Maat: Analyzing and Optimizing Overcharge on Blockchain Storage

Zheyuan He[†], *Zihao Li*[‡*], *Ao Qiao*[†], *Jingwei Li*[†*], *Feng Luo*[‡], *Sen Yang*[†], *Gelei Deng*[§], *Shuwei Song*[†], *XiaoSong Zhang*[†], *Ting Chen*[†], *and Xiapu Luo*[‡]

[†]*University of Electronic Science and Technology of China*

[‡]*The Hong Kong Polytechnic University* [§]*Nanyang Technological University*

Abstract

Blockchain, such as Ethereum, relies on a transaction fee mechanism (TFM) to allocate the costs of on-chain resources, including storage, network, and computation. However, the inconsistency between the transaction fee and the storage workload results in overcharging issues for users.

In this paper, we present Maat, a tool designed to address these overcharging issues on blockchain storage. Maat employs three key techniques: (i) Fine-grained data collection, which captures detailed information on gas fees at the storage operation level (i.e., the operations interact with blockchain storage), enabling precise tracking of resource usage and charges for identifying overcharges; (ii) Consensus-oriented optimizations, which ensure that fee optimizations are consistent across all blockchain nodes by analyzing high-level storage semantics (e.g., accessing account and slot) of storage operations; and (iii) Resource pre-allocation, which ensures storage operation consistent across heterogeneous nodes and clients via preemptively specifying and allocating necessary resources. Extensive evaluations of Maat on Ethereum reveal a 32% reduction in transaction fees, amounting to 5.6M USD in weekly savings and nearly outperforming the baseline by nearly three times. Additionally, Maat achieves optimizations with a minimal performance overhead of 1.4% in block processing time and a 5.6% increase in memory consumption. Finally, Maat demonstrates its scalability, yielding a 31% reduction in transaction fees on Binance Smart Chain (1.54M USD per week).

1 Introduction

Blockchain [56, 70] is a distributed storage system that securely links *blocks* to ensure the authenticity of stored contents. Each block includes *transactions* [40] that are submitted by human users, to transfer money to others or run pre-defined programs (*smart contracts* [57]) over the blockchain system. In addition, since the blockchain system can evolve as users submit transactions [47], it persists different versions of *world states* [70] including its user information (e.g., account balances [45]) and contracts (e.g., the *bytecodes* [55] that are compiled by source codes) in a huge tree structure [73] that spans from memory to disks. For example, to execute a transaction that transfers funds from one account to another, the blockchain system first reads the balances of both accounts from the current world state, makes changes, and writes the new balances to the tree, to update the world state.

To prevent denial-of-service (DoS) attacks [58] for unlimited access and modification on world states (e.g., flooding spam transactions to exhaust disk I/Os), the blockchain adopts a *transaction fee mechanism (TFM)* [70] to charge users with some transaction fee based on the usage of on-chain resource (e.g., storage, computing, and bandwidth) [70]. The transaction fees will be measured in a unit termed *gas* [58] in the blockchain system [32]. For example, a fund transfer transaction is typically charged 21,000 gas [70]. Transaction fees for contract execution are determined based on the executed contract opcodes (i.e., the low-level instructions of the smart contract) [70], different contract opcodes are assigned varying gas fees, reflecting the resources they consume. The gas is converted into the corresponding cryptocurrency, which is then rewarded to the miners/ validators [70].

Motivation. Blockchain, like Ethereum, has been suffering from high transaction fees [53, 59, 68]. In 2023, Ethereum users incurred 2.4 billion USD in transaction fees [78]. No wonder Ethereum co-founder Vitalik said that *high fees are a huge problem and the main thing that prevents the platform from being used for cool stuff today is just the fees [15].*

Three approaches strive to optimize the transaction fees within blockchain systems: EIP-4844 [12], SuperStack [29], and EIP-2929 [11]. (i) EIP-4844 [12] proposes a cost-effective, albeit non-persistent, storage structure (termed blobs [12]) to alleviate the financial burden of transactions. (ii) SuperStack [29] optimizes smart contract code, achieving equivalent functionality with reduced gas consumption through more efficient code generation. (iii) EIP-2929 [11] adjusts gas costs for opcodes based on their de-facto storage

*Corresponding authors

loads [11], thereby reducing fees. For example, it decreases the gas fee for re-accessing an account within the same transaction from 2,600 gas to 100 gas, capitalizing on the fact that the account data is already resident in memory.

Drawing inspiration from EIP-2929, we delve into the discrepancy between actual storage loads and gas fees, which we term the ***overcharge issue***. Through an analysis of the implementations of major Ethereum clients (i.e., geth [4], besu [1], nethermind [5], and erigon [2]), we identify three overcharging issues, containing (i) continuous access to the same object within a block, (ii) continuous access to the same object across a limited number of blocks, and (iii) the deployment of duplicate contracts (§3). To measure the impact of these issues in real-world blockchains, we conduct an in-depth study of transactions on Ethereum for about four months (§3). Our results show that the overcharging issues affect 117M transactions (70.4%) and transaction fees of 147M USD (42.0%) on Ethereum. Note that the overcharge issue on Ethereum has a wide-spreading impact since TFM of Ethereum is reused in more than 500 blockchains [41, 72], including Binance Smart Chain (BSC) [8], Optimism [26], ETC [22], Polygon [27], and Avalanche [17]. Hence, we also investigate the impact of overcharging issues across blockchains by examining the overcharged fees on BSC (§3). Our results indicate that the overcharging issues affect 156M transactions (92.8%) and transaction fees of 11M USD (41.7%) on BSC.

Our work. In our study, we introduce Maat, a novel tool designed to mitigate overcharge issues in Ethereum's storage (§4). Maat leverages three key technologies: (i) A *fine-grained data collection technique* allows Maat to efficiently gather detailed information about gas fees and storage operations (§4.2). This method captures gas fees based on storage operations (i.e., the operations interact with blockchain storage) rather than opcodes, enabling a more fine-grained and precise reduction of overcharges. Moreover, this technique enhances overall efficiency by focusing on high-level semantics of storage operations (e.g., accessing slot and account) [69], rather than all storage components; (ii) A *consensus-oriented optimization technique* ensures that Maat adheres to the blockchain consensus when addressing overcharge issues (§4.3). This technique speculates the optimized gas fee by analyzing the relations of high-level semantics of storage operations. The gas fees optimized by Maat remain consistent across all blockchain nodes as the high-level semantics of storage operations are guaranteed by blockchain consensus; (iii) A *resource pre-allocation technique* ensures consistency of storage operations across different blockchain clients and heterogeneous nodes (§4.4). This technique specifies the parameters of the blockchain client (e.g., cache size) and pre-allocates the necessary resources (e.g., memory) to maintain consistent storage operations across all blockchain nodes.

Evaluation. We evaluate Maat in three aspects, including optimization effectiveness, performance overhead, and scalability (§5). First, we examine the optimization effectiveness of Maat on Ethereum (§5.1 and §5.2). The results demonstrate that Maat can effectively reduce transaction fees by 32%, resulting in a weekly savings of 5.6M USD on Ethereum and outperforming nearly three times the baseline of EIP-2929. Second, we evaluate the performance overhead introduced by Maat (§5.3). The results show that Maat exhibits efficient real-time capabilities with a 1.4% increase in block processing time and practical feasibility with an additional 5.6% memory allocation. Third, we assess the scalability of Maat by evaluating its optimization effectiveness on BSC in §5.4. Our results show a reduction in transaction fees by 31% (1.54M USD per week) on BSC. We port Maat to the 50 popular blockchains, which share the same codebase of TFM and storage components with Ethereum. Our results demonstrate that Maat effectively optimizes the overcharge issue on these platforms, underscoring its scalability and potential for seamless deployment across a wide range of blockchains.

Contributions. This paper makes the following contributions:

- We identify three overcharging issues in blockchain implementations. Measurements on Ethereum and BSC reveal that these issues affect more than 70% of transactions and contribute to 40% of fees, highlighting their significant impact on blockchain ecosystems.
- To address overcharging issues, we present Maat, a novel tool incorporating three key techniques: (i) A *fine-grained data collection technique* that enables Maat to efficiently gather fine-grained gas fee and storage operation data; (ii) A *consensus-oriented optimization technique* that ensures Maat adheres to blockchain consensus while optimizing overcharges; and (iii) A *resource pre-allocation technique* that maintains consistent storage operations across different blockchain clients and heterogeneous nodes.
- We assess the effectiveness of Maat on Ethereum, and find that it can reduce overcharged fees (e.g., 5.6M USD per week), outperforming the baseline by up to three times.
- Evaluations demonstrate that Maat possesses efficient real-time capabilities with minimal time overhead (only 1%), and showcases its scalability through seamless deployment across 50 popular blockchains.

Maat is released at https://github.com/hzysvilla/Maat. Our work has contributed to the proposal of EIP-7863 [10] within the Ethereum community.

2 Background

Ethereum. *Ethereum* [70] is the most popular blockchain allowing the transfer of cryptocurrencies (e.g., *Ethers*) as well as running programs (e.g., *smart contracts*). It has two types of accounts: (i) the *externally owned account (EOA)* [70], which is controlled by a human user; and (ii) the *contract account (CA)* [70], which is associated with a deployed smart contract [34]. Specifically, a human user can commit a *transaction* (via its EOA) to transfer Ethers to another EOA, create

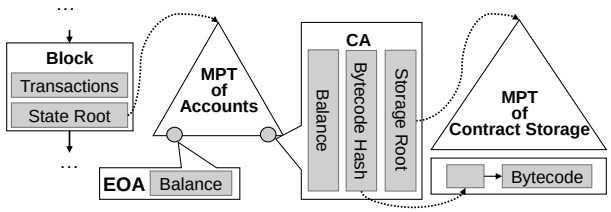


Figure 1: Overview of a world state.

a CA to deploy its smart contract [70] or run some already deployed contract. Ethereum chains transactions in the units of blocks (via cryptographic hashes) to provide data authenticity.

Each account has the *state* about its balance and deployed contract (for CA only) [45]. Specifically, the smart contract is a Turing-complete program that is developed in high-level languages [21] and then compiled into low-level *opcodes* [70] for deployment. To manage a deployed contract, the corresponding CA maintains two fields in addition to its balance: (i) the *bytecode hash* points to the corresponding bytecodes, which store the compiled opcodes of the contract; and (ii) the *storage root* points to the persistent storage (called *contract storage*), which stores variables for running the contract [47]. CA manages deployed bytecodes via a key-value store, and contract storage via a *Merkle Patricia Trie (MPT)* [73], which allows to verify the authenticity of variable values.

Ethereum stores account states of EOA and CA into a *world state*. The world state is initially empty and continuously updates after processing transactions [70]. Fig. 1 presents an overview of a world state. Each block includes a *state root* that refers to the world state after processing all transactions of the block [70]. Specifically, Ethereum manages all accounts of a world state via an MPT, and allows different MPTs (referred by state roots of distinct blocks) to share MPT nodes of unchanged accounts in the blocks [41, 75].

Ethereum introduced a *state snapshot accelerate structure (SSAS)* [9], which snapshots the up-to-date world state from MPTs into a flat key-value store [69] to mitigate the access performance overhead. Specifically, SSAS includes two types of key-value entries (collectively called *SSAS entries*): (i) the account entry, which maps the *address* (that uniquely identifies an account) to its balance; and (ii) the variable entry, which maps each variable defined in each contract to the value of the variable. Note that clients with different languages also possess similar SSAS components [3, 6, 20, 37]. We uniformly refer to such components as SSAS.

Transaction processing. World states evolve as processing transactions [31, 42, 70]. In the following, we detail cache mechanisms adopted in `geth` for accelerating the access and modification on states, followed by three operations on states during processing transactions.

Ethereum employs four cache mechanisms in memory to mitigate the disk I/Os when processing transactions: (i) *CoW cache*, which is generated by *copy-on-write (CoW)* strategy, and buffers accounts or variables at the current block. (ii) *SSAS cache*, which buffers SSAS entries corresponding to at most 128 recent blocks, to accelerate the access to balances and variables. (iii) *MPT cache*, which holds MPT nodes in memory to speed up the access to MPTs. (iv) *bytecode cache*, which maps bytecode hashes and the bytecodes, to improve the performance of loading bytecodes for CAs.

There are three operations for accessing and modifying states (denoted by **storage operations**).

- **Read states.** During processing transactions, Ethereum reads state data from world states, by querying two caches (i.e., CoW cache and SSAS caches). It retrieves the state values from persistent storage only if they are not in the caches. Besides, when reading bytecodes of CAs, Ethereum reads them from the bytecode cache or persistent storage (only if they are unavailable in the cache).
- **Update states.** Ethereum adopts CoW strategy to update SSAS cache [66]. Suppose that a transaction requires a change of the balance of an account. Ethereum (i) checks if there exists any in-memory *replica* of the corresponding account entry, (ii) creates a replica for the entry if it does not exist, and (iii) applies changes only to the replica in the CoW cache. If a transaction creates a CA, Ethereum directly adds the bytecodes of the corresponding CA into the bytecode cache for deployment. The newly changed account will be updated in SSAS cache and MPT cache after the block is executed. Then, the CoW cache is deallocated.
- **Persist states.** After processing all transactions of a block, the changed states in SSAS cache from 128 blocks ago will be written into persistent storage on disk. The changed states in MPT cache will write to disk at every block [36] or block interval [35] according to different strategies [47].

In our work, we focus on six state access behaviors (termed *high-level semantics*), i.e., account loads/stores, slot loads/stores, and bytecode loads/stores. These semantics are fundamental to understanding how blockchain interacts with the underlying state storage, as they represent the core operations involved in the transaction and smart contract execution [45].

Transaction fee mechanism. This paper focuses on the transaction fee mechanism (TFM) of Ethereum [53]. In decentralized networks like Ethereum, the possibility of malicious actors flooding the system with a massive number of transactions to exhaust critical resources such as I/O bandwidth and computational capacity poses a significant threat [41, 58]. To mitigate such risks, Ethereum's TFM is designed to impose a financial cost on resource consumption, thereby deterring attacks by making them economically prohibitive [67].

The core concept of TFM is *gas* [70], a unit that quantifies the resource consumption required to execute transactions and smart contracts. Each operation within a transaction is assigned a specific gas cost, reflecting its resource demand. When submitting transactions, users specify a gas price [70] in Ether [70], which determines the fund they are willing to pay per unit of gas consumed. The total transaction fee is the product of the gas consumed and the gas price.

Gas prices are tied to economic market factors, reflecting the dynamic supply and demand for blockchain resources [53]. Users specify gas prices to incentivize validators, with higher gas prices increasing the likelihood of quicker transaction inclusion. Conversely, gas costs are determined by the Ethereum specification and represent the fixed resource costs for executing specific operations, such as storage access [33]. While gas prices fluctuate based on market dynamics, gas costs are designed to align with the actual resource consumption, ensuring fairness and preventing the underpricing of operations that could otherwise compromise network stability.

EIP-2929. EIP-2929 [11] introduces key optimizations to TFM, aligning gas costs more closely with the actual resource loads imposed by storage accesses. One notable improvement is adjusting gas costs based on whether data has been accessed multiple times within the same transaction. Specifically, while the first access to an account or storage slot incurs the high gas cost (2,600 gas for an account and 2,100 gas for a slot, termed *disk fee*), subsequent accesses within the same transaction benefit from a reduced gas cost of only 100 gas (termed *momery fee*), as the data is already loaded into CoW cache. This approach not only reflects the lower resource demand of in-memory operations but also encourages efficient data access patterns, reducing the overall transaction cost for operations that repeatedly access the same data.

3 Motivation

Overcharge issues. Getting motivation from EIP-2929, we further explore the overcharge issue. We conduct an in-depth analysis across multiple implementations of the Ethereum client, including geth [4], erigon [2], nethermind [5], and besu [1]. By inspecting the codebase of TFM and the storage operations across these diverse implementations, we identified *three* types of overcharging issues that are prevalent within Ethereum. We list three issues as follows.

Issue#1 (continuous access to the same object in a block). Ethereum updates states using a *CoW* strategy, and flushes updated replicas to persistent storage per processing a block (§2). Thus, if an object (e.g., the balance of an account) is accessed multiple times during the processing of a block, the follow-up (i.e., except the first one) access will be responded to based on the CoW cache and should only cost memory fees. However, we find that, whether the object is originally stored in disks, Ethereum charges disk fees for follow-up access to the object, even if its replica is already in memory.

Issue#2 (continuous access to the same object across a limited number of blocks). Ethereum deploys SSAS cache to buffer the objects that are accessed during the processing of recent 128 blocks (§2). Thus, if an object is re-accessed during these blocks, the follow-up (i.e., except the first one) access should be charged for memory fees, since a copy of the object is buffered in SSAS cache. However, we find that Ethereum charges disk fees for accessing the objects that are already in the SSAS cache.

Issue#3 (deployment of duplicate contracts). Recall from §2 that Ethereum refers the bytecodes of deployed contracts via bytecode hashes in the corresponding CAs (Fig. 1). Thus, deploying a duplicate contract (that leads to the same bytecodes as existing contracts) only needs to refer to the corresponding physical bytecode copy without re-writing the redundant bytecodes into persistent storage. However, we find that Ethereum still charges disk fees to write bytecode contents for deploying duplicate contracts.

Pratical Impact. We evaluate the practical impacts of the three overcharging issues in real-world blockchains, i.e., Ethereum and BSC, which are the largest market capitalization blockchains among those reusing Ethereum's TFM. Specifically, we evaluate overcharging issues on Ethereum and BSC within the block range from #18M (Aug 26, 2023) to #19M (Jan 13, 2024) and the block range from #34.15M (Dec 07, 2023) to #35.15M (Jan 11, 2024), respectively.

We first run the original `geth` client and BSC's client [25] to synchronize the world state at the starting block heights (e.g., #18M for Ethereum) in the target blockchains. We then hack both clients to capture gas costs of operations on states (§2). We replay transactions in the follow-up 1M blocks based on our hacked clients, and count the overcharged fees and affected transactions by our uncovered three kinds of overcharging issue. We focus on two metrics to evaluate the impacts of the three issues, i.e., the fraction of transactions and the fraction of fees. The fraction of transactions is obtained from the ratio of (i) the number of transactions occurring in the corresponding case and (ii) the total number of transactions. The fraction of fees is derived from the ratio of (i) the transaction fees of the operations on states triggering the corresponding case and (ii) the total transaction fees.

The fraction of transactions for the union of three issues is 70.4% (roughly 117M transactions) and 92.8% (roughly 156M transactions) on Ethereum and BSC, respectively. The fraction of fees for the union of three issues is 42.0% (roughly 147M USD) and 41.7% (roughly 11M USD) on Ethereum and BSC, respectively. As the three issues can affect at least 70% of transactions and 40% of transaction fees on the two blockchains, it indicates that optimizing these issues can significantly save transaction fees for a wide range of users. We analyze the details of each case as follows.

– Issue#1. The 2nd row of Table 1 shows the fraction of trans-

Table 1: Impacts of overcharging issues in Ethereum and BSC

Overcharging issues		Ethereum	BSC
Issue#1	Fraction of transactions	43.7%	63.1%
	Fraction of fees	25.6%	23.7%
Issue#2	Fraction of transactions	42.4%	62.0%
	Fraction of fees	12.9%	15.3%
Issue#3	Fraction of transactions	0.5%	0.3%
	Fraction of fees	3.5%	2.7%
Total	Fraction of transactions	70.4%	92.8%
	Fraction of fees	42.0%	41.7%

actions for occurring Issue#1. There are 43.7% of total transactions (roughly 69M transactions) affected by Issue#1 on Ethereum. In contrast, Issue#1 affects 63.1% of total transactions on BSC (roughly 106M transactions). The difference in the fraction of transactions on BSC and Ethereum arises because transactions on BSC typically access more storage objects, increasing the likelihood of triggering Issue#1. Concretely, we observed that BSC's average gas cost per transaction is twice that of Ethereum, and more gas cost originates from more access to storage objects on BSC.

The 3rd row of Table 1 shows the fraction of transaction fees for Issue#1. On Ethereum, the fraction of fees for Issue#1 is 25.6% of total transaction fees (roughly 90M USD). Similarly, Issue#1 impacts 23.7% of total transaction fees (roughly 6M USD) on BSC. Although the fraction of transactions occurring Issue#1 on Ethereum is smaller than that on BSC, their fractions of fees are close. This is because each transaction on Ethereum triggers Issue#1 more frequently, resulting in more storage access per transaction for costing more fees. Concretely, transactions triggering Issue#1 on Ethereum have an average of 10.2 times storage access triggering this case, compared to 6.9 times on BSC.

– **Issue#2.** The 4th and 5th rows of Table 1 show the fraction of transactions for occurring Issue#2 and the fraction of fees for Issue#2. On Ethereum, there are 42.4% of total transactions (roughly 67M transactions) affected by Issue#2. In contrast, Issue#2 affects 62.0% of total transactions on BSC (roughly 104M transactions). Moreover, on Ethereum, the fraction of fees for Issue#2 is 12.9% of total transaction fees (roughly 45M USD). Similarly, Issue#2 impacts 15.3% of total transaction fees (roughly 4M USD) on BSC.

Although the fractions of transactions for both Issue#1 and Issue#2 are similar, the fraction of fees for Issue#2 is less than that for Issue#1. This difference arises because transactions are more likely to interact with state objects that were previously accessed within the same current block, triggering Issue#1, compared to state objects accessed in previous intervals of 128 blocks, which trigger Issue#2. Concretely, on Ethereum, transactions average 10.2 (resp. 6.9 on BSC) accesses to state objects within the current block, compared to only 5.3 (resp. 4.1 on BSC) accesses to state objects within previous intervals of 128 blocks.

– **Issue#3.** The 6th row of Table 1 shows the fraction of transactions for occurring Issue#3. There are 0.5% of total transactions (roughly 1M transactions) impacted by Issue#3 on Ethereum. Likewise, on BSC, Issue#3 affects 0.3% of total transactions (roughly 0.5M transactions). The minimal impact of Issue#3 on transactions can be attributed to the low occurrence (e.g., less than 2% of total transactions) of transactions for deploying contracts within blockchains.

The 7th row of Table 1 shows the fraction of fees for Issue#3. On Ethereum, the fraction of fees for Issue#3 is 3.5% of total transaction fees (roughly 12M USD). Similarly, Issue#3 impacts 2.7% of total transaction fees (roughly 1M USD) on BSC. Different from the first two issues, the fraction of transactions for Issue#3 (e.g., 0.5% on Ethereum) is smaller than the fraction of fees for Issue#3 (e.g., 3.5% on Ethereum). This is because transactions for deploying contracts are more expensive than transactions occurring in the first two issues, causing a higher proportion of transaction fees for Issue#3. Concretely, transaction fees for deploying contracts on Ethereum are typically 7 times (resp. 9 times on BSC) more expensive than fees of other transactions.

4 Design of Maat

We design Maat tailored to optimize the overcharging issues of blockchain. Maat aligns workloads of storage operations with their corresponding gas fees to conserve transaction fees for blockchain users. There are three *design principles* of Maat. (i) *Consensus.* Maat ensures that the charges and the resource utilization after optimization remain consistent across each client of the blockchain network. This prevents discrepancies in charges for the same operations across different clients, which will compromise blockchain consensus. (ii) *Fairness.* Maat guarantees the fees of storage operations are consistent with the real storage workloads, which avoids the financial burden caused when users pay excessive transaction fees [39]. (iii) *Efficiency.* Maat's optimization process on TFM for mitigating overcharges must be efficient without adversely affecting the blockchain's throughput and operation.

4.1 Overview

Maat addresses overcharging issues on blockchain storage operations by three techniques. (i) Maat adopts a *fine granular data collection technique* to efficiently obtain information on the gas fee and storage operations. This technique allows for a nuanced correlation of gas fee with its storage operations, rather than coarse-grained operations like opcodes, facilitating an in-depth sense of overcharging issues. Besides, this technique improves the efficiency of Maat by only collecting the high-level semantics (e.g., the access of the account or slot) of the storage operations, instead of inefficiently capturing the storage operations of all storage components. (ii) Maat utilizes an *consensus-oriented optimization technique* to solve the overcharging issue. This technique speculates the optimized gas fees by only analyzing the high-level semantics of storage operations. Given the consensus on the high-level semantics of storage operation (i.e., accessing each account and slot), this technique also ensures the optimization of gas fees to hold consensus in every blockchain node. (iii) Maat applies *a resource pre-allocation technique* to ensure that the actual storage operations on the blockchain align with the charged fee. Variations in client implementations and node hardware configurations of blockchain can lead to inconsistencies between optimized transaction fees and actual storage

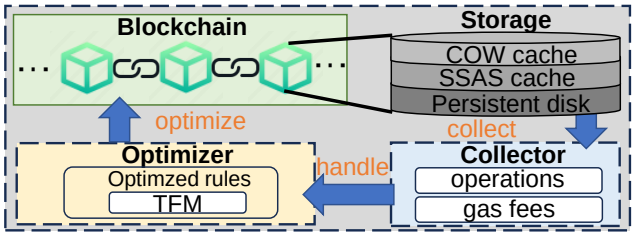


Figure 2: The architecture of Maat.

operations. This technique ensures consistency of optimization's storage operations by specifying execution parameters and pre-allocating the required resources to heterogeneous nodes and clients.

Architecture. Fig. 2 illustrates the architecture of Maat, which compasses three entities: blockchain clients with its storage, a collector, and an optimizer. (i) Blockchain clients. Maat will launch with the blockchain client. When the clients execute block and transaction, Maatwill monitor the blockchain with its storage operations and optimize its gas fee in real-time. (ii) Collector (§4.2). By monitoring the client, the collector gathers storage operations and fine-grained gas fees. Every storage operation corresponds to a gas fee, and the collector will dispatch them to the optimizer. (iii) Optimizer (§4.3). We deploy four optimization rules against the overcharging issue on the optimizer. The optimizer will handle the storage operations and gas fees by the four optimization rules. If the optimizer detects overcharging issues, Maat will optimize the corresponding gas fee. Note that efficiency plays a vital role in the deployment of Maat as it will deploy in every node of the blockchain network.

4.2 Collector

To optimize overcharging issues, the collector gathers storage operations alongside their gas fees for Maat. The naive method of collecting data [24,33,58] is to monitor the opcodes related to blockchain storage and all the storage components of the blockchain (i.e., CoW cache, SSAS cache, MPT cache, SSAS on disk, MPT on disk, and bytecode on disk). However, the naive method has two disadvantages as follows.

(i) Coarse granularity of gas cost. Within naive methods, Maat is unable to pinpoint the overcharging part from the coarse-grained gas cost. Concretely, these methods can only aggregate and report cumulative gas cost for opcodes, without distinguishing the gas cost in the level of storage operations. The coarse-grained gas cost of opcode can correspond to multiple storage operations, with only one being overcharged. For example, opcode `Call` has multiple storage operations with multiple charges, which can transfer ether (read and write account balances) and invoke contracts (read bytecodes of contracts). Without fine-grained insights, Maat cannot optimize the overcharged storage operation. **(ii) Inefficiency data collection.** Comprehensive monitoring of all blockchain storage components would introduce substantial overhead, which would compromise the Maat's design goal of high efficiency and degrade the performance of blockchain nodes.

Table 2: Charges for opcodes on blockchain storage.

opcode	$g_{memload}$ (gas)	$g_{memstore}$ (gas)	$g_{diskload}$ (gas)	$g_{diskstore}$ (gas)
sload	100	N.A.	2,100	N.A.
balance	100	N.A.	2,600	N.A.
extcodesize	100	N.A.	2,600	N.A.
extcodehash	100	N.A.	2,600	N.A.
extcodecopy	100	N.A.	2,600	N.A.
call/callcode	100	N.A.	2,600	9,000
delegatecall	100	N.A.	2,600	N.A.
staticcall	100	N.A.	2,600	N.A.
$call_{external}$	N.A.	N.A.	N.A.	9,000
sstore	100	100	2,100	2,900
create/create2	N.A.	N.A.	N.A.	L_{code}*200

$call_{external}$ indicates external transactions (i.e., EOA to EOA), L_{code} indicates the length of the bytecode.

To address the above two disadvantages, we propose the *fine granular data collection technique* to obtain gas costs and storage operations. Specifically, our technique breaks down the gas costs with fine granularity, segmented in alignment with each specific storage operation. Ethereum decouples four dimensions of storage operations and their corresponding gas costs for each opcode, containing memory loading/storing and disk loading/storing, as shown in Table 2. Inspired by the decoupling design, Maat captures the actual gas cost and storage operations at the granular level of Table 2. For instance, for the opcode `Call`, Maat records the specific storage operations performed, such as memory loads and disk accesses, along with their actual gas costs. By leveraging the insights from Table 2, Our technique bridges the gap of gas cost between opcode-level and operation-level, enabling accurate and efficient gas cost adjustments.

Besides, to improve the efficiency, Maat does not monitor every storage component to obtain all storage operations. Instead, Maat captures the high-level semantics of storage operations (i.e., account loads/stores, slot loads/stores, and bytecode loads/stores). This selective monitoring significantly enhances Maat performance by reducing overhead while still providing sufficient detail for the optimization of gas costs. Besides, high-level semantic operations are deterministic and consistent across all blockchain nodes, facilitating the optimization of overcharge issues (see §4.3).

4.3 Optimizer

By taking the break-up gas cost and its high-level semantic of storage operations from the collector, the optimizer wields optimization rules to address the overcharging issues in two steps. **Step 1:** The optimizer identifies the type of overcharge issue based on the high-level semantics of storage operations. **Step 2:** According to the type of overcharge issues, the optimizer tunes the break-up gas cost to their actual workload.

Table 3 details the four optimization rules, namely O1 to O4. The 1st column categorizes the optimization rules, and the 2nd column outlines the corresponding issues addressed by each optimization rule. The parts above and below the horizontal separator represent the conditions for triggering rules, and details about how to adjust the gas cost to address the overcharges, respectively. The red superscripted numbers partition conditions in each optimization rule. We elaborate on each optimization rule as follows.

- O1&O2 are designed to address overcharging issues caused by repeated memory accesses being incorrectly charged as disk fees in the CoW cache (i.e., Issue#1). Specifically, O1 handles memory reads, while O2 addresses memory writes.

Condition parts 1 to 3 of O1 denote the existence of two transactions tx_i and tx_j within block B, where tx_i uses opcode *ins* to access loc_i object in the world state, and tx_j later uses opcode *ins* to access loc_j object in the world state. Note that two opcodes *ins* in tx_i and tx_j can be different. Condition part 4 of O1 represents that the loc_i object is the same as the loc_j object, indicating the two transactions tx_i and tx_j access the identical object in the world state. Condition part 5 of O1 indicates that the object accessing of tx_j will be optimized as the memory fee (i.e., $g_{memload}$). Note that we introduced *ins*, *tx*, and B in the rule for clarity and the actual optimization relies solely on high-level semantics (i.e., access to *loc*).

- O3 mitigates overcharging caused by accessing objects stored in SSAS cache, where memory operations are mistakenly charged as disk operations (i.e., Issue#2). Condition parts 1 and 2 indicate the most 128 recent blocks (i.e., $[B_{n-128}, B_{n-1}]$) involves tx_i, and the recent block B_n includes tx_j. Condition parts 3 to 5 represent tx_i (resp. tx_j) uses *ins* read the loc_i (resp. loc_j) object, which loc_i object is the same as loc_j object. Condition part 6 denotes that the object reading of tx_j will be optimized as the memory fee of $g_{memload}$.

- O4 addresses overcharges associated with deploying duplicate contracts, where redundant disk fees are applied to identical contract bytecodes (i.e., Issue#3). Condition parts 1 and 3 of O4 represent, in previous blocks (i.e., $[B_0, B_{n-1}]$), contract SC_i has been deployed by transaction tx_i. Condition parts 2, 4, and 5 of O4 denote that, in the current block (i.e., B_n), contract SC_j has been deployed by transaction tx_j, and contract SC_j has identical bytecode with contract SC_i. Condition part 6 of O4 denotes that the charge for tx_j deploying the duplicate contract SC_j becomes a disk fee (i.e., $g_{diskload}$) for querying whether the contract exists in the world state.

Table 3: Optimization rules against overcharging issues.

Optimization rules	Issues
$O1: \frac{^1(tx_i, tx_j \in B) \wedge^2 (ins.load(loc_i) \in tx_i) \wedge^3 (ins.load(loc_j) \in tx_j) \wedge^4 (loc_i == loc_j)}{^5G[ins.load(loc_j)] := G_{memload}}$	Issue#1
$O2: \frac{^1(tx_i, tx_j \in B) \wedge^2 (ins.store(loc_i) \in tx_i) \wedge^3 (ins.store(loc_j) \in tx_j) \wedge^4 (loc_i == loc_j)}{^5G[ins.store(loc_j)] := G_{memstore}}$	Issue#1
$O3: \frac{^1(tx_i \in [B_{n-128}, B_{n-1}]) \wedge^2 (tx_j \in B_n) \wedge^3 (ins.load(loc_i) \in tx_i) \wedge^4 (ins.load(loc_j) \in tx_j) \wedge^5 (loc_i == loc_j)}{^6G[ins.load(loc_j)] := G_{memload}}$	Issue#2
$O4: \frac{^1(tx_i \in [B_0, B_{n-1}]) \wedge^2 (tx_j \in B_n) \wedge^3 (ins.create(SC_i) \in tx_i) \wedge^4 (ins.create(SC_j) \in tx_j) \wedge^5 (SC_i == SC_j)}{^6G[ins.create(SC_j)] := G_{diskload}}$	Issue#3

Tradeoff. There is a tradeoff to consider in setting the fee for deploying duplicate contract bytecodes in O4. Specifically, when determining whether the bytecodes of contracts to be deployed already exist in the world state, we introduce a newly added charge ($g_{diskload}$) to account for the operation's cost of querying the bytecodes in the world state. Consequently, while users deploying contracts with duplicate bytecodes can save on deployment fees by applying O4, users deploying non-duplicate contracts will incur an additional $g_{diskload}$ charge, imposing extra financial burdens on them.

However, the new charge is reasonable for two reasons. (i) Most deployed contracts are duplicated [43], hence, the extra charges will only occur in a few cases. In our experimental range (i.e., 1M blocks) in §3, 93% of Ethereum (88% for BSC) contracts are duplicated. It indicates that over 88% of contracts deployed by users can benefit from O4. (ii) The fee of deploying the contract is much higher than that of $g_{diskload}$ (i.e., 2,600 gas) [74]. We also observed that the average fee of deploying a contract on Ethereum is 131,582 gas (144,190 gas on BSC), which is over 50 times than $g_{diskload}$ on Ethereum (55 times for BSC). Hence, even for 12% users who deploy non-duplicate contracts, the newly added charges are minimal.

4.4 Design consideration

Then, we answer three design questions of Maat, containing charge consensus, resource consensus, and price reasonability.

Q1: How does Maat ensure that gas charges are consistent across all nodes? To ensure that gas charges are consistent across all blockchain clients, we harness the inherent data structures of the blockchain to formalize the conditions for optimization rules O1 through O4. Given that blockchain's data structures encompass blocks (B), transactions (tx), opcodes (*ins*), and state access locations (*loc*), these elements construct an integral framework maintained through blockchain consensus. This consensus mechanism ensures a deterministic and unique sequencing for these elements, thereby guaranteeing the applicability of the rules to all clients. Furthermore, we provide proof of the safety of these rules as follows.

Let $\mathcal{B} = \{B, tx, ins, loc\}$ denotes blockchain data, where B signifies blocks, tx denotes transactions, *ins* refers to opcodes, and *loc* indicates the location of state access. The consensus mechanism, denoted by $\mathcal{C}$, ensures that each element in $\mathcal{B}$ adheres to a unique and sequential order, i.e., $\mathcal{B} \vdash_{\mathcal{C}} \mathcal{U}$, where $\mathcal{U}$ represents the deterministic and unique sequence guaranteed by blockchain consensus. In this context, the conditions of four optimization rules are strictly defined as functions of blockchain data and are thus bound by the constraints imposed by $\mathcal{C}$. This signifies that $\forall x \in \{O1, O2, O3, O4\}, x \in \mathcal{B}$, ensuring that these rules are inherently secure and consistently

executable across all participating blockchain nodes.

Q2: How does Maat guarantee the storage operations are consistent across all nodes? O1, O2, and O3 depend on cache mechanisms to conduct optimization. However, the different hardware configurations and various blockchain clients (i.e., client diversity [64, 65]) will affect the functionality of the cache mechanism, which will lead to inconsistent optimization of the corresponding storage operations on different blockchain nodes. For instance, if a client does not cache the accessed slots and accounts in recent 128 blockchains, the O3 optimization doesn't work.

To guarantee the storage operations are consistent in all nodes, we propose a resource pre-allocation technique. The core idea of our technique is to speculate and pre-allocate the space required for optimization rules based on a limited block range and block gas limits. Concretely. O1 and O2 cache only the data accessed by the current block, whereas O3 caches data accessed by 128 nearby blocks. Hence, our method requires the blockchain client to cache the data accessed in 128 blocks and pre-allocate the corresponding memory. Moreover, we pre-allocate the maximum cache size required for each block based on the upper bound analysis, which shows that all gas of a block is used to read account data. In Ethereum, the gas of each block is limited to 30M gas, and the gas cost for reading an account is 2,600 gas. Each account occupies 164 bytes in memory, which includes 8 bytes for a memory pointer, 8 bytes for metadata, 20 bytes for the account address, 32 bytes for the nonce, 32 bytes for the balance, 32 bytes for the storage root, and 32 bytes for bytecode hash. Hence, the memory required to cache access data for one block is $\frac{30Mgas}{2.6Kgas} \times 164bytes$, resulting in approximately 1.80 MiB of cache space per block. By pre-allocating 230 MiB of memory ($1.80MiB \times 128$), Maat can ensure that the storage operation adheres to the optimization rules and maintains consistency across all nodes in terms of storage operations.

Besides, the re-allocated cache of our techniques builds upon the management strategy of Ethereum's SSAS cache (§2). Specifically, the pre-allocated cache organizes key-value entries for accounts and storage slots in a flat structure. Based on the cache, Maat enables deterministic behaviors (i.e., in memory or disk) for cache accesses of the storage operations across diverse implementations and hardware heterogeneity.

Q3: How does Maat keep the reasonability of gas cost? We follow Ethereum specification [11, 70] to set the optimized charges (e.g., $g_{memstore}$), which guarantees the reasonability of gas cost. Specifically, the current pricing adopted by Ethereum (e.g. EIP-2929 [11]) considers the scenario of accessing the cache within a transaction and defines the corresponding memory fee and disk fee. Therefore, we adhere to the current pricing mechanism. Consequently, in O1 to O3, we set the price of $G_{memstore}$ and $G_{memsload}$ at 100 gas, while the $G_{diskload}$ of O4 is set to 2,600 gas.

4.5 Implementation

We develop the prototype implementation of Maat based on go-ethereum v1.12.2 [16]. Maat is launched in multiple threads to bolster its operational efficiency, and its implementation consists of 2,717 lines of code.

- `Collector()`. Maat collects the high-level semantics of storage operations and their fee information by instructing `Colletor()` to the corresponding functions of storage components in go-ethereum. For instance, we obtain the account and slot accesses for in-memory SSAS cache by hooking `Colletor()` in function `Account()` and `Storage()` of package `snapshot`. Besides, we retrieve the fee information of `Call`'s execution by hooking `Colletor()` in function `gasCall()` of package `vm`.
- `Handler()`. After acquiring the collected information related to storage operations and their gas costs, `Handler()` transmits to `Optimizer()` by golang's channel (i.e., a communication mechanism between threads [54]).
- `Optimizer()`. For each high-level semantic of a storage operation (e.g., account access), `Optimizer()` first searches for the corresponding object (i.e., account, slot, and bytecode) in the available cache (O1–O3) or in the state storage (O4). If a hit is found in either the cache or the state, `Optimizer()` then adjusts the gas cost based on optimization rules.

5 Evaluation

We evaluate Maat in the following four research questions.
RQ1: How does Maat mitigate overcharging issues? (§5.1)
RQ2: What are the effects of the optimization rules? (§5.2)
RQ3: What are the overheads of Maat? (§5.3)
RQ4: How scalable is Maat in various blockchains? (§5.4)
Experimental setup. The hardware specs comprise an Intel Xeon Gold 5218R CPU (2.1 GHz, 12 cores), 64 GB RAM, and a 4TB SSD. The software stacks contain go-ethereum v1.12.2, and BSC client v1.2.15. Note that native blockchain clients do not support fine-grained output of gas costs and storage behaviors. Hence, Maat instruments the codebase of these clients to enable JSON-formatted gas cost and storage behaviors, facilitating us in data collection and analysis.

5.1 RQ1: Optimizing Overcharging Issues

To evaluate the efficacy of Maat, we launch experiments on Ethereum from block #18M (Aug-26-2023) to #19M (Jan-13-2024), which contains 1M blocks, 1.58×10^8 transactions, 4.34×10^9 related opcodes, and 5.86×10^9 storage operations.

Table 4 quantifies the optimization result of Maat and its baseline. We set EIP-2929 (see §2) as the baseline of Maat. The 2nd to 6th rows indicate the different data granularity and the 7th row indicates the rate of optimized fee to the total between the Maat and baseline. The 2nd to 4th columns indicate the gas, ether, and charge fee optimized by Maat. We

Table 4: Optimization effect on Ethereum overcharging issues.

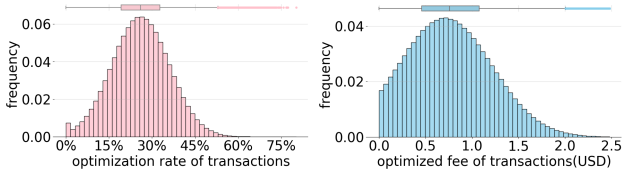
Metrics	Optimized gas (gas)	Optimized ether (ETH)	Optimized fee (USD)
1M blocks (Maat)	2.01×10^{12}	58,358.52	1.12×10^{8}
1M blocks (Baseline)	0.67×10^{12}	21,221.28	0.39×10^{8}
Per block (Maat)	2.01×10^{6}	0.058	112.09
Per block (Baseline)	0.67×10^{6}	0.021	39.02
Per transaction (Maat)	12,730	3.69×10^{-4}	0.71
Per transaction (Baseline)	4,247	1.34×10^{-4}	0.25
Per opcode (Maat)	463.08	1.34×10^{-5}	0.026
Per opcode (Baseline)	155.54	4.93×10^{-5}	0.009
Per operation (Maat)	342.67	9.95×10^{-6}	0.019
Per operation (Baseline)	113.66	3.60×10^{-6}	0.006
Optimized rate (Maat)	33%	32%	32%
Optimized rate (Baseline)	11%	12%	11%

1M blocks represents Ethereum block #18M (Aug-26-2023) to #19M (Jan-13-2024)

extract gas price within each block header to convert each optimized gas to ether and derive the open dataset [23] of price to convert ether to charge of TFM every day.

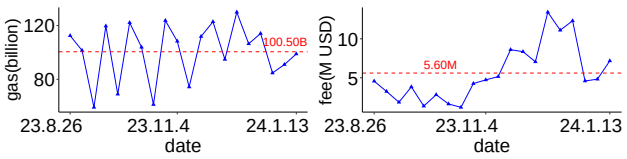
In Table 4, Maat achieves a significant optimization of 2.01 trillion gas (i.e., 58,358.52 ETH and 112M USD) for overcharging issues within 1M blocks of Ethereum, nearly 3× higher than the 0.67 trillion gas (i.e., 21,221.28 ETH and 39M USD) saved by the baseline. When we scrutinize the optimization effects at a finer level, Maat optimizes overcharging issues by 2.01M gas (i.e., 0.058 ETH and 112.09 USD) per block, 12,730 gas (i.e., 3.69×10^{-4} ETH and 0.71 USD) per transaction, 463 gas (i.e., 1.34×10^{-5} ETH and 0.026 USD) per opcode and 342 gas (i.e., 9.95×10^{-6} ETH and 0.019 USD) per storage operation. Regarding the optimization rate, Maat optimizes 32% charge fee, outperforming the baseline's 11%. The variation in optimization rate for gas, ether, and transaction fees stems from the fact that gas and ether prices always fluctuate. The evaluation outcome demonstrates that Maat optimizes 3× more transaction fees than baseline and thereby results in significant fee savings for users.

Fig. 3a displays the frequency distribution of optimization rates for all transactions. The x-axis represents the scope of the optimization rates. The y-axis represents the frequency of a certain optimization rate among all optimization rates. The 25th percentile achieves a 19.25% reduction in transaction fee, while the median (50th percentile) reaches 25.99%. Moreover, the upper quartile (75th percentile) demonstrates savings at 32.74%. Fig. 3b depicts the frequency distribution of optimized fees on each transaction. The x-axis represents the scope of the optimized fees. The y-axis represents the frequency for a specific amount of optimized fees among all optimized fees. The lower quartile yields a fee reduction of



(a) fee optimization rates (b) optimized fee

Figure 3: Gas optimization rates and optimized fees.



(a) gas optimization rate (b) fee optimization rate

Figure 4: Optimization performance weekly.

0.45 USD, the median presents a 0.75 USD reduction, and the upper quartile realizes savings of 1.07 USD. Maat consistently reduces transaction fees, with a median decrease of 0.75 USD and optimizations exceed 25.99% for the median of transactions, underlining the efficacy and performance stability of Maat in optimizing transaction fees for users.

Fig. 4 represents the weekly optimization results of Maat for overcharging issues over 20 weeks. The x-axis and y-axis represent the date and the corresponding gas or fee optimization rate per week, respectively. Figure 4a displays the weekly gas optimization. The maximum gas optimization occurs in the 15th week, with 129.93 billion gas. The minimum gas optimization occurs in the 3rd week, with 59.18 billion gas. On average, 100.5 billion gas is optimized weekly. Fig. 4b presents the weekly fee optimization. It peaks at 13.41M USD in the 15th week, and it hits a minimum of 1.21M USD in the 8th week. On average, 5.6M USD is optimized weekly. The agility of Maat in adapting to various periods is reflected in the overarching trend, maintaining an impressive baseline with an average weekly gas optimization of 100.5B gas and transaction fee of 5.6M USD, indicating significant cost-saving capabilities over an extended period.

Fig. 5 depicts optimization effects for overcharging issues on three types of transactions, i.e., contract invocation, ether transfer, and contract creation. Each type corresponds to three bars, which represent gas of non-storage operations, gas of storage operations, and gas optimized by Maat, respectively. In addition, the percentages at the top of the bars indicate the proportions of different classes of gas to the total amount of gas on the current transaction type. In contract invocation and creation transactions, Maat attains gas reductions of 39.55% and 34.7%. However, Maat achieves a lower optimization rate of 12.43% for ether transfer transactions, because ether transfer involves less storage operations, accounting for only 34.92% of the total gas in these transactions.

Answer to RQ1: *Maat mitigates overcharging issues across all transaction types, optimizing transaction fees by over 32% overall and achieving threefold optimizations than baseline.*

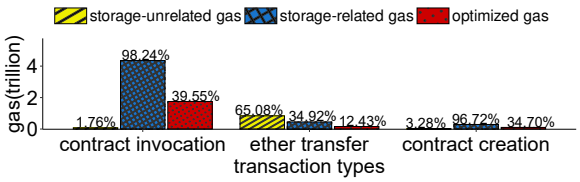


Figure 5: Optimization effects for three types of transactions.

Table 5: The optimization rate of 16 combinations.

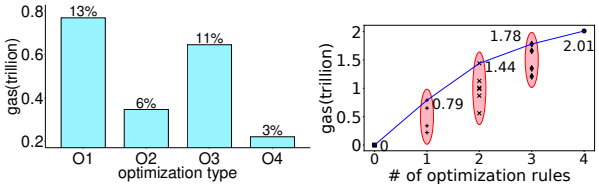
Items	O1	O2	O3	O4	Optimization rate		
					gas	ether	fee
COMB1	✗	✗	✗	✗	0%	0%	0%
COMB2	✓	✗	✗	✗	13%	12%	13%
COMB3	✗	✓	✗	✗	6%	6%	6%
COMB4	✗	✗	✓	✗	11%	11%	10%
COMB5	✗	✗	✗	✓	3%	3%	3%
COMB6	✓	✓	✗	✗	19%	18%	19%
COMB7	✓	✗	✓	✗	24%	23%	23%
COMB8	✓	✗	✗	✓	16%	15%	16%
COMB9	✗	✓	✓	✗	17%	17%	16%
COMB10	✗	✓	✗	✓	9%	9%	9%
COMB11	✗	✗	✓	✓	14%	14%	13%
COMB12	✓	✓	✓	✗	30%	29%	30%
COMB13	✓	✓	✗	✓	22%	21%	22%
COMB14	✓	✗	✓	✓	27%	26%	26%
COMB15	✗	✓	✓	✓	20%	20%	19%
COMB16	✓	✓	✓	✓	33%	32%	32%

5.2 RQ2: Effect of Distinct Optimization Rules

In this section, we conduct experiments to assess the optimized effect of each optimization rule and their combinations.

We compound O1 to O4 to obtain 16 unique combinations (COMB1 to COMB16). Table 5 displays the optimization rates of them. The 2nd to 5th column indicates which optimization rules are included for each combination. The 6th to 8th column represents the optimization rate on gas, ether, and fee. COMB1 serves as the baseline, which signifies the absence of any optimization rule. In contrast, COMB16 turns on all optimization rules. The results reveal a varied landscape of optimization rates. COMB1, where no rules are applied, shows a stark zero percent contrasted with the significant gains observed in subsequent combinations where one or more rules are employed. The full optimization efficiency is achieved by COMB16, where the convergence of all optimization rules, saves 33% reduction in transaction fees.

Fig. 6a compares the optimization effect of each optimization rule. Each bar is annotated with a ratio indicating the gas optimization rate of this rule, which is dividing (i) the optimized gas of the optimization rule by (ii) the total gas cost of all transactions. O1 optimizes the most amount of gas, i.e., 0.79 trillion gas accounting for 13% of the total gas, while O4 provides the least amount of gas optimizations, with 0.22 trillion gas accounting for 3% of the total gas. The reason that O4 optimizes less gas (i.e., 3%) is because O4 is only triggered when deploying contracts, and the gas cost for deploying contracts is only 5% of the total amount of gas.



(a) four optimization rules (b) 16 combinations

Figure 6: Optimized gas of different optimization rules.

Fig. 6b displays the variation of optimized gas for all combinations. The x-axis and y-axis represent the number of rules enabled and the amount of optimized gas, respectively. Distinct points represent different combinations in Table 5. The points on the blue line represent the combinations with the best optimization effect (i.e., optimizing the most amount of gas) among a certain number of rules enabled. Enabling one rule allows for optimizing up to 0.79 trillion gas (i.e., COMB2), while enabling two rules increases this optimization to 1.44 trillion gas (i.e., COMB7). With three rules enabled, up to 1.78 trillion gas can be optimized (i.e., COMB12), and with all four rules enabled, the optimization effect can reach a maximum of 2.01 trillion gas (i.e., COMB16).

Answer to RQ2: *All optimization rules and their combinations exhibit effectiveness in mitigating overcharging issues.*

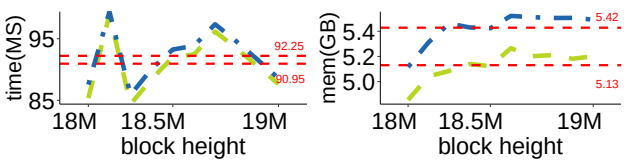
5.3 RQ3: Performance Overhead of Maat

There are two main overheads of Maat, time and space overhead. (i) The time overhead might influence the time of block generation, which could reduce the blockchain's transaction throughput, thus compromising the overall efficiency of the blockchain. (ii) Excessive space overhead demands could make deployment across diverse devices impractical, jeopardizing scalability and widespread adoption of Maat.

Fig. 7 illustrates the results of those evaluations, which involve Maat and geth. (i) Maat: Maat enables all modules including the data collection and optimization. We represent it with the blue line. (ii) geth: A vanilla geth client without any components from Maat. We set it as the baseline, and mark it with the green line. Then, we conduct the evaluations ten times on Ethereum scope from #18M to #19M. Finally, we mark the mean value with the red dashed line and number.

Fig. 7a displays time overhead introduced by Maat, with the x-axis indicating the experimental range, and the y-axis indicating the processing time of each block. We can observe that the maximum time overhead is 2.6% ($\frac{2.19MS}{85.36MS}$) around block #18.1M, the minimum time overhead is 0.4% ($\frac{0.44MS}{98.95MS}$) in block #18.2M, and the average time overhead is 1.4% ($\frac{1.30MS}{90.95MS}$). Hence, the time overhead of 1.4% (roughly 1.3MS) caused by Maat does not impose any detrimental impact on block processing [28, 71]. It indicates that the deployment of Maat on the blockchain is practical because it will not influence the blockchain's throughput and performance.

Fig. 7b depicts the space workload introduced by Maat.



(a) time overhead (b) space overhead

Figure 7: The time and space overhead of Maat.

The x-axis also means the evaluation range and the y-axis represents the memory allocation. We can observe that the maximum memory overhead is 7.4% ($\frac{0.38GiB}{5.08GiB}$) in block height 18.3M, the minimum memory overhead is 5.5% ($\frac{0.27GiB}{4.85GiB}$) in block height 18.1M, and the average memory overhead is 5.6% ($\frac{0.29GiB}{5.13GiB}$). Similarly, the space overhead of Maat remains 5.6% (roughly 0.29GiB), highlighting the feasibility of deploying Maat to address overcharging issues.

Answer to RQ3: *Maat effectively tackles overcharges, showcasing its potential for widespread adoption.*

5.4 RQ4: Scalability of Maat

The scalability of Maat stems from blockchains reusing Ethereum's repository [14, 72]. We first migrate Maat to BSC and measure the optimization effect of it. Then, we examine its scalability by deploying Maat on 50 related blockchains.

We launch Maat within BSC block range from #34.15M (Dec-7-2023) to #35.15M (Jan-11-2024), which contains 1M blocks, 1.68×10^8 transactions, 1.33×10^{10} related opcodes, and 1.73×10^{10} storage operations. We also set EIP-2929 as the baseline. Table 6 delineates the evaluation outcome, showcasing an optimization of 4.56 trillion gas (equivalent to 27,615.54 BNB or 7.71M USD). A detailed examination reveals that Maat attains an optimization rate of 4.56M gas (0.027 BNB or 7.71 USD) per block, 27,069 gas (1.64×10^{-4} BNB or 0.045 USD) per transaction, 341 gas (2.07×10^{-6} BNB and 5.78×10^{-4} USD) per opcode, and 264 gas (1.60×10^{-6} BNB and 4.47×10^{-4} USD) per storage operation. Hence, Maat also outperforms nearly three times than the baseline. In summary, Maat successfully reduces transaction fees by 31% by addressing overcharging issues, demonstrating its potential in mitigating such issues on BSC.

We further select 50 popular blockchains [18] to explore the scalability of Maat, e.g., BSC [8], Ethereum Classic [22], Polygon [27], Optimism [26], and Avalance [17]. First, by utilizing Diffuse [19] (a code comparison tool), we investigate the implementation variance of these blockchains and uncover that they have an identical implementation with `geth` on TFM and storage components. Then, we deploy Maat to those 50 blockchains, without extra code needed to resolve compatibility. We evaluate the deployment of Maat by launching the private chain environments. Concretely, we assess the four scenarios. (i) Multiple transactions within the same block access the same account or storage slot. (ii) Multiple transactions within the same block repeatedly update the same account or storage slot. (iii) Multiple transactions accessing the same account or storage slot in 128 blocks. (iv) Multiple transactions deploy smart contracts with identical bytecodes. For all scenarios, we record the gas costs and storage behaviors before and after applying Maat. The evaluation result confirms that the optimization rules of Maat are effectively and correctly applied across the aforementioned 50 platforms.

Answer to RQ4: *Maat has the capability to extend on other blockchains and address overcharging issues.*

Table 6: Optimization for overcharging issues on BSC.

Metrics	Optimized gas (gas)	Optimized BNB (BNB)	Optimized fee (USD)
1M blocks (Maat)	4.56×10^{12}	27615.54	7.71×10^{6}
1M blocks (Baseline)	1.66×10^{12}	9492.84	2.74×10^{6}
Per block (Maat)	4.56×10^{6}	0.027	7.71
Per block (Baseline)	1.66×10^{6}	0.009	2.74
Per transaction (Maat)	27069.63	1.64×10^{-4}	0.045
Per transaction (Baseline)	8535.21	5.63×10^{-5}	0.016
Per opcode (Maat)	341.70	2.07×10^{-6}	5.78×10^{-4}
Per opcode (Baseline)	124.25	7.11×10^{-7}	2.05×10^{-4}
Per operation (Maat)	264.30	1.60×10^{-6}	4.47×10^{-4}
Per operation (Baseline)	96.11	5.50×10^{-7}	1.59×10^{-4}
Optimized rate (Maat)	33%	32%	31%
Optimized rate (Baseline)	12%	11%	11%

1M blocks represents BSC block #34.15M (Dec-7-2023) to #35.15M (Jan-11-2024)

6 Discussion

Security risks from overcharge issues. The overcharge issues in blockchain systems pose several critical security risks. Specifically, it undermines user trust as they perceive the fee mechanism to be unfair or exploitative, potentially driving users away. Besides, the loss of users can lead to reduced ecosystem participation and transaction volumes, weakening the blockchain network effects and value.

Safety guarantee by Maat. Maat ensures safety guarantees by strictly adhering to blockchain consensus rules during optimization. All optimization rules rely on deterministic data structures, such as blocks, transactions, and state access locations, ensuring consistent behavior across all nodes. To address inconsistencies caused by client diversity or hardware variations, Maat employs a resource pre-allocation technique that predefines cache requirements, ensuring uniform storage operations across heterogeneous environments. The optimized gas fees follow Ethereum specifications, maintaining alignment with existing pricing mechanisms to avoid introducing new vulnerabilities. These measures collectively ensure that Maat's optimizations neither disrupt network consensus nor compromise the integrity of transaction processing.

Impact on incentive mechanism. Maat potentially raises concerns about its economic feasibility, as the optimized transaction fees are the income of miners/validators. However, the reduced revenue will not undermine their participation for three reasons. (i) All the approaches of gas optimization [7, 11, 29] will cause a decrease in the income of miners/validators, but the blockchain continues to thrive. For instance, EIP-2929 causes an 11% reduction (Table 4) and EIP-1559 [7] results in a 33% deduction on the transaction fee's revenue of miners/validators [13]. Although such optimization impairs the short-term profits of miners/validators, it will not affect their participation if the upgrade benefits long-term blockchain development [53]. Besides, lower transaction fees will bring more users flocking to blockchains, increasing interest for miners/validators. (ii) Maat only affects part of the income of the miners/validators (i.e., transaction

fees), miners/validators can still get profit from base block reward [52, 70]. (iii) Several blockchains are known for low transaction fees [17], and they still maintain the participation of miners/validators and draw an extensive user base.

Straightforward design. We proactively adopt rule-based optimizations to balance efficiency and practicality. Integrating Maat into each client necessitates minimizing complexity, as sophisticated algorithms could introduce considerable time overhead and hinder real-time processing. By focusing on lightweight yet effective rules, Maat achieves significant gas fee optimizations with minimal performance impact, ensuring seamless integration across blockchain clients.

Limitations. (i) Client termination can result in the loss of cached data, leading to inconsistent gas costs and compromising the safety guarantees fundamental to Maat. To ensure the cache consistency, Maat integrates a write-ahead log (WAL) [46] for the cache allocated by the resource pre-allocation technique. By implementing WAL, Maat ensures that cache states are persistently recorded, enabling seamless recovery and preserving consistency across clients, even during downtime. Although WAL introduces additional overhead, Maat's total time overhead remains minimal at 1.4%, demonstrating the practicality of WAL.

(ii) Maat does not consider non-blockchain storage components (e.g., OS page cache) as they do not follow consensus. It is non-trivial to ensure the consistency of storage operations on these components on different nodes.

(iii) There are still overcharging issues whose optimization rules do not exist, impacting the comprehensiveness of optimization effects. For example, the collector identified the overcharging issues when blockchain interacts between MPT in cache and disk, as well as the bytecodes in cache and disk. However, deriving rules for optimizing these issues violates the safety principle for designing optimization rules (§4.3), because MPT cache (resp. bytecode cache) differs across different blockchain clients under distinct environments.

O3 Scope. In the context of O3 optimization within Maat, we maintain a buffer for the accessed data from the most recent 128 blocks as it only consumes approximately 230 MiB of memory. There is an inherent tradeoff: augmenting the quantity of stored access data can enhance both the optimization efficacy and blockchain performance; however, this comes at the expense of increased resource utilization. We plan to explore such balance in our future work.

7 Related work

Blockchain storage: Several works [45, 48, 50, 51, 60–62, 69, 76, 77] enhance the efficacy of operations on blockchain storage. For example, Ethanos [45] extracts 5% of highly frequently accessed objects of state to form a separate subtree [63], which speeds up access and alteration operations, boosting the speed of transaction processing by threefold. Verkle tree [48] improves the access speed of the state's object through a flatter structure and optimizes the validation performance through vector commitment approach [44]. LB-Chain [51] adopts sharding technology to optimize blockchain storage by separating it into multiple small disparate shards of blockchain nodes. SlimArchive [37] improves blockchain performance by replacing MPT with a flattened and transaction-level structure to speed up the accessing of accounts and slots.

Previous works focus on single aspect of blockchain storage (e.g., MPT) narrowly, neglecting to account for the multi-faceted nature (e.g., caches and SSAS) of blockchain storage. Maat offers a comprehensive view by assimilating an extensive array of blockchain storage components, yielding a more holistic understanding of blockchain storage landscape.

TFM: Previous studies focus on examining the vulnerabilities originating from gas cost in TFM [30, 38, 58], and enhancing blockchain design to reduce gas cost for executing contracts [49, 68]. For example, iBatch [68] optimizes TFM charges by consolidating multiple contract invocations into a single transaction, resulting in a 19.6% reduction in TFM charges compared to executing these invocations individually. Brokenmetre [58] launches Denial-of-Service attacks against blockchains by exploiting the opcodes whose charges are lower than their actual resource consumption.

Prior works [30,38,58] have confined their analysis of TFM charges solely to the level of smart contracts, lacking a more granular analysis of TFM on blockchain storage. Maat offers richer insights that were previously unexplored, enabling the optimization of TFM in areas that have been overlooked.

8 Conclusion

We identify three overcharge issues in Ethereum, leading to heavy financial burdens for users. To address the overcharging issues, we develop Maat. Maat mitigates these overcharges by employing a combination of fine-grained data collection, consensus-oriented optimizations, and resource pre-allocation techniques. Experimental results demonstrate that Maat effectively optimizes transaction fees on Ethereum by up to 32%, while outperforming three times than baseline, and maintaining a 1% performance overhead. Besides, Maat shows exceptional scalability across 50 other blockchains.

Acknowledgements. We thank reviewers and our shepherd, Alan Liu, as well as Julian Ma and Ansgar Dietrichs from Ethereum Foundation for their advice. This work was supported by National Natural Science Foundation of China (U2336204, 62332004), Hong Kong RGC Projects (PolyU15222320 and Theme-based Research Scheme Project T43-513/23-N), National Key R&D Program of China (2022YFB3103500), Sichuan Science and Technology Program (2024NSFTD0031, 2024YFHZ0339), Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (2023NSFSC1963), and Fundamental Research Funds for Chinese Central Universities (ZYGX2021J018).

References

[1] Besu, java implementation of ethereum. https://github.com/hyperledger/besu.

[2] Erigon, golang implementation of ethereum. https://github.com/ledgerwatch/erigon.

[3] Flat database layout of nethermind. https://github.com/NethermindEth/Paprika.

[4] Geth, golang implementation of ethereum. https://github.com/ethereum/go-ethereum.

[5] Nethermind, c sharp implementation of ethereum. https://github.com/ethereum/go-ethereum.

[6] Plain state of erigon. https://github.com/ledgerwatch/erigon/blob/devel/docs/programmers_guide/db_walkthrough.MD#table-plainstate.

[7] Eip-1559: Fee market change for eth 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559, 2019.

[8] Bnb smart chain (bsc). https://www.bnbchain.org/, 2020.

[9] Snapshot acceleration structure. https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration, 2020.

[10] Block-level warming. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-7863.md, 2021.

[11] Eip-2929: Gas cost increases for state access opcodes. https://eips.ethereum.org/EIPS/eip-2929, 2021.

[12] Eip-4844: Shard blob transactions. https://eips.ethereum.org/EIPS/eip-4844, 2021.

[13] Eip 1559 hasn't affected ethereum miner revenue. https://finance.yahoo.com/news/valid-points-eip-1559-hasn-113000028.html, 2023.

[14] The ethereum ecosystem. https://chainlist.org/, 2023.

[15] Fees are a huge problem. https://coincu.com/75987-vitalik-buterin-admits-fees-are-huge-problem/, 2023.

[16] go-ethereum v.1.12.2. https://github.com/ethereum/go-ethereum/releases/tag/v1.12.2, 2023.

[17] Avalanche. https://www.avax.network/, 2024.

[18] The coinmarketcap: Today's cryptocurrency prices by market cap. https://coinmarketcap.com/, 2024.

[19] Diffuse:graphical tool for merging and comparing text files. ttps://diffuse.sourceforge.net/, 2024.

[20] The document of bonsai tries. https://besu.hyperledger.org/public-networks/concepts/data-storage-formats/, 2024.

[21] The document of solidity language. https://docs.soliditylang.org/, 2024.

[22] Etc. https://ethereumclassic.org/, 2024.

[23] Ether price dataset. https://www.coingecko.com/en/coins/ethereum, 2024.

[24] geth metric. https://geth.ethereum.org/docs/monitoring/metrics, 2024.

[25] The implementation of bsc. https://github.com/bnb-chain/bsc, 2024.

[26] Optimism. https://www.optimism.io/, 2024.

[27] Polygon. https://polygon.technology/, 2024.

[28] Time, slots, and the ordering of events in ethereum proof-of-stake. https://www.paradigm.xyz/2023/04/mev-boost-ethereum-consensus, 2024.

[29] Elvira Albert, Maria Garcia de la Banda, Alejandro Hernández-Cerezo, Alexey Ignatiev, Albert Rubio, and Peter J Stuckey. Superstack: Superoptimization of stack-bytecode via greedy, constraint-based, and sat techniques. *PLDI*, 2024.

[30] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. The opbench ethereum opcode benchmark framework: Design, implementation, validation and experiments. *Performance Evaluation*, 2021.

[31] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. Soda: A generic online detection framework for smart contracts. In *NDSS*, 2020.

[32] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, 2020.

[33] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *ISPEC*, 2017.

[34] Ting Chen, Yuxiao Zhu, Zihao Li, Jiachi Chen, Xiaoqi Li, Xiapu Luo, Xiaodong Lin, and Xiaosong Zhange. Understanding ethereum via graph analysis. In *Infocom*, 2018.

[35] Ethereum community. The archive nodes of ethereum. https://ethereum.org/en/developers/docs/nodes-and-clients/archive-nodes/, 2024.

[36] Ethereum community. The full nodes of ethereum. https://ethereum.org/en/developers/docs/nodes-and-clients/#full-node, 2024.

[37] Hang Feng, Yufeng Hu, Yinghan Kou, Runhuai Li, Jianfeng Zhu, Lei Wu, and Yajin Zhou. {SlimArchive}: A lightweight architecture for ethereum archive nodes. In *USENIX ATC*, 2024.

[38] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. etainter: detecting gas-related vulnerabilities in smart contracts. In *ISSTA*, 2022.

[39] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[40] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *EuroSys*, 2023.

[41] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. Nurgle: Exacerbating resource consumption in blockchain state storage via mpt manipulation. In *SP*, 2024.

[42] Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. Tokenaware: Accurate and efficient bookkeeping recognition for token smart contracts. *TOSEM*, 2023.

[43] Mingyuan Huang, Jiachi Chen, Zigui Jiang, and Zibin Zheng. Revealing hidden threats: An empirical study of library misuse in smart contracts. In *ICSE*, 2024.

[44] Assimakis A Kattis, Konstantin Panarin, and Alexander Vlasov. Redshift: transparent snarks from list polynomial commitments. In *CCS*, 2022.

[45] Jae-Yun Kim, Junmo Lee, Yeonjae Koo, Sanghyeon Park, and Soo-Mook Moon. Ethanos: efficient bootstrapping for full nodes on account-based blockchain. In *EuroSys*, 2021.

[46] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *ACM SIGPLAN Notices*, 2016.

[47] Yeonsoo Kim, Seongho Jeong, Kamil Jezek, Bernd Burgstaller, and Bernhard Scholz. An {Off-The-Chain} execution environment for scalable testing and profiling of smart contracts. In *USENIX ATC*, 2021.

[48] John Kuszmaul. Verkle trees. *Verkle Trees*, 2019.

[49] Arnaud Laurent, Luce Brotcorne, and Bernard Fortz. Transaction fees optimization in the ethereum blockchain. *Blockchain: Research and Applications*, 2022.

[50] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. LVMT: An efficient authenticated storage for blockchain. In *OSDI*, 2023.

[51] Mingzhe Li, Wei Wang, and Jin Zhang. Lb-chain: Load-balanced and low-latency blockchain sharding via account migration. *IEEE TPDS*, 2023.

[52] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Xi Chen, and Ting Chen. Demystifying defi mev activities in flashbots bundle. In *CCS*, 2023.

[53] Yulin Liu, Yuxuan Lu, Kartik Nayak, Fan Zhang, Luyao Zhang, and Yinhong Zhao. Empirical analysis of eip-1559: Transaction fees, waiting times, and consensus security. In *CCS*, 2022.

[54] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. Who goes first? detecting go concurrency bugs via message reordering. In *ASPLOS*, 2022.

[55] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS*, 2016.

[56] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

[57] Rui Pan, Chubo Liu, Guoqing Xiao, Mingxing Duan, Keqin Li, and Kenli Li. An algorithm and architecture co-design for accelerating smart contracts in blockchain. In *ISCA*, 2023.

[58] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in evm. *NDSS*, 2019.

[59] Giuseppe Antonio Pierro and Henrique Rocha. The influence factors on ethereum transaction fees. In *WETSEB*, 2019.

[60] Soujanya Ponnapalli, Aashaka Shah, Souvik Banerjee, Dahlia Malkhi, Amy Tai, Vijay Chidambaram, and Michael Wei. {RainBlock}: Faster transaction processing in public blockchains. In *USENIX ATC*, 2021.

[61] Xiaodong Qi. S-store: A scalable data store towards permissioned blockchain sharding. In *Infocom*, 2022.

[62] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. {mLSM}: Making authenticated storage faster in ethereum. In *HotStorage*, 2018.

[63] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick PC Lee. {ELECT}: Enabling erasure coding tiering for {LSM-tree-based} storage. In *FAST*, 2024.

[64] Javier Ron, Zheyuan He, and Martin Monperrus. Proving and rewarding client diversity to strengthen resilience of blockchain networks. *arXiv preprint arXiv:2411.18401*, 2024.

[65] Javier Ron, César Soto-Valero, Long Zhang, Benoit Baudry, and Martin Monperrus. Highly available blockchain nodes with n-version design. *TDSC*, 2023.

[66] Liuba Shrira and Hao Xu. Thresher: An efficient storage manager for copy-on-write snapshots. In *USENIX ATC*, 2006.

[67] Hongyi Wang, Qingfeng Jing, Rishan Chen, Bingsheng He, Zhengping Qian, and Lidong Zhou. Distributed systems meet economics: pricing in the cloud. In *USENIX HotCloud*, 2010.

[68] Yibo Wang, Qi Zhang, Kai Li, Yuzhe Tang, Jiaqi Chen, Xiapu Luo, and Ting Chen. ibatch: saving ethereum fees via secure and cost-effective batching of smart-contract invocations. In *FSE*, 2021.

[69] Qian Wei, Zehao Chen, Xiaowei Chen, Yuhao Zhang, Xiaojun Cai, Zhiping Jia, Zhaoyan Shen, Yi Wang, Zili Shao, and Bingzhe Li. A semantic-integrated lsm-tree based key-value storage engine for blockchain systems. *TCAD*, 2023.

[70] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[71] Aviv Yaish, Gilad Stern, and Aviv Zohar. Uncle maker:(time) stamping out the competition in ethereum. In *CCS*, 2023.

[72] Xiao Yi, Yuzhou Fang, Daoyuan Wu, and Lingxiao Jiang. Blockscope: Detecting and investigating propagated vulnerabilities in forked blockchain projects. In *NDSS*, 2023.

[73] Cong Yue, Zhongle Xie, Meihui Zhang, Gang Chen, Beng Chin Ooi, Sheng Wang, and Xiaokui Xiao. Analysis of indexing structures for immutable data. In *SIGMOD*, 2020.

[74] Abdullah A Zarir, Gustavo A Oliva, Zhen M Jiang, and Ahmed E Hassan. Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. *TOSEM*, 2021.

[75] Ce Zhang, Cheng Xu, Haibo Hu, and Jianliang Xu. {COLE}: A column-based learned storage for blockchain systems. In *FAST*, 2024.

[76] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *ICDE*, 2019.

[77] Jingyu Zhang, Siqi Zhong, Jin Wang, Xiaofeng Yu, and Osama Alfarraj. A storage optimization scheme for blockchain transaction databases. *Comput. Syst. Sci. Eng.*, 2021.

[78] Jimmy Zheng. Ethereum key financial 2023 overview. https://www.artemis.xyz/research/ethereum-fy-2023-financial-overview, 2023.