Journal of Systems Architecture 177 (2026) 103848

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Efficient blockchain transaction execution by mitigating MPT I/O overhead on the critical path

Ze Yin [a,b], Chubo Liu [a,b,c,*], Kai Zhong [a,b], Leilei Du [a,b], Rui Pan [a,b], Keqin Li [a,b,d], Kenli Li [a,b,c]

[a] The College of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, Hunan, China
[b] The National Supercomputing Center in Changsha, Changsha, 410082, Hunan, China
[c] The Ministry of Education Key Laboratory of "Fusion Computing of Supercomputing and Artificial Intelligence", Changsha, 410082, Hunan, China
[d] The Department of Computer Science, State University of New York, New Paltz, 12561, NY, USA

ARTICLE INFO

Keywords:
Blockchain
I/O overhead
Transaction execution
Merkle patricia trie

ABSTRACT

Blockchain has become a core infrastructure for the digital economy and is increasingly deployed in weak-trust multi-party settings. In each round of blockchain processing, the critical path spans consensus and execution. As consensus protocols improve, transaction execution has become the primary bottleneck limiting throughput. In particular, each read or write on state in the Merkle Patricia Trie (MPT) is amplified to $O(\log n)$ I/O operations, and this access overhead dominates execution efficiency. In view of this, we propose MOP, an efficient transaction execution mechanism by **M**itigating MPT I/O **O**verhead on the critical execution **P**ath. Unlike prior approaches that mainly redesign the authenticated storage structure itself, MOP preserves the existing storage model and improves throughput by moving most of this overhead off the critical path. First, MOP exploits the predictability of most reads and uses instruction traceback and access-table construction to prefetch state before execution by overlapping prefetching with consensus. Second, MOP defers state write-back to the next consensus phase and verifies consistency with a state-forest structure. This removes most read I/O from the execution phase and masks most write overhead. Experiments show that MOP improves throughput by 1.57×–3.13×.

1. Introduction

Blockchain technology has emerged as an indispensable foundation of the digital economy [1], offering decentralization, tamper resistance, and traceability across a wide range of applications, including applications in data sharing and storage [2–5], Web 3.0 [6], and federated learning [7–9]. One of the key challenges facing the rapidly evolving digital economy is to increase the throughput of underlying blockchain systems [10,11]. Popular blockchain systems, such as Bitcoin [12] and Ethereum [13], are limited to processing approximately 20 transactions per second (TPS).

The critical path of blockchain throughput consists of a recurring cycle between the consensus and execution phases [14]. Distributed nodes first propose a block containing new transactions through consensus. They then all execute the transactions within that block to produce a consistent update to the system state. Recent optimizations to consensus algorithms and network protocols have pushed achievable blockchain throughput to over several thousand TPS [15–17]. As a result, inefficient transaction execution has become the primary bottleneck.

Transaction execution time is dominated by reads and writes to the blockchain state (Fig. 1), which account for 81% of the total execution cost [18,19]. This considerable inefficiency stems from the authentication requirements, because nodes do not trust any information from other nodes in a decentralized environment. Generally, there are two types of nodes in blockchain: full nodes and light nodes. Only the former store the complete state data, while the latter merely retain block headers. The state data, represented as key–value pairs, is stored in the MPT [13]. Each leaf in the MPT stores a value, and the path from root to this leaf corresponds to the associated key. Each inner node in the MPT stores a hash of the concatenated content of all its children. The proof of a specific piece of state data consists of the sequence of hashes along the root-to-leaf path. This proof allows the light nodes to independently verify the authenticity of the state data received from the full nodes, relying solely on the root hash stored in the block header [20]. This mechanism ensures that other nodes can participate in the consensus without having to maintain the entire state data, thus saving resources while maintaining a secure trust level in the distributed system.

* Corresponding author at: The College of Computer Science and Electronic Engineering, Hunan University, Changsha, 410082, Hunan, China.
E-mail addresses: zyin@hnu.edu.cn (Z. Yin), liuchubo@hnu.edu.cn (C. Liu), startkz@hnu.edu.cn (K. Zhong), leileidu@hnu.edu.cn (L. Du), prui@hnu.edu.cn (R. Pan), lik@newpaltz.edu (K. Li), lkl@hnu.edu.cn (K. Li).

https://doi.org/10.1016/j.sysarc.2026.103848
Received 7 March 2026; Received in revised form 23 April 2026; Accepted 15 May 2026
Available online 21 May 2026
1383-7621/

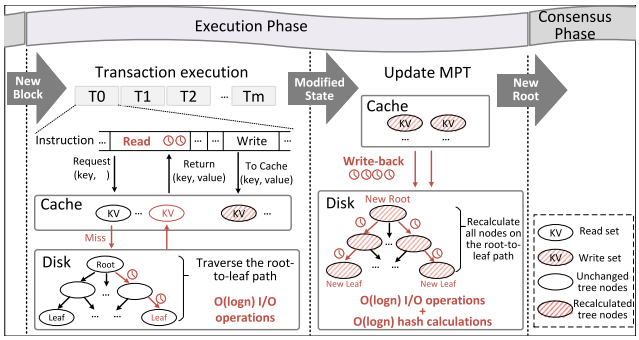


Fig. 1. Time-consuming but crucial read and write operations.

Unfortunately, the structure of MPT degrades read/write performance, as shown in Fig. 1:

(1) First, a read or write operation to state data is amplified into multiple disk I/O operations for all MPT nodes along the corresponding root-to-leaf path. In the worst case, the amplification is $O(\log n)$, where n is the size of the state.

(2) Second, the execution of each transaction involves both reading from and writing to the state. Transactions perform batch updates to the state according to the logic defined in the smart contract, which functions as a customizable library of callable functions.

(3) Finally, read and write operations on the MPT typically lie on the critical execution path. The thread processing a transaction must wait for the completion of expensive read operations before executing the next instruction. The modified state is temporarily buffered in memory until all transactions in the block are processed, after which it is written back to the on-disk MPT. To maintain state consistency across distributed nodes, each node must compute the updated MPT root to verify consistency before entering the next phase of consensus.

Consequently, time-consuming but crucial read and write operations limit the efficiency of transaction execution. Current works modify the MPT structure to reduce amplification, but this often necessitates complex cryptographic design or protocol construction [18,21–23].

To overcome the above problems, we propose MOP, an efficient blockchain transaction execution mechanism by **M**itigating MPT I/O **O**verhead on the critical **P**ath. In particular, we propose a data prefetching mechanism to load relevant state into Cache[1] before execution, substantially reducing read latency on the critical path. Additionally, we propose a lazy update mechanism that decouples consistency verification from write-back and defers write-back to the subsequent consensus phase, thereby hiding write overhead within consensus time. By parallelizing read and write operations with the consensus phase, MOP effectively hides the overhead incurred by frequent accesses to the on-disk MPT during execution, thereby improving overall throughput. The main contributions can be summarized as follows:

- We propose MOP, a transaction execution mechanism that mitigates significant MPT-related I/O overhead on the critical path by leveraging data prefetching and lazy update strategies.
- We employ a novel instruction traceback approach combined with access history analysis to achieve accurate and efficient data prefetching.
- We design a MPT construction approach that quickly verifies state consistency without incurring disk I/O, by constructing an additional modified state tree (MST) over modified states.
- Experiments show that shortening the critical path by optimizing the I/O overhead can lead to a speedup of 1.57×-3.13×.

[1] A software-managed in-memory region used for the proposed data prefetching and lazy update is referred to as the "Cache".

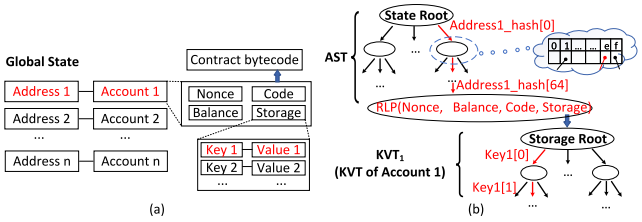


Fig. 2. (a) Global state; (b) Two-level MPT: AST and KVT.

2. Background and related work

2.1. Background

2.1.1. State transitions

Ethereum is the largest permissionless blockchain supporting smart contracts [13]. Smart contracts are developed in high-level languages, compiled into bytecode, and executed by the ethereum virtual machine (EVM) on each distributed node [24]. Ethereum is essentially a transaction-based state machine, where the state is a mapping from account addresses to their associated data (*Nonce*, *Balance*, *Code*, and *Storage*), as shown in Fig. 2(a). When a smart contract is deployed, it is assigned an address, with its *Code* storing bytecode and *Storage* maintaining an internal key–value store. Transactions drive state transitions by updating account data.

Ethereum state transitions follow a three-phase model [14], i.e., dissemination, consensus, and execution. Dissemination proceeds concurrently, while consensus must be reached before execution to ensure consistency. Constructing a new block also requires verifying the latest state. Thus, the critical path for throughput consists of repeated cycles of consensus and execution. Nowadays, efficient consensus mechanisms have boosted blockchain throughput to thousands of TPS, but large-scale real-time scenarios require higher throughput on the order of hundreds of thousands of TPS [25]. Under high transaction volume, execution may still cause congestion. Consequently, inefficient transaction execution becomes a new bottleneck limiting further throughput improvements.

2.1.2. MPT

The global state of Ethereum is stored in a MPT. Fig. 2(b) depicts its two-level structure: the account state tree (AST) and the key–value store tree (KVT). In the AST, the root-to-leaf path is determined by the 256-bit hash of the account address, where each trie level corresponds to one hexadecimal digit. The leaf stores the account data encoded in Recursive Length Prefix (RLP). Each smart contract account possesses its own key–value store, organized as an MPT with the same structure. The root of the KVT is stored in the *Storage* field of the account. These key–value stores can only be accessed and modified through the execution of their corresponding smart contracts.

The nodes in the MPT are stored in Level DB. Reading the leaf value typically requires traversing up to 64 nodes, with each requiring a read operation on Level DB. Path compression can coalesce multiple nodes, but MPT reads remain amplified by tens of I/O accesses. Similarly, writes to the MPT are also amplified. Since each parent node contains the hash commitment over its children, modifying a leaf triggers updates to all nodes along the path to the root. Moreover, updates to a smart contract account undergo two successive rounds of write amplification: first, the KVT root is recalculated and written back to the account's *Storage*, which then triggers a second round of AST updates.

Table 1
Comparison of existing I/O bottleneck mitigation approaches and our MOP.

Method		mLSM [21]	LVMT [18]	RainBlock [23]	LMPT [22]	*MOP*
Design	Target	Read and write I/O in execution phase				
	Architecture	Software				
	Read path optimization	Bloom-filtered trie lookup	Direct KV reads ①	Distributed state prefetch	Layered lookup and read	Data prefetching
	Write path optimization	Batched writes with deterministic compaction	Batched commit writes	Asynchronous write-back	Delta-trie writes and periodic batched merge	Lazy update
	Design trade-off	I/O reduction, MPT replacement, deterministic compaction	I/O reduction, complex cryptography, proof sharding	Critical-path I/O decoupling, network overhead, role coordination	Layered MPT, high compatibility, merge overhead	Critical-path I/O decoupling, prefetch dependence, deferred write-back
Characteristic	MPT strategy	Replace		Optimize		
	Read amplification	Low		Low ②	Medium ④	Low
	Write amplification	High	Low		Almost none	
	Compatibility	Low		Medium ③	High	

Notes: ① Proof sharding; ② Extra network overhead; ③ Incrementally deployable but complex; ④ Low on hits, high on misses.

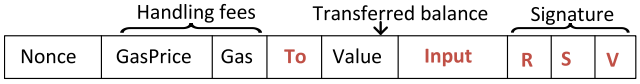


Fig. 3. Transaction data structure.

2.1.3. Transaction execution

As shown in Fig. 3, transactions are serialized using RLP for network dissemination and persistent storage. The transactions are classified into two categories: token transfer transactions and smart contract transactions. A token transfer transaction reads from the AST and updates the *Balance* of both accounts, with updates written back only after all transactions have been executed. For smart contract transactions, the EVM loads the smart contract bytecode based on the *To* field and executes it, with state primarily accessed via `SLOAD` and `SSTORE`.[2]

`SLOAD(key)→ value`: `SLOAD` reads the value from KVT based on the given key. The thread must wait for the completion of `SLOAD` before proceeding to the next instruction. During this period, other resources such as CPU and memory are not effectively utilized.

`SSTORE(key,value)`: `SSTORE` records key–value updates in memory temporarily. Subsequently, persisting the updates after executing all transactions in a block often incurs two rounds of write amplification and hash calculation.

Transactions in a block are executed serially by default to ensure state consistency. Read/write amplification in each transaction extends the critical path. Recently, transaction parallelism has emerged as a prevailing approach to improve throughput [26–28]. However, limited disk I/O resources still constrain concurrency: (1) numerous concurrent reads from the MPT are likely to be blocked, and (2) the time cost of write amplification remains constant. Therefore, the I/O overhead caused by read/write amplification limits the execution efficiency, regardless of the execution strategy employed.

2.2. Related work

Current research has proposed numerous approaches to improve blockchain throughput. We group these approaches into three categories: dedicated hardware accelerators, parallel transaction execution, and I/O bottleneck mitigation.

[2] The EVM also provides interfaces for accessing the AST. Similar to `SLOAD`, less frequent instructions such as `BALANCE` and `EXTCODECOPY` retrieve the *Balance* and *Code* of an account from the AST using the account address. However, write operations are only permitted in the KVT.

2.2.1. Dedicated hardware accelerators

Recent studies design dedicated accelerators for transaction execution. BPU [29] employs a modular, pipelined design to enhance the performance of general smart contract execution, and further establishes a dedicated dataflow for widely used ERC20 contracts. MTPU [30] adopts an algorithm and architecture co-design approach, leveraging parallelism by eliminating redundant execution and optimizing hot contracts.

Although dedicated hardware accelerators, especially when combined with software optimizations, can substantially increase blockchain throughput, blockchains are inherently multi-party distributed systems. Requiring all participants to deploy specialized accelerators is impractical, particularly for large-scale platforms such as Ethereum, which severely limits the applicability of this approach.

2.2.2. Parallel transaction execution

To ensure the blockchain state consistency, transaction parallelism must be serializable. Static analysis derives a safe parallel schedule from the observed read and write sets of transactions [31,32]. Dynamic conflict detection enables parallel execution of transactions in a speculative manner [26–28]. Software transaction memory techniques, such as pessimistic locking and optimistic concurrency control, are employed to capture transaction dependencies.

As noted earlier, transaction execution time is largely dominated by reads and writes to the blockchain ledger state, accounting for 81% [18,19]. Although parallel approaches improve processing efficiency by executing multiple transactions concurrently, the underlying I/O bottleneck remains, and overall throughput continues to be constrained by I/O.

2.2.3. I/O bottleneck mitigation

These works can be classified into two primary categories: reducing read and write amplification and mitigating the impact of disk I/O on the critical paths. These approaches are shown in Table 1.

mLSM [21] combines the MPT with log-structured merge trees. New updates are first buffered in memory and then written to Level 0; when a level reaches its threshold, it is compacted into the next level through deterministic compaction. Since most reads and writes are served in lower levels, mLSM reduces the I/O amplification of read and write operations, at the cost of replacing the original MPT and maintaining multi-level compaction logic.

LVMT [18] integrates a multi-level Authenticated Multipoint evaluation Tree (AMT) with a series of append-only merkle trees to achieve high efficiency. On one hand, instead of using value hashes, LVMT stores version numbers to avoid costly elliptic curve multiplication operations, enabling quick updates of the hash root. On the other

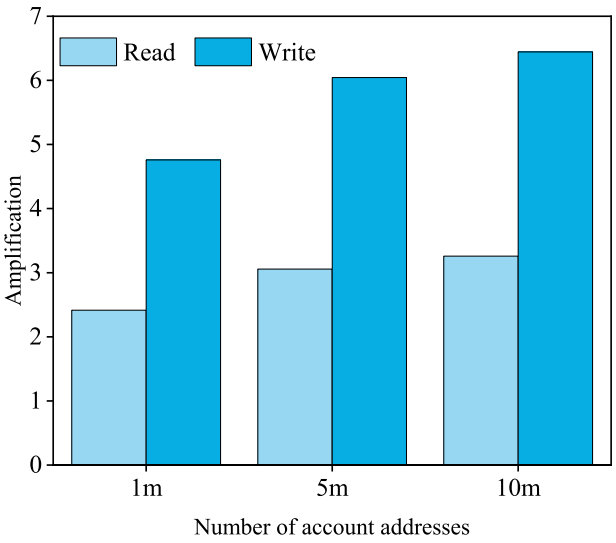


Fig. 4. The read and write amplification of MPT.

hand, LVMT employs a multilayer design where each level of AMT can accommodate 2^{16} children. This effectively reduces the depth of LVMT, resulting in less amplification for each read or write on the state. The trade-off, however, is that it requires a more complex cryptographic design and relies on additional mechanisms such as proof sharding.

RainBlock [23] introduces three entities for transaction execution, namely, storage nodes, miners, and state prefetchers. The storage nodes maintain a sharded, multi-version MPT in memory, enabling efficient read and write operations. Miners retrieve data through state prefetchers and transmit the execution results to storage nodes. RainBlock decouples state access from the miner execution process, thereby alleviating I/O overhead on the critical path. However, it relies on additional network communication and introduces extra system roles, leading to higher architectural and verification complexity.

LMPT [22] adopts a layered MPT structure, maintaining the snapshot MPT on disk while keeping the Intermediate MPT and Delta MPT, which record recent updates, in memory. New updates are first written to the in-memory layers and are periodically merged into the snapshot MPT. Since most reads and writes can be served in the in-memory layers, LMPT reduces the I/O overhead in the execution phase, at the cost of maintaining the layered structure and incurring periodic merge overhead.

In summary, mLSM, LMPT, and LVMT reduce the amplification of each read and write by modifying the MPT structure. In contrast, our design does not actually reduce disk I/O, but advances or delays these costs to avoid impacting the critical path. These two approaches are orthogonal and can be combined to further enhance performance. Although both our design and RainBlock eliminate disk I/O overhead during transaction execution, RainBlock remains limited by network inefficiency, which becomes the next bottleneck.

3. Insight

We collected statistics on the read and write amplification of MPT, and the results are shown in Fig. 4. Read and write amplification further increase the I/O overhead during the execution phase. Existing studies have shown that transaction execution is dominated by state read and write operations, which account for approximately 81% of the total execution time [18,19]. As shown in Fig. 5, we introduce the data prefetching and lazy update mechanisms to mitigate I/O overhead on the critical path of transaction execution. Our approach is informed by insights at the system, instruction, and data levels:

3.1. Insight 1: Predictable access behavior under intermittent transaction execution

In the three-phase model [14],[3] dissemination occurs concurrently, whereas the consensus and execution phases alternate. Transactions received by distributed nodes are not executed immediately, but are deferred to subsequent execution phases. This delay can even span multiple consensus-execute phases. The resources required by each phase remain largely underutilized outside their respective phases. Consensus algorithms primarily utilize computation resources (such as POW [12]) or network bandwidth (multi-round BFT negotiation [33]), while execution primarily consumes CPU and storage I/O resources. Therefore, there is sufficient time and idle resources to perform pre-optimizations before transactions are executed.

Meanwhile, transaction details such as *From*, *To*, and *Input* are known before execution. For token transfer transactions, execution must access the *From* and *To* accounts and update both *Balance* values. For smart contracts, we observe that operands of access instructions are typically the hash of a constant or the hash of the concatenation of a constant and an attribute value, such as *From*, *To*, and *Input*. Furthermore, since deployed smart contracts are immutable, these regularities of access behavior are always applicable once mined.

3.2. Insight 2: Write amplification triggered by small-scale state modifications during write-back

The prerequisite for consensus is a consistent initial state among distributed nodes, which is verified by the AST root. Therefore, it is necessary to write back the modified state to disk before entering the next consensus phase. One key insight is that if state consistency can be guaranteed without updating the AST root, then write overhead can be removed from the critical path.

Nowadays, Ethereum has hundreds of millions of accounts, but the average daily number of active accounts, namely those whose state changes, is approximately on the order of a few hundred thousand [34]. Thus, the state modified in each round is significantly smaller than the world state. However, each state modification is amplified equally, with the magnification proportional to the size of the world state. Moreover, this amplification process is deterministic. In other words, if two MPTs have the same root and perform the same modification on their leaves, the two new roots must also be the same.

3.3. Our solution

Inspired by Insight 1, we propose prefetching relevant state into the Cache before execution to mitigate most read overhead on the critical path. The intermittent nature of transaction execution provides sufficient time and I/O resources to complete prefetching. For token transfer transactions with fixed execution logic, accurate prefetching is achievable. In the following sections, we focus on prefetching for smart contract transactions. At the bytecode level, we employ instruction traceback and access tables to capture access rules, incurring this cost only once per smart contract since deployed contracts are immutable. Importantly, even if the prefetched state becomes dirty, a thread can still read the most recent data, as the modified state is temporarily stored.

Inspired by Insight 2, we design a lazy update mechanism that decouples write-back from consistency verification and defers write-back to the next consensus phase, thereby masking write overhead with consensus time. During execution, consistency is verified only for the modified state and not for the updated AST root. We construct an additional MPT, namely the MST, to represent the modified state. The MST can be efficiently built in Cache because the hash calculation scale

[3] This work is based on the DiCE three-phase model.

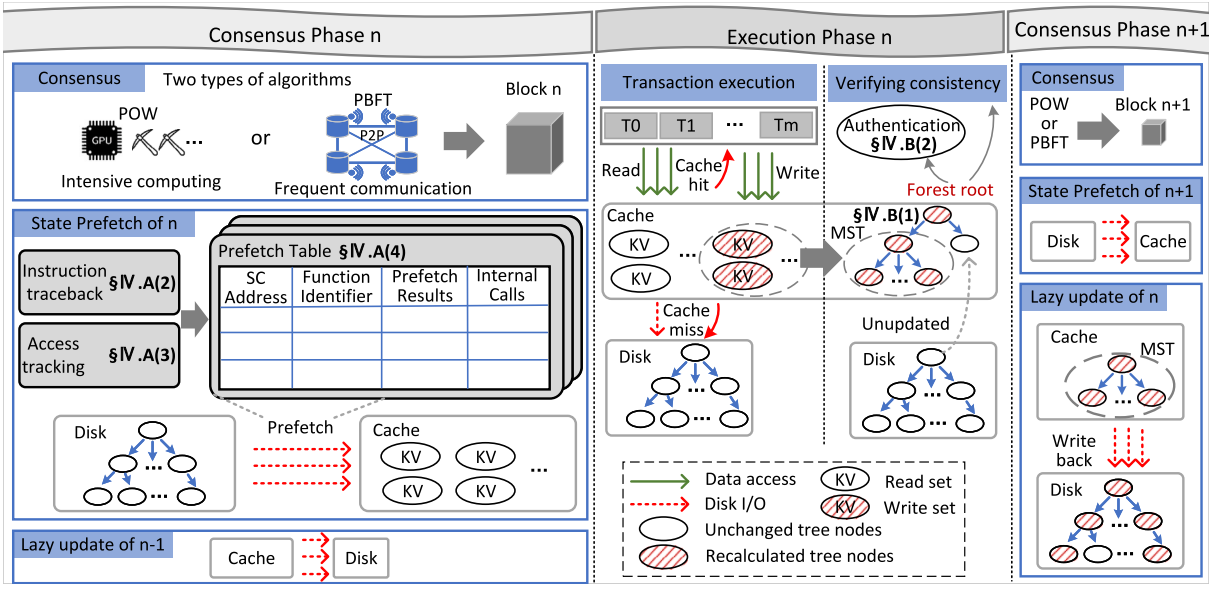


Fig. 5. State is prefetched in the consensus phase n and written back in the consensus phase n+1.

is small and there is no disk I/O overhead. The hash value of its root along with the root of unupdated AST are stored in the block header, which provides authentication and consistency checking.

Our design targets higher blockchain throughput. Although a blockchain is essentially a distributed database, unlike centralized databases, its throughput critical path includes an additional consensus phase. As a result, regardless of how efficient consensus becomes, blockchain throughput is inherently limited compared to that of a centralized database. To address this limitation, our design overlaps transaction reads and writes with the consensus phase, thereby reducing the critical path and improving throughput. In a theoretical scenario where consensus fully hides read and write costs, blockchain throughput is solely determined by transaction execution efficiency. Therefore, our MOP approach has the potential to raise the throughput ceiling. When combined with an efficient consensus algorithm, the system can achieve performance comparable to that of a centralized database.

4. Design of MOP

In this section, we present the design details of MOP. Specifically, Section 4.1 describes the data prefetching mechanism for smart contracts, while Section 4.2 presents the lazy state update strategy.

4.1. Data prefetching for smart contracts

Smart contracts are typically written in high-level languages such as Solidity [35]. Subsequently, they are compiled into bytecode, deployed on the blockchain, and executed by the EVM. For illustration, we use an ERC20 smart contract as a running example [36].

As shown in Fig. 6, the state variables declared in lines s3-s7 represent information such as token name, total amount, and the mapping between addresses and token balances. These variables are persisted as key–value pairs in the KVT. The EVM executes the SLOAD instruction to retrieve the value of a specific state variable based on its operand (i.e., the key of the key–value pair). Therefore, accurate inference of SLOAD operands before transaction execution is essential for prefetching. This is achieved through bytecode-level instruction traceback and access tables.

```
s1  contract TestToken {
s2      // persistent state variables of the contract
s3      string public name;
s4      uint8 public decimals;
s5      uint256 public _totalSupply;
s6      mapping (address => uint256) public balances;
s7      mapping (address => mapping (address => uint256)) public allowed;
s8
s9      // the methods are invoked to modify the state variable
s10     function transfer(address receiver, uint256 numTokens) public
                returns (bool success) {
s11         require(balances[msg.sender] >= numTokens );
s12         balances[msg.sender] = balances[msg.sender]-numTokens;
s13         balances[receiver] = balances[receiver]+numTokens;  }
s14     ...
```

Fig. 6. Source code of ERC20.

4.1.1. Key–value store

The key–value stores of smart contracts are all limited to 256 bits, with each key–value pair corresponding to a storage slot. To save space, multiple adjacent short variables are packed into a single slot as much as possible. For instance, `uint128` and `byte16` are placed in the same slot and accessed by the same key. In contrast, certain state variables may be stored across multiple slots such as dynamic arrays or mappings.

The specific access logic, i.e., the key for each slot, is compiled into the corresponding bytecode. Moreover, the source code of a smart contract is not necessarily published, while the bytecode is stored on the blockchain and directly accessible. Therefore, we perform analysis and prefetching at the bytecode level rather than at the high-level language level.

4.1.2. Instruction traceback

Ethereum adopts a stack-based instruction set, where the operands of all instructions are maintained on the stack. To adjust stack elements and provide appropriate operands for subsequent instructions, a large number of stack instructions (e.g., PUSH, POP, DUP, SWAP) are extensively used. Among them, the PUSH instruction, which accounts for approximately 26% of all instructions, pushes constant values onto the

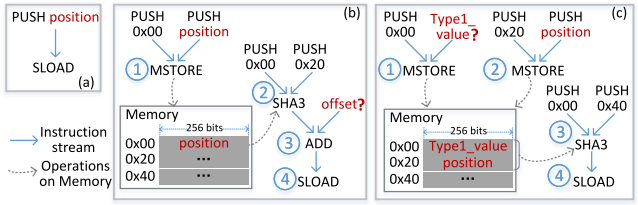


Fig. 7. Access logic of three typical types of SLOAD instructions.

stack.[4] In addition, DUP (20.32%) and SWAP (11.39%) are used solely to duplicate and reorder stack elements [30].

The extensive use of stack instructions implies that a large fraction of operands appearing on the stack are constants. Consequently, the operand of the SLOAD instruction can often be determined prior to execution.[5]

Access logic for state variables:

We further analyze the access logic of state variables to verify the above conjecture:

- Fixed-length variables: For example, `int`, `uint`, `bool`, `address`, and `byte1`-`byte32` fixed-length arrays. Each of these variables occupies at most one storage slot. Their storage keys are constants determined by the declaration order, and we refer to this constant as the variable position. In Fig. 6, `name` is declared first, and the key to access it is the constant 0.
- Dynamically sized arrays: Logically, the array's length is stored at the key position, which is used to verify that the access is reasonable. Subsequently, the array elements are stored contiguously starting at $Hash(position)$. Therefore, accessing an array element typically requires two SLOAD instructions: the first one retrieves the length to check for out-of-bounds conditions, and the second one accesses the actual element. In particular, if the array is small enough to pack both the length and all elements into a single slot, only one SLOAD is needed.
- Mapping: A `mapping(Type1→Type2)` represents a mapping relationship between two objects, where the former serves as an index and the latter represents a corresponding value. For instance, `mapping(address→uint)` in ERC contracts denotes the token balance at an address. Since mappings are non-contiguous, each entry is stored at key $Hash(position$ + `Type1_value`), which is solely determined by `Type1_ value`.

Access logic determined by constants:

The access logic for fixed-length variables, dynamically sized arrays, and a subset of mapping structures (where `Type1_ value` is a constant) is independent of execution context. Therefore, we are able to quickly determine the constant operands of such SLOAD instructions by static analysis.

Although the operand stack is complex, we only perform instruction traceback in a small range for the SLOAD instruction without full emulation. A single access to a state variable will be compiled into a contiguous piece of bytecode, including stack instructions, SHA3, SLOAD, and related operations.

The SLOAD instructions shown in Fig. 7(a) correspond to accesses to fixed-length variables, short arrays, and the length of long arrays. Although the bytecode fragment contains logical constructs such as checks and jumps, the operand of SLOAD is solely monitored to detect its appearance or modification on the stack. It can be confirmed that the operands are constants pushed by the PUSH instructions.

The logic for accessing elements of a long array is illustrated in Fig. 7(b). The operand of SLOAD consists of two components: $Hash(position)$ and offset. The SHA3 instruction performs a hash operation on a contiguous space of Memory,[6] and its two operands are the start address and the length of the space. The MSTORE instruction writes a 256-bit value to Memory, with operands being the value and the Memory address. The position is written into Memory by MSTORE, and then it is hashed by SHA3. Finally, ADD adds the offset to $Hash(position)$ as the operand of SLOAD. Unfortunately, in most cases such as common traversal operations, the offset is a variable.

The keys for the slots that hold the array elements are contiguous and differ only in the lower bytes. Thus, the root-to-leaf paths corresponding to these key–value pairs differ only in the last few levels. Based on this discovery, instead of prefetching the final leaves, we opt to prefetch the last node on the overlapping root-to-leaf paths. For instance, if all elements of an array occupy 16 slots, the root-to-leaf paths may only diverge at the terminal leaves. The common parent node of these leaves is prefetched. Based on this node, only one disk I/O is required to access any element of this array during actual execution. Naturally, all array elements are prefetched at once if they occupy only a small number of slots.

Access logic determined by transaction attributes:

In contrast to fixed-length variables and dynamically sized arrays, the keys of the slots that hold mapping variables are neither constant nor continuous. As shown in Fig. 7(c), MSTORE writes the position and `Type1_ value` into Memory, SHA3 hashes this Memory space, and the hash value serves as the operand of SLOAD. Except in rare cases, such as sending token to a fixed account or voting on a unique proposal, `Type1_ value` typically represents a variable value. In theory, it is impossible to prefetch accurately when the execution context is unknown.

By analyzing popular smart contracts, we observe that the `Type1_ value` is mostly derived from transaction attributes. All transaction attributes are known prior to execution, including *From*, *To*, and *Input*, among others.

One of the most important data structure in ERC contracts is `mapping (address→uint)`. The address type corresponds to an account address, which is derived from the account key using secp256k1 and Keccak-256 algorithms.[7] Consequently, variables of the address type are typically directly associated with an existing address constant. For example, the operand of SLOAD is associated with the caller address (`msg.sender`) or an address in `Input` (`receiver`), as shown in Fig. 6 (lines s11–s13).

The common indexes (`Type1`) of mapping variables, in addition to address type, include string and uint type. For instance, the `mapping(string→address)` signifies domain ownership, and the `mapping(uint→uint)` represents the number of votes received for a proposal. When executing the function of selling a domain or agreeing to a proposal, the index is naturally one of the parameters in *Input*. Similarly, the operand of SLOAD can be predetermined.

Therefore, for the mapping structure, we try to find out whether `Type1_ value` is a constant or an attribute value by instruction traceback, such as the constant pushed by the PUSH instruction, the caller address returned by the CALLER instruction, or the *Input* returned by the CALLDATA instruction. However, there are rare cases where `Type1_ value` is a variable, such as range traversals and random queries of mapping. In these situations, it is difficult to achieve good prefetching effect only by static analysis.

Example of traceback:

[4] The PUSHx instruction pushes the subsequent x bytes onto the stack. For instance, the bytecode '6080610647' represents PUSH1 0x80, PUSH2 0x0647.

[5] The operand of SLOAD is the key to access the key–value store.

[6] In order to differentiate, the term "Memory" refers exclusively to an infinite address space in the EVM that is used to hold temporary data.

[7] The process of creating a smart contract involves the generation of a unique address, which does not require reading the state and is beyond our scope of concern.

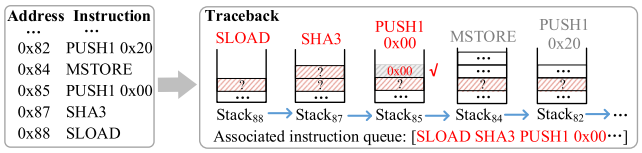


Fig. 8. An example of SLOAD instruction traceback.

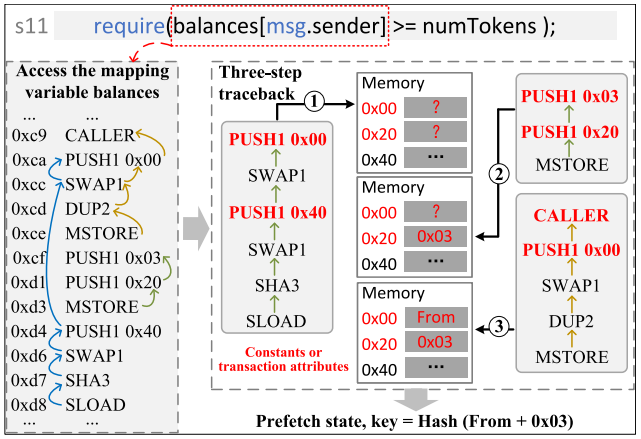


Fig. 9. Bytecode fragment for accessing the variable `balances` and the associated instruction queues with three backtraces.

The process of instruction traceback is illustrated in Fig. 8. In contrast to transaction execution, during each step of traceback, we restore the operand stack to its state before an instruction is executed. For instance, the SHA3 instruction pops the top two elements and pushes the hash result. This result at the top of $Stack_{88}$ is then fetched by the SLOAD as the access key. Hence, we track the two operands of the SHA3 instruction, i.e., the two top elements of $Stack_{87}$. The instructions that impact the tracked elements are recorded in a queue and subsequently executed in reverse order to locate the corresponding access object. Additionally, for SLOAD instructions belonging to the second and third class in Fig. 7, we also need to perform similar traceback for MSTORE.

In an ERC20 contract, it is necessary to check whether the sender has enough tokens before performing a transfer (s11 of Fig. 6). Fig. 9 illustrates the three instruction tracebacks for this access. In the first traceback, the operand of the SLOAD instruction is the result of the SHA3 instruction. The two operands of the SHA3 instruction are constants, and the 0x00-0x40 Memory space is hashed. The next step is to determine the values in this Memory space. In the second traceback, the two operands of the MSTORE instruction are two constants, and 0x03 is written into the Memory space. In the third traceback, the CALLER instruction pushes the sender's address *From* onto the stack, so the two operands of the MSTORE instruction are a transaction attribute and a constant. *From* is written into the Memory space. Finally, the state is prefetched and the key is $Hash(\texttt{From} + \texttt{0x03})$. If any of the traceback step fails, it is judged that this SLOAD instruction cannot complete the prefetching. It is worth noting that, to support prefetching, MOP constructs a jump map once per contract by resolving the targets of JUMP/JUMPI and caches the resulting control-flow links in the JUMP table; later tracebacks for SLOAD/MSTORE can then directly query this table instead of re-resolving jumps.

4.1.3. Access table

Static analysis enables the accurate prefetching of most types of state variables, except when the `Type1_ value` of a mapping is a variable. In this case, relying solely on static analysis makes it difficult to obtain good prefetching performance. Therefore, we construct an access table to record the operand of SLOAD for accomplishing operand inference with high probability.

The key discovery is that the invocation of smart contracts follows a long-tail distribution. A small number of hotspot contracts are executed with exceptionally high frequency. Over 65 million smart contracts have been deployed on Ethereum [34]. However, the Top5 smart contracts are invoked by up to 37% of transactions [30]. If the execution of these hotspot contracts were no longer constrained by I/O bottlenecks, overall performance could be dramatically improved.

The operand of SLOAD is a 256-bit hash, as shown in Fig. 7(c). We trace the operands of the SHA3 instruction and the data distribution in Memory to get the value before the hash. The overhead by this tracking process is minimal due to three reasons: Firstly, the number of objects to track is small. After static analysis of the bytecode, unfetchable SLOAD instructions are flagged and then tracked. Secondly, the trace scope is limited. Accesses to variables are translated into a bytecode sequence with a fixed pattern, consisting of MSTORE→SHA3→SLOAD, interspersed with stack instructions. Thirdly, only hotspot contracts are tracked. Given that static analysis already covers a sufficient number of state variables, it is not worthwhile to incur overhead by optimizing a small subset of accesses in less frequently used contracts.

It should be noted that smart contracts can internally call other smart contracts, similar to function nesting. However, such invocation information is not reflected in the transaction attributes, which only explicitly label the entry point of the call (i.e. *To*). Therefore, the access table will also keep track of the internal calls of the smart contract. These smart contracts will be analyzed later.

4.1.4. Prefetch table

The prefetch tables are maintained in Cache and persisted on disk, recording prefetching information for all smart contracts. Initially, the Cache and disk prefetch tables are empty. Over time, the prefetch results of smart contracts are populated into the prefetch table. If the transaction invokes a smart contract that already exists in the table, the prefetch is completed directly based on the contents of the table. The entry is evicted from the Cache prefetch table when it becomes full, following the principle of least recently used (LRU). The evicted entries are then persisted to the disk prefetch table. The specific structure of the prefetch table is illustrated in Fig. 5.

Function identifier. A smart contract is typically comprised of multiple state variables and a series of functions. A transaction invokes one of these functions, and its internal logic is executed to modify the state variables. While nesting of functions is allowed, there exists only one entry function, with its identifier encoded in the first four bytes of *Input*. Based on this identifier, it becomes possible to determine the approximate execution path of the transaction. The prefetching process only needs to target the access objects of SLOAD instructions located on the path, thereby effectively reducing the data size of prefetching.

Internal calls. The addresses of other contracts that are internally called during execution are recorded. As in the previous analysis of variables, these contract addresses are fixed values. Therefore, these call relationships are also fixed. The status data of this contract and the internal contracts will be prefetched simultaneously.

The disk prefetch table theoretically contains the prefetching results of all smart contracts. However, according to Ethereum statistics, fewer than 0.5% of accounts are active on a daily basis [34]. This substantial number of "zombie" accounts implies that the vast majority of smart contracts are only invoked immediately after deployment and are seldom or never invoked thereafter. Therefore, it is unnecessary to allocate storage space for storing the prefetching results of these smart contracts. Hence, when the size of the prefetch table in disk exceeds the threshold, some entries are removed based on the LRU principle. Even if future transactions invoke such zombie contracts, only one additional quick static analysis is performed, which incurs a minimal overhead.

Furthermore, to facilitate fast lookups, a Bloom filter is constructed for the disk prefetch table [37]. When a smart contract address is not found in the Cache prefetch table but matches an entry in the Bloom filter, its corresponding entry on disk is once again written back into Cache.

4.1.5. Other access instructions

The access scope of the smart contract is limited to its own private key–value store and the global account state. The majority of accesses occur in KVT through the SLOAD instruction. Additionally, the smart contract instruction set includes instructions like BALANCE, EXTCODECOPY, EXTCODEHASH, etc. These instructions retrieve data such as the balance and contract code from the specified address in AST. The operands of these instructions are of address type. Prefetching is performed through static analysis.

4.1.6. Prefetch for complex smart contracts

In complex scenarios, such as the deeply nested cross-contract calls, the effectiveness of prefetching based on static instruction traceback may decline, because the target contract address or storage key may be determined dynamically at runtime. To mitigate this issue, MOP records internal call relationships when constructing the prefetch table, enabling the joint prefetching of frequently occurring call chains. Meanwhile, the access table learns historical access patterns of hotspot contracts and provides high-probability coverage for accesses that are difficult to resolve through static analysis. Even in extreme cases where prefetching is incomplete, the system can still fall back to the conventional MPT access path without affecting execution correctness. In addition, since prefetching is performed during the consensus phase, its additional overhead is limited; in the worst case, the system simply degrades to the standard MPT access path. Therefore, the proposed method remains robust and applicable in complex execution scenarios such as deeply nested cross-contract calls.

4.2. Lazy update of state

In contrast to read operations, the bottleneck for write operations is not at the bytecode level. The SSTORE instruction writes only key–value pairs to Cache. After all transactions have completed, these key–value pairs are written back into the disk, and the root is computed for all KVTs and AST. The primary objective of this mechanism is to facilitate quick rollbacks. The execution of a transaction may be interrupted due to factors such as insufficient gas, resulting in the invalidation of modified states. Similarly, for an entire block, if it fails validation, the modified state of all transactions within the block becomes invalid.

Within the same tree, multiple paths from different leaves to the same root can be updated concurrently. Additionally, updates across multiple KVTs are independent and can therefore be executed concurrently. However, it should be noted that the update of AST must wait for the completion of the updates of all KVTs. The overall time cost of this process is still determined by the two longest paths in KVTs and AST. This includes the disk I/O cost associated with reading and writing intermediate nodes as well as the hash calculation cost.

4.2.1. Modified state tree

Instead of updating AST and KVTs, a separate MPT called MST is constructed. MST includes all modified states in the AST and each KVT. The key–value stores of smart contracts adhere to the same key coding rules, and their respective key–value stores are independent of one another. In order to distinguish key–value pairs belonging to different smart contracts but having the same key, leaves in MST are encoded uniformly as triples $(flag, key, value)$. For states in AST, $flag$ is set to 0; for states in KVTs, $flag$ represents the smart contract address.

In the original design, the block header contains the root of the updated AST. In MOP, a forest is constructed with the unupdated AST and the newly constructed MST, and the new root of the forest is stored in the block header. If two forests have the same root, it indicates that they have identical initial states and modified states, ensuring that they will reach an identical new state.

After computing the forest root, distributed nodes can skip the time-consuming write process and proceed directly to the next consensus phase. The actual write operations are performed concurrently with the consensus phase, thus obtaining the updated AST and KVTs before the next execution phase begins.

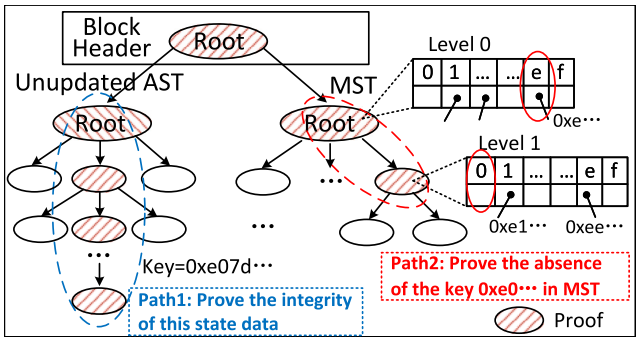


Fig. 10. State forest and proof for state data.

It is obvious that MST increases storage redundancy. The initial state is S_0, and the state after n rounds of updates is S_n, with $S_0 + \sum_{i=0}^{n-1} \Delta S_i = S_n$. Each modified state ΔS_i corresponds to an MST, which increases storage overhead. Therefore, we propose that distributed nodes only retain the MST of the latest block, as the modified state represented by the MST will reappear in the AST and KVTs of the next block.

4.2.2. Authentication

In addition to ensuring consistency, another important purpose for constructing an MPT is authentication. The light nodes can authenticate the state data sent by the full nodes solely based on the root in the block header. When a subset of the global state S_i is requested by the light nodes, the proof is constructed based on the unupdated AST in $block_{i+1}$, as this AST corresponds to the initial state (S_i) of consensus in round $i+1$. When the latest state is requested, the proof is redesigned utilizing the characteristics of the MPT.

Since MST is not merged into the AST, there may be two candidate leaves for the same key, namely, the old state located in AST and the modified state located in MST. These two paths cannot be distinguished by the root node alone. Logically, if the path is from MST, it can be ascertained that the state has been updated. Conversely, if the path is from AST, additional evidence is necessary to ensure that the state does not belong to the set of modified states. Without this evidence, a malicious full node could potentially deceive a light node by providing an outdated state.

As shown in Fig. 10, the branch node of MPT contains up to 16 child pointers, one for each hexadecimal nibble of the key. The branch nodes permit the rapid determination of whether a given key is present. If the pointer corresponding to e in the level 0 branch node is empty, it can be demonstrated that there is no leaf in MST whose key has the prefix 0xe...; conversely, if the pointer corresponding to 0 in the level 1 branch node is null, it can be demonstrated that there is no leaf in MST whose key has the prefix 0xe0...; and so on. Consequently, when an unmodified state is requested, the full node should provide two proofs: one being the path from root to leaf in AST and another being the path from root to branch node at a specific level in MST.

The requested state is unmodified with high probability. To reduce the proof size, we introduce Bloom filters [37] for the set of updated states, which are stored in the block header and used for efficient querying of whether a certain state has been modified. The construction of two complete proofs is only necessary in the special case of false positives.

4.2.3. The efficiency of MST

In MOP, we construct an additional MST to decouple write-back from consistency verification and delay the write-back to the next

consensus phase. Given its importance, we analyze the efficiency in this section.

Fast construction of MST. Due to the limited size of the modified state, the number of layers and nodes of MST are significantly smaller than AST. This implies that the hash computation in this process is minimal. MST is constructed solely based on the modified state without any disk I/O.

Minimized additional overhead. High-overhead write back processes are deferred to the next consensus phase and executed in parallel. Only the latest MST is retained to minimize storage redundancy. Additional proof paths are only required for requests of the latest state with false positives, while proofs for other states remain unchanged from the original design.

4.2.4. Atomicity analysis of lazy update

In this section, we further explain how the mechanism handles system crashes and chain reorganization scenarios.

First, even if a crash occurs during write-back, the system can still guarantee state consistency. After the execution of each block, the system generates a verification root based on the unupdated AST and the constructed MST, and records it in the block header. This ensures that the state transition of the block is deterministic. After recovery, a node can reproduce the same updated state and complete the write-back operation either by re-executing the blocks whose write-back was not finished, or by synchronizing the latest blocks from other nodes and re-executing them. Since re-execution produces the same state as before the crash, the final persisted state remains consistent.

Second, in the case of chain reorganization, MOP adopts a rollback mechanism consistent with that of conventional blockchain systems. Since the MST contains only the modified state of the current block, and the system retains only the MST corresponding to the latest block, when a chain reorganization occurs, the node only needs to discard the affected block and its MST, and then re-execute the transactions on the new main-chain branch to restore the correct state. Therefore, delayed state updates do not introduce additional consistency or recovery risks, thereby preserving the atomicity of the lazy update.

5. Evaluation

5.1. Evaluation methodology

This section introduces the evaluation objectives and experimental setup of MOP. Specifically, we first present the key questions addressed in our evaluation, and then describe the experimental setup.

5.1.1. Evaluation objectives

Our evaluation primarily focuses on the following aspects:

(1) Can the analysis of state prefetching cover SLOAD instructions? How does state prefetching contribute to improved storage I/O and execution time across various workloads?

(2) Do lazy updates quickly verify state consistency? Are the computational and time costs of generating the MST sufficiently small?

(3) How does our MOP perform in read/write amplification and end-to-end performance compared to existing approaches?

(4) Does MOP impose noticeable memory pressure on the system? How does MOP behave under parallel execution scenarios, and can it maintain its performance advantage as parallelism increases?

5.1.2. Experimental setup

Since MOP focuses on optimizing the execution phase rather than inter-node coordination, all experiments are conducted in a single-node environment. Experiments are run on a server equipped with an AMD Ryzen Threadripper 3970X processor, 64 GB of DDR4 RAM, and SSD storage. Our prototype uses RocksDB as the underlying key–value store [38].

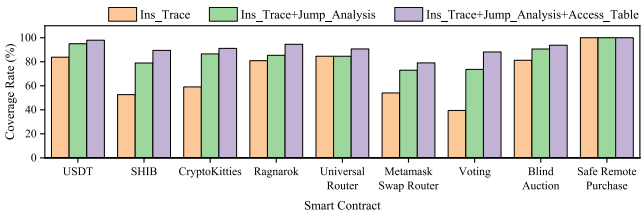


Fig. 11. Coverage rate of some typical smart contracts.

Table 2
Coverage of the three classes of SLOAD instructions in Fig. 7, and the average length of the traceback instruction queue.

Type of SLOAD	Num		Cover		Avg length of SLOAD	Avg length of MSTORE
1	583906	69.28%	551042	94.37%	3.17	0
2	28764	3.41%	23414	81.4%	10.92	8.21
3	230132	27.31%	202232	87.88%	7.82	11.29
Sum/Avg	842802	100%	776688	92.16%	4.38	11.05

We evaluate MOP using both synthetic workloads and real Ethereum trace-driven replay workloads. For the evaluation of data prefetching coverage, we first randomly select a set of Ethereum mainnet blocks from September 2023 to March 2024 and extract approximately 17000 smart contract bytecode for analysis, in order to measure the coverage of the prefetching mechanism for SLOAD instructions.

For execution performance evaluation, we use both synthetic and real workloads. For the real workload, we extract a subset of blocks from the Ethereum mainnet block range [18,000,000, 19,000,000] and replay them to recover the state accesses and I/O behavior during execution. This replay focuses on the performance overhead along the authenticated storage and execution critical path, rather than a full online replay including network propagation, mining, and consensus. For the synthetic workload, we initialize state sizes of 1M, 5M, and 10M accounts according to the specific experiment, and in some experiments further extend the state scale to 20M. The transaction types include token-transfer transactions and smart-contract transactions, with a ratio of 2:3. In the execution performance experiments, each round processes approximately 5000 transactions.

5.2. State prefetching

5.2.1. Coverage rate

We evaluate the coverage rate of the SLOAD instruction under instruction traceback and the access table. The coverage rate is defined as the ratio of the number of prefetched SLOAD instructions to the total number of SLOAD instructions in the bytecode. We randomly select Ethereum mainnet blocks spanning September 2023 to March 2024 and collect nearly 17000 smart contract bytecode.

The coverage rates for popular Ethereum smart contracts, such as ERC20, ERC721, ERC1155, voting, auction, and exchange contracts, are shown in Fig. 11. Overall, the state prefetching mechanism of MOP consists of two parts.

The first part is a static analysis method based on bytecode-level instruction traceback. For analyzable contracts, this method enables accurate data prefetching. Experimental results show that, on average across the sampled smart contracts, instruction traceback alone enables operand analysis for approximately 70% of SLOAD instructions. We observe that the majority of traceback failures are attributed to path aborts. The JUMPDEST instruction marks a valid jump destination for some JUMP instruction. When the operand of a JUMP instruction, i.e., the target address, can be obtained through static analysis, encountering a JUMPDEST instruction during traceback that cannot be matched with the target address set of the corresponding JUMP

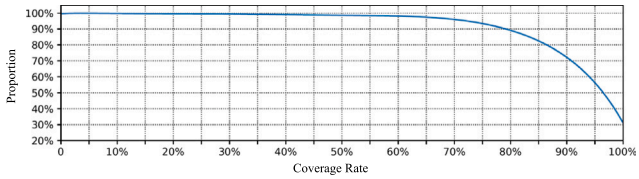


Fig. 12. Distribution of coverage rate. y percentage of smart contract whose coverage rate exceed x.

instruction signifies that the traceback process is terminated. As shown in Fig. 11, after further incorporating optimizations such as jump analysis, the average prefetch coverage across these smart contracts can be improved to approximately 85.3%. The second part targets smart contracts with high complexity and strong dynamic behavior. When static analysis cannot provide effective coverage, MOP uses the access table to record the historical access patterns of hotspot contracts and thereby assist in inferring prefetch targets. For the remaining SLOAD instructions that cannot be prefetched through static analysis, prefetching can still be performed if the access table indicates that they are associated with a constant or predetermined value. During the early stage of execution, especially in the cold-start phase, the effectiveness of prefetching is constrained because the access table contains only limited historical information. As historical access patterns continue to accumulate, the prefetching accuracy gradually improves and eventually stabilizes. As shown in Fig. 11, after introducing the access table, the average prefetch coverage can be further improved by about 6.35 percentage points, reaching approximately 91.65%. Therefore, when static analysis is already able to accurately prefetch the vast majority of smart contracts, this method serves as an important complement to data prefetching. It should be noted that inference based on historical access patterns is essentially a heuristic method, and its effectiveness is generally lower than that of deterministic static-analysis-based prefetching. Nevertheless, it can still effectively cover cases that are difficult to handle with static analysis, thereby improving the overall execution efficiency.

As shown in Table 2, our approach achieves 92% coverage across the three classes of SLOAD instructions illustrated in Fig. 7. The shortest dependent instruction queue lengths for the three classes of SLOAD instructions with successful traceback are 2, 6, and 4, respectively. Experimental results indicate that the average length ratio is approximately 1 : 3.44 : 2.47, which aligns with our expectations. We also traceback the MSTORE instructions associated with the second and third SLOAD classes. Since this instruction has two operands, its queue length is typically more than twice as long as that of SLOAD. Overall, the queue length is usually within 10, indicating a limited range of traceback. The distribution of coverage rates across approximately 17000 contracts is shown in Fig. 12. Our approach achieves over 80% SLOAD instruction coverage for 90% of smart contracts. In addition, we further analyze 20 representative DeFi contracts and commonly used infrastructure contracts in the DeFi ecosystem. The results, as shown in Fig. 13, indicate that most DeFi-related contracts can still achieve effective prefetching performance. However, for contracts with complex call paths and highly dynamic access patterns, the prefetching effectiveness degrades significantly.

When facing the extreme worst-case scenario in which static analysis fails and effective inference based on historical access patterns is also difficult, we analyze the performance of MOP from the perspective of its execution workflow. Specifically, MOP performs state prefetching during the consensus stage and prioritizes looking up prefetched results in the Cache during the execution stage. If prefetching fails, the system falls back to the conventional execution path, namely reading states from the on-disk state trie. Therefore, in the worst case, the execution overhead mainly consists of Cache lookup and disk access. Since the time cost of Cache lookup is negligible compared with disk I/O and

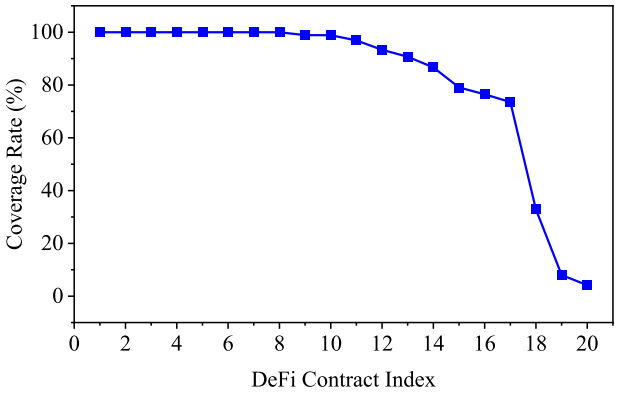


Fig. 13. Coverage rate for DeFi-related smart contracts.

Table 3
Transaction execution performance.

Workloads	Read amplification			Execution time (us)		
	MPT	MOP	Reduction	MPT	MOP	Speed up
1 m	2.415	0.238	90.14%	59.6	37.6	1.59
5 m	3.057	0.281	90.81%	65.8	37.9	1.74
10 m	3.259	0.296	90.92%	67.7	38.1	1.78
Real	2.16	0.399	81.53%	56.5	38.0	1.49

the associated read amplification, MOP degrades to an execution path that is essentially the same as that of the original MPT-based scheme in the worst case, without introducing any other significant overhead.

5.2.2. Transaction execution performance

We evaluate the performance speedup of state prefetching on transaction execution. The speedup is directly related to the number of layers of the MPT; therefore, we initialize states of different scales. Among them, 1 m, 5 m and 10 m represent randomly generated account addresses for one million, five million, and ten million respectively. We randomly select two distinct addresses as the sender and receiver and generate a token-transfer transaction with a nonzero transfer amount. Several common types of smart contracts in Fig. 11 are selected. The smart contract transactions invoke them with approximately uniform frequency, and the input parameters are randomly generated within a reasonable range. The rate of token transfer transactions to smart contract transactions is 2:3 [34]. Each time around 5000 transactions are processed. In addition, we further test transactions in real Ethereum scenarios, replaying some blocks 18000000-19000000.

To highlight the benefits of prefetching, we evaluate only transaction execution performance and exclude state updates. Table 3 illustrates the reduction in read I/O and the speedup achieved through the implementation of state prefetching. The increase of state size only affects a small fraction of read I/O caused by prefetching failures, so the performance of our method remains largely stable. Compared with the random datasets, the real dataset exhibits higher access locality, and thus workloads with more frequent MPT reads benefit more from prefetching. The internally invoked smart contracts recorded in the Cache access table are deferred for analysis. Prefetching for these contracts during this interval is not feasible. As a result, the performance of our method suffers partially on the real dataset.

5.3. Lazy updates

We focus on performance along the critical path. Concurrent write-back during the consensus phase does not affect the overall performance of the system. Although the MST hashes all nodes in the tree, its smaller height reduces the computation on the critical path compared with write-back updates, which require two rounds of root-to-leaf path

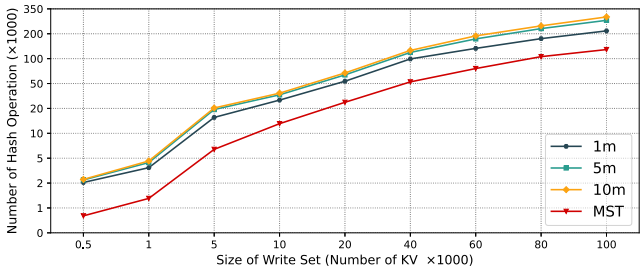


Fig. 14. Number of hash operation for various sizes of write set.

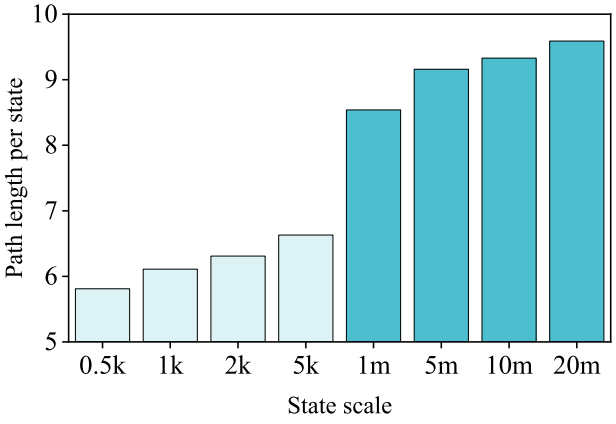


Fig. 15. Average state proof lengths of the MST and AST under different state scales. The light-blue bars denote the average proof length of the MST under different state scales, whereas the dark-blue bars denote the average proof length of the AST under different account scales.

recomputation. As shown in Fig. 14, the MST decreases the hash computation by 35%–70% compared to write-back across various state scales. Moreover, the MST can be rapidly constructed without incurring I/O overhead. Therefore, the time required for verifying consistency is reduced by over 90%.

Since lazy update introduces a newly constructed MST, the authentication procedure for light nodes requesting the latest state must be adjusted accordingly. As described in Section 4.2.2, if the requested state has been modified in the current execution round, the full node can return the state directly from the MST, and the corresponding proof path in the MST is sufficient for authentication. In contrast, if the requested state has not been modified, the full node must provide the proof path of that state in the unupdated AST and further prove that the queried key does not belong to the set of modified states represented by the MST, thereby preventing a stale value in the AST from being accepted as the latest state.

Experimental results are shown in Fig. 15. The MST stores only the states modified in the current round, whereas the AST maintains the global state. Consequently, the proof length in the MST is significantly shorter than that in the AST. In our simulated datasets, the average proof length of the MST is about 60.58%–77.63% of that of the AST. When a light node requests a modified state, the full node can return the corresponding proof path directly from the MST. Compared with the original MPT-based proof, the shorter MST proof reduces the authentication overhead at the light node. For an unmodified state, without Bloom-filter-based optimization, the light node needs to verify both the membership proof in the AST and the non-membership proof in the MST. This increases the authentication cost. In our simulated datasets, this worst-case cost is about 1.61×-1.78× that of the original MPT-based authentication. However, MOP maintains a Bloom filter for the modified state in the block header to efficiently test whether a

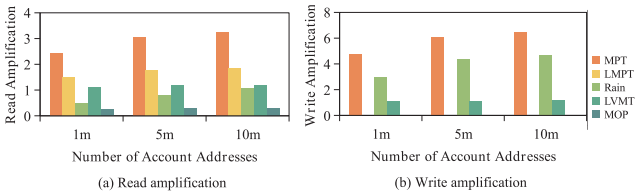


Fig. 16. Compared to other work. (a) Read amplification; (b) Write amplification.

Fig. 17. Compared to other work in terms of end-to-end performance.

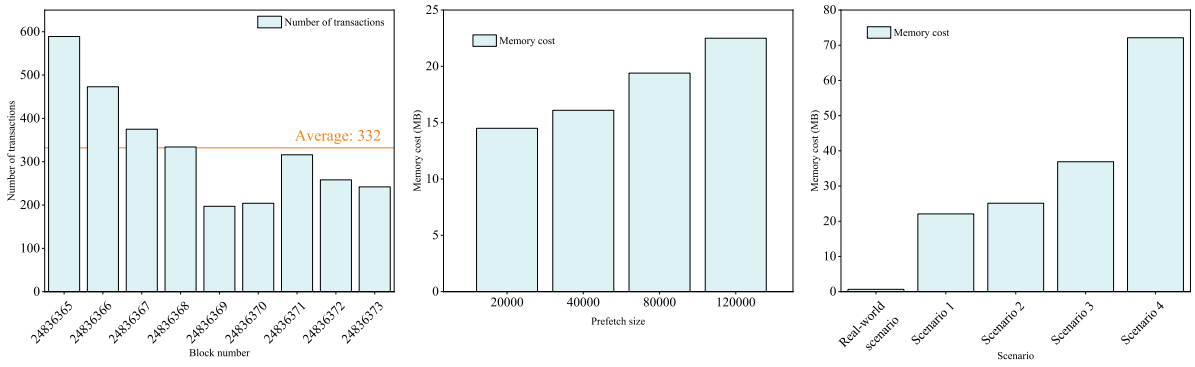
queried state may have been modified. Therefore, an additional non-membership proof in the MST is required only when the Bloom filter yields a false positive. As a result, the extra authentication overhead introduced by lazy update arises only in rare cases and remains limited overall.

5.4. Compared to other work

We compare read/write amplification and end-to-end performance with other works, LMPT [22], Rain [23] and LVMT [18]. LMPT maintains a small MPT in memory where each read and write operation is initially processed, while writes are subsequently batch-processed through background threads. Rain transforms disk I/O into distributed storage I/O by assigning read states, executing transactions, and writing states to different distributed nodes for completion. LVMT combines a multi-level AMT and a series of append-only MPTs to enable reads and writes with significantly lower overhead.

The comparison results are shown in Fig. 16. The read and write performance of MPT, LMPT, and Rain all degrade as the state size gradually increases. LVMT uses a two-level AMT that can accommodate up to 21 billion keys, so its read and write amplification remains largely unaffected. However, the time per read and write increases accordingly. LMPT primarily benefits from its caching strategy on the read path, but its cache hit rate is relatively low. Similar to our design, the write operation in LMPT occurs outside the critical path, resulting in a write amplification of zero. Rain achieves state prefetching through transaction pre-execution. However, the accuracy of this prefetching is limited and there is significant overhead associated with such pre-execution in high-throughput scenarios. Although Rain parallelizes the write process across multiple distributed nodes, these remain on the critical path for the blockchain system. Therefore, during our simulation experiments, we introduce a fixed network delay [39] for Rain.

We further conduct end-to-end performance evaluation, and the results are shown in Fig. 17. Compared with the original MPT scheme, MOP achieves performance improvements of 60.46%, 66.84%, and 68.06% under different account-scale settings, respectively. Compared with LVMT, the most efficient baseline among the competing algorithms, MOP also delivers performance gains of 30.50%, 30.74%, and 36.34%, respectively. This is because the MOP scheme not only substantially improves read efficiency through data prefetching, but also



(a) Number of transactions in real blockchain blocks (b) Memory overhead under different prefetch sizes (c) Memory overhead under different scenario settings

Fig. 18. Memory overhead analysis.

removes write operations from the critical execution path through lazy updates. As a result, the number of I/O operations during execution is significantly reduced, thereby avoiding extensive stalls caused by I/O waiting. Overall, compared with other methods, MOP achieves a speedup ranging from 1.57× to 3.13×.

5.5. Memory analysis

Since both data prefetching and lazy updates introduce additional memory overhead, analyzing peak memory usage is important for validating the effectiveness of MOP. In this section, we evaluate the memory overhead under different scenarios. To measure peak memory usage, we select the moment when block execution has finished but the MST has not yet been written back. At this point, the complete MST is still retained in memory, which reflects the maximum additional memory overhead introduced by MOP. As shown in Fig. 18(a), we collected statistics on the number of transactions in recent blocks and found that the average is 332 transactions per block. We therefore use this value to simulate memory overhead in the real-world scenario. To evaluate memory usage under complex conditions, all scenarios except the real-world scenario are configured with 20,000 transactions per block, while the real-world scenario adopts the average transaction volume observed in real blocks, namely 332 transactions per block.

We first evaluate the impact of different prefetch sizes on memory overhead by setting four prefetch-size configurations. As shown in Fig. 18(b), when the prefetch size is set to 120,000, the memory overhead is approximately 25 MB. This indicates that the additional memory overhead caused by changes in prefetch size is controllable and does not impose noticeable pressure on system deployment.

We further evaluate memory usage under different memory-pressure settings. As shown in Fig. 18(c), under the real-world settings, where each block contains 332 transactions and transaction complexity is relatively low, memory consumption is almost negligible. We construct four pressure scenarios, denoted as Scenario 1 to Scenario 4, in which transaction complexity increases progressively, leading to a steady rise in memory pressure. Across these scenarios, the average number of internal smart contract calls, the size of the prefetch table, and the size of the access table all increase gradually. In Scenario 4, the peak memory usage reaches approximately 72 MB. Although this value is significantly higher than that in the real-world scenario, it still remains well within a practical range.

5.6. Analysis under parallel execution

The core idea of MOP is to move MPT-related read and write I/O as much as possible out of the critical path of transaction execution through data prefetching and lazy updates, thereby improving system throughput. Since MOP primarily targets I/O overhead during execution rather than transaction scheduling or execution parallelism itself, it does not directly optimize the parallel execution mechanism. For this reason, MOP is naturally orthogonal and highly compatible with a wide range of parallel execution schemes. Their integration has the potential to further shorten the execution critical path and unlock additional throughput gains. This is also consistent with our core idea: MOP improves efficiency by decoupling critical-path I/O, whereas parallel execution exploits performance gains from computational parallelism.

Based on this observation, this section further analyzes the potential performance gains obtained by combining MOP with parallel execution schemes. Since MOP itself does not incorporate any dedicated optimization for execution parallelism, we adopt a coarse-grained parallel simulation to more clearly characterize their synergistic effects, where the horizontal axis represents different levels of parallel execution capability. It should be noted that this experiment is intended to estimate the potential upper-bound benefit of integrating MOP with parallel execution, and is therefore conducted under the following idealized assumptions: (1) MOP is able to complete the prefetching of all states required for execution during the consensus phase, so that read operations no longer constitute the primary bottleneck in the execution phase; (2) the parallel execution of transactions is conflict-free and introduces no additional overhead from conflict detection, rollback, or synchronization. Under these assumptions, we further configure each block to contain 20k transactions to validate performance under large workloads. The experimental results therefore more directly reflect the additional benefits that MOP can provide by reducing critical-path I/O as the degree of parallel execution increases.

The experimental results are shown in Fig. 19. When the parallelism level is set to 1, the system operates in serial execution. As the degree of parallelism increases, the throughput of MOP increases significantly. This is mainly because MOP removes most MPT-related I/O operations from the critical path of transaction execution through state prefetching and lazy updates, thereby substantially alleviating the I/O bottleneck during execution. Under such conditions, computation itself gradually becomes the dominant bottleneck, allowing transaction parallelism to be exploited more effectively, which significantly improves execution efficiency and further translates into higher system throughput. This observation is also consistent with the design objective of MOP, namely, to decouple state read/write overhead from the execution critical path through prefetching and deferred write-back.

In contrast, although the throughput of the original MPT scheme also increases with higher parallelism, the improvement is limited. This observation is consistent with prior studies [18,19], which show that state-read/write-related I/O accounts for more than 81% of the total transaction execution cost in blockchain systems. Therefore, for the original MPT scheme, the primary bottleneck is not insufficient computational parallelism, but rather the frequent and amplified I/O overhead of state access. In other words, even if transaction parallelism alone is optimized, the resulting performance gain remains significantly constrained by the I/O bottleneck. The key advantage of MOP

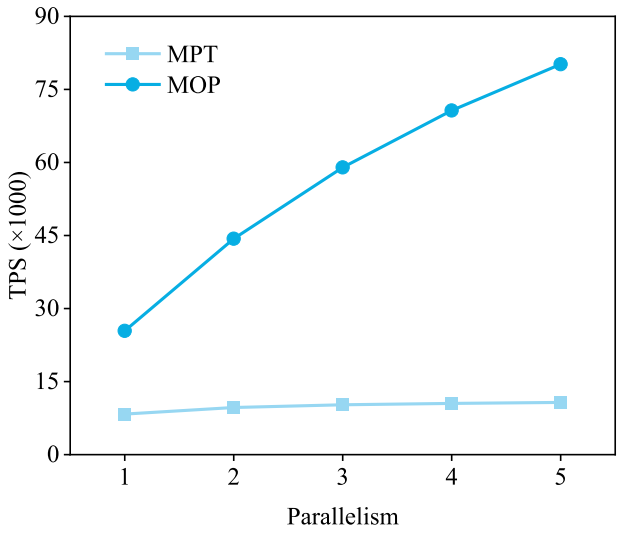


Fig. 19. Analysis under parallel execution.

lies in mitigating this limitation, thereby enabling more substantial throughput gains as parallel execution capability continues to improve.

6. Conclusions

Critical and time-consuming read and write operations dominate the transaction execution, becoming a new bottleneck for blockchain throughput. To address the problem, we introduce MOP to achieve efficient blockchain transaction execution by mitigating MPT I/O overhead on the critical path. Specifically, by implementing data prefetching and lazy updates, MOP parallelizes read and write operations with the consensus phase, thereby mitigating I/O overhead out of the critical path, improving the efficiency of transaction execution. Extensive experimental results demonstrate that MOP can achieve a throughput improvement ranging from 1.57×–3.13× compared to existing approaches.

MOP is designed to mitigate state-access I/O overhead on the execution critical path, and its applicability is therefore not restricted to Layer-1 systems alone. In particular, if a Layer-2 system still adopts an Ethereum-style execution environment with EVM-compatible execution and trie-based state management, and state read/write operations remain a major source of execution latency, the core ideas of MOP, including prefetching and lazy update, may still be beneficial. On the other hand, for Layer-2 systems whose performance bottlenecks mainly arise from proof generation, proof verification, or data-availability-related overhead, the benefits of MOP may be less significant. Exploring how to integrate MOP into representative Layer-2 execution environments and quantitatively evaluate its effectiveness under different bottleneck distributions will be an important direction for our future work.

CRediT authorship contribution statement

Ze Yin: Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Chubo Liu:** Writing – review & editing, Supervision, Project administration. **Kai Zhong:** Writing – review & editing. **Leilei Du:** Writing – review & editing. **Rui Pan:** Writing – original draft, Visualization, Validation, Software, Methodology, Data curation. **Keqin Li:** Writing – review & editing, Supervision, Project administration, Funding acquisition. **Kenli Li:** Writing – review & editing, Supervision, Project administration.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported in part by the National Key Research and Development Project of China under Grant 2021YFA1000600, and in part by the Programs of National Natural Science Foundation of China under Grant Nos. 62441234 and 62532005.

Data availability

The authors do not have permission to share data.

References

[1] W. Jie, W. Qiu, A.S.V. Koe, J. Li, Y. Wang, Y. Wu, J. Li, A secure and flexible blockchain-based offline payment protocol, IEEE Trans. Comput. 73 (2) (2024) 408–421.
[2] X. Fu, H. Wang, P. Shi, X. Zhang, Teegraph: A blockchain consensus algorithm based on TEE and DAG for data sharing in IoT, J. Syst. Archit. 122 (2022) 102344.
[3] S. Ghosh, S.H. Islam, A.V. Vasilakos, Blockchain-assisted provably secure identity-based public key encryption with keyword search scheme for medical data sharing, J. Syst. Archit. (2025) 103619.
[4] B. Yin, T. Xie, Supporting efficient and verifiable keyword queries on dynamic blockchain data, J. Syst. Archit. 175 (2026) 103781.
[5] G. Lin, H. Wang, J. Wan, L. Zhang, J. Huang, A blockchain-based fine-grained data sharing scheme for e-healthcare system, J. Syst. Archit. 132 (2022) 102731.
[6] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, Q. Li, Y. Hu, Make web3.0 connected, IEEE Trans. Dependable Secur. Comput. 19 (5) (2022) 2965–2981.
[7] T.D. Tran, P.-D. Bui, V.H. Pham, EdgeTrust-Shard: Hierarchical blockchain architecture for federated learning in cross-chain IoT ecosystems, J. Syst. Archit. (2026) 103701.
[8] Y. Liu, Z. Jia, Z. Jiang, X. Lin, J. Liu, Q. Wu, W. Susilo, BFL-SA: Blockchain-based federated learning via enhanced secure aggregation, J. Syst. Archit. 152 (2024) 103163.
[9] Z. Yin, H. Wang, C. Liu, Y. Ding, K. Li, K. Li, EBFL: An efficient blockchain framework for federated learning services, IEEE Trans. Serv. Comput. 19 (1) (2025) 281–294.
[10] M.J. Amiri, D. Agrawal, A.E. Abbadi, Permissioned blockchains: Properties, techniques and applications, in: SIGMOD '21: International Conference on Management of Data, Virtual Event, ACM, 2021, pp. 2813–2820.
[11] J. Liu, K. Ren, Improving blockchains with client-assistance, IEEE Trans. Comput. 71 (5) (2022) 1230–1236.
[12] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2008, https://bitcoin.org/bitcoin.pdf.
[13] Vitalik Buterin, Ethereum whitepaper, 2024, https://ethereum.org/en/whitepaper/.
[14] Y. Chen, Z. Guo, R. Li, S. Chen, L. Zhou, Y. Zhou, X. Zhang, Forerunner: Constraint-based speculative transaction execution for ethereum, in: SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, ACM, 2021, pp. 570–587.
[15] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, N. Zeldovich, Algorand: Scaling Byzantine agreements for cryptocurrencies, in: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017, ACM, 2017, pp. 51–68.
[16] L. Jia, Y. Liu, K. Wang, Y. Sun, Estuary: A low cross-shard blockchain sharding protocol based on state splitting, IEEE Trans. Parallel Distrib. Syst. 35 (3) (2024) 405–420.
[17] T. Cai, W. Chen, J. Zhang, Z. Zheng, SmartChain: A dynamic and self-adaptive sharding framework for IoT blockchain, IEEE Trans. Serv. Comput. 17 (2) (2024) 674–688.
[18] C. Li, S.M. Beillahi, G. Yang, M. Wu, W. Xu, F. Long, LVMT: An efficient authenticated storage for blockchain, in: 17th USENIX Symposium on Operating Systems Design and Implementation, USENIX Association, 2023, pp. 135–153.
[19] A. Li, J.A. Choi, F. Long, Securing smart contract with runtime validation, in: Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, ACM, 2020, pp. 438–453.

[20] R.C. Merkle, A digital signature based on a conventional encryption function, in: Advances in Cryptology - CRYPTO '87, a Conference on the Theory and Applications of Cryptographic Techniques, in: Lecture Notes in Computer Science, vol. 293, Springer, 1987, pp. 369–378.
[21] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, I. Abraham, mLSM: Making authenticated storage faster in ethereum, in: 10th USENIX Workshop on Hot Topics in Storage and File Systems, USENIX Association, 2018.
[22] J.A. Choi, S.M. Beillahi, S.F. Singh, P. Michalopoulos, P. Li, A.G. Veneris, F. Long, LMPT: A novel authenticated data structure to eliminate storage bottlenecks for high performance blockchains, IEEE Trans. Netw. Serv. Manag. 21 (2) (2024) 1333–1343.
[23] S. Ponnapalli, A. Shah, S. Banerjee, D. Malkhi, A. Tai, V. Chidambaram, M. Wei, RainBlock: Faster transaction processing in public blockchains, in: 2021 USENIX Annual Technical Conference, USENIX Association, 2021, pp. 333–347.
[24] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum Proj. Yellow Pap. 151 (2014) (2014) 1–32.
[25] P. Ruan, T.T.A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, B.C. Ooi, Blockchains vs. Distributed databases: Dichotomy and fusion, in: SIGMOD '21: International Conference on Management of Data, ACM, 2021, pp. 1504–1517.
[26] P.S. Anjana, H. Attiya, S. Kumari, S. Peri, A. Somani, Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory, in: Networked Systems - 8th International Conference, in: Lecture Notes in Computer Science, vol. 12129, Springer, 2020, pp. 77–93.
[27] P.S. Anjana, S. Kumari, S. Peri, S. Rathor, A. Somani, OptSmart: a space efficient optimistic concurrent execution of smart contracts, Distrib. Parallel Databases 42 (2) (2024) 245–297.
[28] T.D. Dickerson, P. Gazzillo, M. Herlihy, E. Koskinen, Adding concurrency to smart contracts, in: Proceedings of the ACM Symposium on Principles of Distributed Computing, ACM, 2017, pp. 303–312.
[29] T. Lu, L. Peng, BPU: A blockchain processing unit for accelerated smart contract execution, in: 57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020, IEEE, 2020, pp. 1–6.
[30] R. Pan, C. Liu, G. Xiao, M. Duan, K. Li, K. Li, An algorithm and architecture co-design for accelerating smart contracts in blockchain, in: Proceedings of the 50th Annual International Symposium on Computer Architecture, ACM, 2023, pp. 32:1–32:13.
[31] M. Bartoletti, L. Galletta, M. Murgia, A theory of transaction parallelism in blockchains, Log. Methods Comput. Sci. 17 (4) (2021).
[32] S. Baheti, P.S. Anjana, S. Peri, Y. Simmhan, DiPETrans: A framework for distributed parallel execution of transactions of blocks in blockchains, Concurr. Comput. Pr. Exp. 34 (10) (2022).
[33] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A.D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S.W. Cocco, J. Yellick, Hyperledger fabric: a distributed operating system for permissioned blockchains, in: Proceedings of the Thirteenth EuroSys Conference, ACM, 2018, pp. 30:1–30:15.
[34] Etherscan, Ethereum charts and statistics, 2024, https://etherscan.io/charts.
[35] Github, The solidity contract-oriented programming language, 2015, https://github.com/ethereum/solidity.
[36] Ethereum Improvement Proposals, Eip-20: Token standard, 2015, https://eips.ethereum.org/EIPS/eip-20.
[37] M. Mitzenmacher, Compressed bloom filters, IEEE/ACM Trans. Netw. 10 (5) (2002) 604–612.
[38] Facebook Database Engineering Team, Rocksdb: A persistent key-value store for flash and ram storage, 2022, https://rocksdb.org.
[39] Y. Huang, J. Zhang, J. Duan, B. Xiao, F. Ye, Y. Yang, Resource allocation and consensus on edge blockchain in pervasive edge computing environments, in: 39th IEEE International Conference on Distributed Computing Systems, IEEE, 2019, pp. 1476–1486.

Ze Yin received the M.S. degree in computer science and technology from Southwest University, China, in 2022. He is currently working toward the Ph.D. degree in computer science and technology at Hunan University, China. His research interests include blockchain, distributed computing, high-performance computing, privacy preservation, and artificial intelligence.

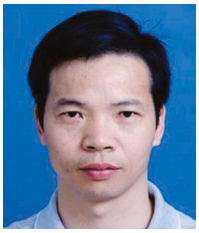
Chubo Liu received the B.S. and Ph.D. degrees in computer science and technology from Hunan University, China, in 2011 and 2016, respectively. He is currently a full professor with Hunan University. His research interests include parallel and distributed computing, computer architecture, artificial intelligence, game theory, and approximation and randomized algorithms. He has published over 40 papers in journals and conferences including the IEEE TPDS, IEEE TCC, IEEE TMC, IEEE TII, IEEE IoTJ, ACM ToMPECS, Theoretical Computer Science, ISCA, DAC, and NPC. He won the IEEE TCSC Early Career Researcher (ECR) Award in 2019.

Kai Zhong received the Ph.D. degree in computer science from Hunan University, Changsha, China, in 2024. He is currently a postdoctoral researcher at Hunan University. His research interests include parallel and distributed computing, cyber security, and reinforcement learning.

Leilei Du received the MS degree in computer science from Hunan University, China in 2020, and the Ph.D. degree in software engineering from East China Normal University, China, in 2024. He is currently a postdoctoral researcher at Hunan University. His research interests include privacy preserving, differential privacy, spatial crowdsourcing, searchable encryption.

Rui Pan received the Ph.D. degree in computer science and technology from Hunan University, China, in 2025, and the B.S. degree in computer science and technology from Southwest Jiaotong University, Chengdu, China, in 2020. His research interests include blockchain, game theory, and mobile-edge computing.

Keqin Li received the B.S. degree in computer science from Tsinghua University in 1985 and the Ph.D. degree in computer science from the University of Houston in 1990. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University, China. He has authored or coauthored over 1000 journal articles, book chapters, and refereed conference papers. He is an AAAS Fellow, an IEEE Fellow, an AAIA Fellow, an ACIS Founding Fellow, and a Member of Academia Europaea.



Kenli Li received the M.S. degree in mathematics from Central South University, China, in 2000, and the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003. He is a full professor with Hunan University. His research interests include parallel and distributed processing, supercomputing and cloud computing, and high-performance computing for big data and artificial intelligence. He has published more than 300 papers in international journals and conferences and serves on the editorial board of IEEE Transactions on Computers.