributing to almost half of the pre-execution overhead. In contrast, Seer's prediction-based pre-execution strikes a balance, achieving a reduction of up to 5× compared to Forerunner, while maintaining effectiveness. Besides, as the disorder ratio increases, more transactions require pre-execution repair. Seer's lightweight repair mechanism ensures minimal impact on overall latency.

5.3 Overall Performance

This series of experiments evaluate the overall performance of the execution phase to explore how much Seer can enhance phase-level performance compared to the four baselines. We employ *synthetic-replay* to conduct experiments under varying transaction scales.

Phase-level speedup. First, we evaluate the phase-level speedup performance over vanilla Ethereum. We use OCC-DA to enable concurrent execution for Seer and MTPU, and compare them with R/W+OCC-DA. Forerunner, however, lacks concurrency control support and is presented with its native speedup. Figure 13 presents the comparison results, with the thread count indicated in the suffix of OCC-DA. The speedup for all approaches declines as transaction scale increases, due to Ethereum's lower memory cost, which marginally boosts its throughput with larger scales (Figure 14). Despite this, all approaches achieve over 9× speedup by offloading much of the computation and I/O costs off the execution phase (Figure 16(c)). Both R/W+OCC-DA and MTPU+OCC-DA achieve better performance with 8 threads compared to 4, due to greater execution concurrency. MTPU slightly enhances R/W+OCC-DA's performance, with its 8-thread speedup performance comparable to Forerunner using serial execution. In comparison, the 4-thread performance of Seer+OCC-DA exceeds its 8-thread counterpart, as precise conflict detection reduces false negatives, potentially limiting execution concurrency. Overall, Seer+OCC-DA achieves the best speedup performance, averaging 20.6× at large scales.

Overall throughput. Next, we evaluate the overall throughput performance during the execution phase. For Seer+OCC-DA, MTPU+OCC-DA, and R/W+OCC-DA, we showcase their optimal concurrent execution performances with varying threads. As shown in Figure 14, except for vanilla Ethereum, all comparison schemes exhibit throughput performance with little fluctuation at different transaction scales. Due to the close speedup performance, MTPU+OCC-DA

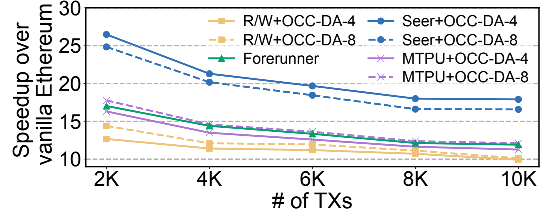


Figure 13: Comparison of speedup performance over vanilla Ethereum during the execution phase

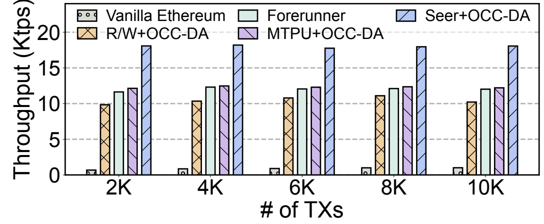


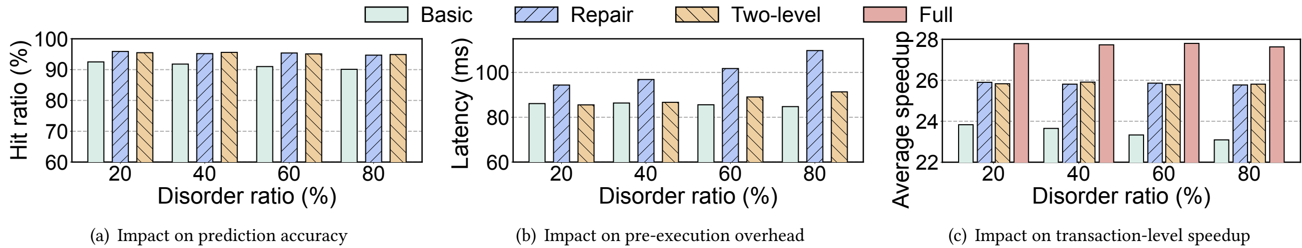
Figure 14: Comparison of throughput performance during the execution phase

and Forerunner deliver comparable throughput, both slightly outperforming the native OCC-DA. Similarly, Seer+OCC-DA achieves the highest throughput among all baselines, averaging 18 Ktps. The overall throughput improvement of Seer is expected to be even greater with increasingly complex transactions in the future.

5.4 Factor Analysis

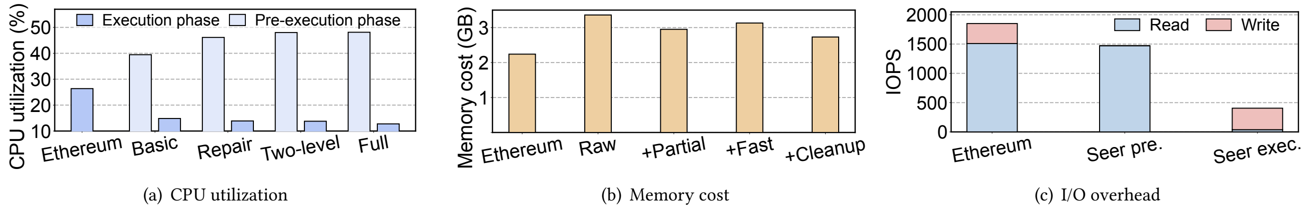
In this series of experiments, we study the impact of each core design component of Seer on pre-execution effectiveness and resource utilization. For pre-execution effectiveness and CPU utilization evaluations, we generate four Seer variants. *Basic* uses only multi-version state-based prediction. *Repair* adds the pre-execution repair feature. *Two-level* incorporates the *perceptron+* model for two-level branch prediction. The full version, *Full*, incorporates checkpoint-based fast-path execution. For memory tests, we compare various memory-saving optimizations: *Raw* stores all state variable versions in the *multi-version cache*, *+Partial* caches only those involved in *SV-conditions*, *+Fast* caches checkpoint snapshots, and *+Cleanup* conducts memory management as detailed in Section 3.8. For fairness, the four Seer variants employ *+Cleanup* for pre-execution effectiveness and CPU utilization evaluations.

Impact on pre-execution effectiveness. The following experiments investigate the impact of design breakdown with varying disorder ratios by employing *realistic-replay* over consecutive 1,000 blocks. First, we compare the branch prediction hit ratios of Seer variants. As depicted in Figure 15(a), the hit ratio of *Basic* drops with the disorder ratio increases. In comparison, *Repair* uses pre-execution repair to maintain a stable hit ratio. *Two-level* performs similarly to *Repair*, as both can accurately predict regular branch directions. Figure 15(b) presents the average pre-execution latency per block interval for the three Seer variants. Due to extra repair overhead, *Repair* exhibits the highest latency, which further increases with a higher disorder ratio. By leveraging the *perceptron+* model, *Two-level* reduces the pre-execution latency by up to 16.8% compared to *Repair*, while remaining only slightly higher than *Basic*.



(a) Impact on prediction accuracy (b) Impact on pre-execution overhead (c) Impact on transaction-level speedup

Figure 15: Pre-execution effectiveness under design breakdown



(a) CPU utilization (b) Memory cost (c) I/O overhead

Figure 16: Resource utilization under design breakdown

Note that we exclude the hit ratio and pre-execution latency for *Full* since it adopts the same pre-execution method as *Two-level*. Figure 15(c) illustrates the average transaction-level speedup for each Seer variant. *Basic* shows a decline in speedup performance as the disorder ratio rises due to its decreasing prediction accuracy. With similar prediction accuracy, *Repair* and *Two-level* show comparable speedup performance. In comparison, *Full* achieves a notable average speedup of 27.7×, surpassing all other variants.

Impact on resource utilization. The following experiments evaluate Seer's runtime resource utilization over 1,000 blocks using *realistic-replay*, averaging results per block generation interval. Figure 16(a) shows the average peak CPU utilization for Seer variants and Ethereum. By offloading much computational burden to pre-execution, Seer significantly reduces CPU usage during execution compared to Ethereum. Among variants, *Full* exhibits the highest pre-execution CPU utilization but the lowest during execution, aligning with its speedup results in Figure 15(c). Next, we evaluate the average memory cost for each memory-saving optimization, as depicted in Figure 16(b). Compared to *Raw*, *+Partial* optimizes memory usage by caching only the state variable versions relevant to *SV-conditions*. *+Fast* caches checkpoint snapshots, causing only a slight increase in memory usage. *+Cleanup* periodically purges all in-memory structures, achieving notable memory savings, with memory cost only 1.22× higher than Ethereum. Figure 16(c) presents the disk IOPS (*I/O operations per second*) of Seer and vanilla Ethereum. As Seer lacks specific disk access optimizations, only one result among variants is shown. Compared to Ethereum, Seer handles most I/O reads during pre-execution and incurs slightly lower I/O read costs due to some inaccurate prediction cases. During actual execution, Seer's write operations dominate the disk activity.

6 RELATED WORK

Blockchain pre-execution. The pre-execution technique is widely adopted by current blockchains. In permissioned blockchains, pre-execution is performed by a set of trusted nodes before the consensus phase [1, 2, 17, 18, 33]. In public blockchains, transactions can be speculatively pre-executed in parallel with transaction dissemination and consensus. Forerunner [8] and MTPU [32] are two state-of-the-art pre-execution schemes in public blockchains. Forerunner's brute-force pre-executing of all possible paths causes inefficiencies in handling complex contracts. MTPU restricts pre-execution to constant parts, leaving state-dependent operations on the critical path. Both approaches fail to generate precise read-write sets, hindering accurate conflict detection during concurrent execution.

Blockchain concurrency control. To enable concurrent transaction execution, many works leverage *optimistic concurrency control* (OCC) techniques from databases [6, 11, 26, 41]. Early blockchain OCC research [9, 23, 36, 37, 39, 45, 46] concurrently executes transactions based on the same state version, suffering from significant transaction aborts [28, 49]. To mitigate aborts, current research [15, 16, 34, 49] leverages MVCC to allow concurrent transactions to read their dependent state versions, which requires read-write sets for dependency (i.e., conflict) detection. However, their reliance on pre-computation or static analysis for obtaining read-write sets leads to inaccurate dependency detection in dynamic contract execution environments with numerous *SV-branches*.

7 CONCLUSIONS

This paper proposes Seer, an advanced transaction execution engine for public blockchains using best-effort pre-execution. Seer employs two-level branch prediction to improve the accuracy and reusability of pre-execution results and leverages checkpoint-based fast-path execution to enable effective reuse of cached results, enhancing execution efficiency. Intensive evaluations demonstrate that Seer outperforms state-of-the-art blockchain pre-execution and concurrency control schemes in speedup performance.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments. This work was supported by National Key Research and Development Program of China (Grant No. 2021YFB2700700), National Natural Science Foundation of China (Grant No. 62472185).

REFERENCES

- [1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *Proceedings of the IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.
- [2] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 2018 European Conference on Computer Systems (EuroSys)*. ACM, 1–15.
- [3] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. 2019. Prism: Deconstructing the Blockchain to Approach Physical Limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 585–602.
- [4] Binance. 2022. BNB Chain Document. https://docs.bnbchain.org/ [Last accessed on 2024-12-16].
- [5] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.
- [6] Matthew Burke, Florian Suri-Payer, Jeffrey Helt, Lorenzo Alvisi, and Natacha Crooks. 2023. Morty: Scaling Concurrency Control with Re-Execution. In *Proceedings of the 2023 European Conference on Computer Systems (EuroSys)*. ACM, 687–702.
- [7] Vitalik Buterin. 2014. A next-generation smart contract and decentralized application platform. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf [Last accessed on 2024-12-16].
- [8] 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 (SOSP)*. ACM, 570–587.
- [9] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding Concurrency to Smart Contracts. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, 303–312.
- [10] Solidity Documentation. 2023. Layout of State Variables in Storage. https://docs.soliditylang.org/en/latest/internals/layout_in_storage.html [Last accessed on 2024-12-16].
- [11] Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1930–1943.
- [12] Ethereum Foundation. 2024. Gas and Fees. https://ethereum.org/en/developers/docs/gas/ [Last accessed on 2024-12-16].
- [13] Ethereum Foundation. 2024. Merkle Patricia Trie. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/ [Last accessed on 2024-12-16].
- [14] Ethereum Foundation. 2024. Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum [Last accessed on 2024-12-16].
- [15] 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 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2315–2326.
- [16] 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 (PPoPP)*. ACM, 232–244.
- [17] Christian Gorenflo, Lukasz Golab, and Srinivasan Keshav. 2020. XOX Fabric: A hybrid approach to blockchain transaction execution. In *Proceedings of the 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.
- [18] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. 2019. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In *Proceedings of the 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 455–463.
- [19] Zicong Hong, Song Guo, Enyuan Zhou, Wuhui Chen, Huawei Huang, and Albert Zomaya. 2023. GriDB: Scaling Blockchain Database via Sharding and Off-Chain Cross-Shard Mechanism. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1685–1698.
- [20] Huawei Huang, Xiaowen Peng, Jianzhou Zhan, Shenyang Zhang, Yue Lin, Zibin Zheng, and Song Guo. 2022. BrokerChain: A Cross-Shard Blockchain Protocol for Account/Balance-based State Sharding. In *Proceedings of the 2022 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1968–1977.
- [21] Daniel A. Jiménez. 2003. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 243–252.
- [22] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 197–206.
- [23] 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.
- [24] Taeyoung Kim, Yunhee Jang, Chanjong Lee, Hyungjoon Koo, and Hyoungshick Kim. 2023. Smartmark: Software watermarking scheme for smart contracts. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 283–294.
- [25] Yeonsoo Kim, Seongho Jeong, Kamil Jezek, Bernd Burgstaller, and Bernhard Scholz. 2021. An off-the-chain execution environment for scalable testing and profiling of smart contracts. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. USENIX Association, 565–579.
- [26] Hsiang-Tsung Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226.
- [27] 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 *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. USENIX Association, 515–528.
- [28] Haoran Lin, Yajin Zhou, and Lei Wu. 2022. Operation-level concurrent transaction execution for blockchains. *arXiv preprint arXiv:2211.07911* (2022).
- [29] Deepak Maram, Harjasleen Malvai, Fan Zhang, Nerla Jean-Louis, Alexander Frolov, Tyler Kell, Tyrone Lobban, Christine Moy, Ari Juels, and Andrew Miller. 2021. Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability. In *Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1348–1366.
- [30] Microsoft. 2021. Official implementation of Forerunner. https://github.com/microsoft/Forerunner [Last accessed on 2024-12-16].
- [31] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf [Last accessed on 2024-12-16].
- [32] Rui Pan, Chubo Liu, Guoqing Xiao, Mingxing Duan, Keqin Li, and Kenli Li. 2023. An Algorithm and Architecture Co-design for Accelerating Smart Contracts in Blockchain. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 1–13.
- [33] 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. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2585–2598.
- [34] Xiaodong Qi, Jiao Jiao, and Yi Li. 2023. Smart contract parallel execution with fine-grained state accesses. In *Proceedings of the IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 841–852.
- [35] Xiaodong Qi and Yi Li. 2024. LightCross: Sharding with Lightweight Cross-Shard Execution for Smart Contracts. In *Proceedings of the 2024 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1681–1690.
- [36] 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 International Conference on Management of Data (SIGMOD)*. ACM, 543–557.
- [37] 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 (SIGMOD)*. ACM, 105–122.
- [38] Polygon Technology. 2024. POL: One token for all Polygon chains. https://polygon.technology/papers/pol-whitepaper [Last accessed on 2024-12-16].
- [39] Parth Thakkar and Senthilnathan Natarajan. 2021. Scaling blockchains using pipelined execution and sparse peers. In *Proceedings of the 2021 ACM Symposium on Cloud Computing (SoCC)*. ACM, 489–502.
- [40] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–38.
- [41] Jiachen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 198–216.
- [42] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447* (2021).
- [43] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. 2022. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies (AFT)*. 30–46.
- [44] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf [Last accessed on 2024-12-16].

[45] 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 *Proceedings of the IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 269–279.

[46] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling Blockchain Transactions through Off-Chain Storage and Parallel Processing. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2314–2326.

[47] Jie Xu, Qingyuan Xie, Sen Peng, Cong Wang, and Xiaohua Jia. 2023. AdaptChain: Adaptive Scaling Blockchain With Transaction Deduplication. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1909–1922.

[48] Haifeng Yu, Ivica Nikolić, Ruomu Hou, and Prateek Saxena. 2020. OHIE: Blockchain Scaling Made Simple. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 90–105.

[49] Shijie Zhang, Jiang Xiao, Enping Wu, Feng Cheng, Bo Li, Wei Wang, and Hai Jin. 2024. MorphDAG: A Workload-Aware Elastic DAG-Based Blockchain. *IEEE Transactions on Knowledge and Data Engineering* 36, 10 (2024), 5249–5264.

[50] Yuanzhe Zhang, Shirui Pan, and Jiangshan Yu. 2023. TxAllo: Dynamic Transaction Allocation in Sharded Blockchain Systems. In *Proceedings of the 39th IEEE International Conference on Data Engineering (ICDE)*. 721–733.

[51] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.

[52] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.