Towards Reducing Blockchain Gas Fees via Multi-Query Optimization [Vision]

Duy Cuong Nguyen
Hanoi University of Science and Technology
cuong.nd195845@sis.hust.edu.vn

Vinh Duc Tran
Hanoi University of Science and Technology
ductv@soict.hust.edu.vn

Van Dang Tran
Hanoi University of Science and Technology
dangtv@soict.hust.edu.vn

Quang-Trung Ta
National University of Singapore
taqt@comp.nus.edu.sg

Tien Tuan Anh Dinh
Deakin University
anh.dinh@deakin.edu.au

Robin Doss
Deakin University
robin.doss@deakin.edu.au

ABSTRACT

Smart contracts in Ethereum blockchain are executed by the Ethereum Virtual Machine (EVM), which tracks the contracts' resource consumptions in units of *gas*. Each transaction appearing on the blockchain incurs a transaction fee proportional to the number of gas it consumed. The blockchain imposes a limit on the maximum amount of gas per block. Our goal is to reduce gas consumption per transaction, which can lead to lower transaction fees and more transactions per block. Our vision is to exploit mutli-query optimization — a common technique in database — to realize this goal. This is based on the insight is that many transactions in a block involve the same smart contract, thus there are opportunities for sharing computation and state access.

We confirm that our vision is viable, by collecting data from Ethereum blockchain and analyzing it to estimate the opportunities for execution sharing. We find that accessing data in the ledger accounts for more than 50% of total gas per block, and up to 25% of data access operations involve the same addresses. We evaluate the potential gas saving amount under an ideal block-level cache that stores results of data access operations in the same block. Any operation that can be served directly from the cache is considered *warm access*. We show that this mechanism achieves up to 23% gas reduction on average, most of which comes from popular Decentralized Finance applications.

We discuss four technical challenges in realizing our vision. The first is to design an efficient caching mechanism for EVM. The second is to ensure the correctness, security, and compatibility when implementing it into existing EVM clients. The third is to restructure the existing transaction fee mechanism to fairly distribute the benefits from multi-query optimization. The final challenge is to understand and improve the incentives of the validators in adopting our vision.

PVLDB Reference Format:
Duy Cuong Nguyen, Vinh Duc Tran, Van Dang Tran, Quang-Trung Ta, Tien Tuan Anh Dinh, and Robin Doss. Towards Reducing Blockchain Gas Fees via Multi-Query Optimization [Vision]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code and data is available at https://github.com/dangtv/Ethereum-Transaction-Trace.

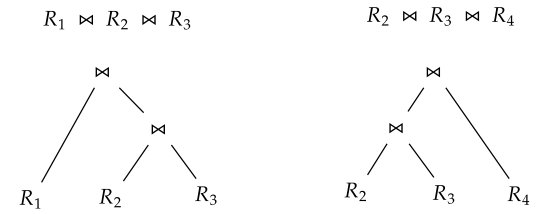
1 INTRODUCTION

Ethereum is the second most valuable blockchain with a market cap of over 400B USD [2]. Its support for smart contracts drives the innovations in decentralized finance applications whose total locked value peaked at 110B USD in 2021 [3]. A smart contract encodes user-defined computation and is executed by the Ethereum Virtual Machine (EVM). Since the execution is replicated over the blockchain nodes (called *validators*), there is a risk of denial of service (DoS) attacks. Ethereum's main defense against DoS is to restrict resource consumption. In particular, EVM tracks resource consumption in units of *gas* and enforces a limit on the number of gas per block. A user sending a transaction to Ethereum must include a gas fee as payment for the transaction [10]. The more gas a transaction consumes, the higher the fee.

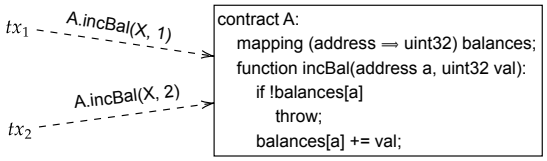
Our goal is to reduce the number of gas consumed per block. Given a list of transactions comprising a block, we aim to execute them with the smallest number of gas. Realizing this goal leads to three immediate benefits to the Ethereum ecosystem. First, the users pay lower transaction fees, given a fixed price per gas unit. Second, reducing the gas consumption of a transaction can prevent the out-of-gas errors that lead to transaction failures. Third, each block can include more transactions, earning higher fees for the validators.

Our key insight is that transactions in the same block have common operators whose execution context and results can be reused. This is based on the observation that many transactions in the block may invoke the same smart contract with the same execution paths. As a result, there are overlapping in computation and data access operations. We propose to apply multi-query optimization — a well established database technique — to extract the benefits of sharing across transactions. Our approach differs from existing works on gas optimization that modify smart contracts to produce gas-efficient codes [1, 5, 19]. These works focus on how much gas can be reduced in one transaction, whereas we focus on how much

gas can be saved (or re-used) across multiple transactions. Our approach also differs from layer-2 (L2) systems such as rollups [9]. L2 reduces gas cost by execute transaction in a separate blockchain and periodically update the states on the original (layer-1) blockchain, whereas we target transactions on layer-1 directly.



(a) Multi-query optimization with relations.



(b) Data sharing across blockchain transactions.

Figure 1: Examples of sharing computation and data access across queries.

Figure 1 shows an example of multi-query optimization in relational databases, and how the technique is extended to Ethereum transaction execution. In Figure 1a two join queries, namely $R_1 \bowtie R_2 \bowtie R_3$ and $R_2 \bowtie R_3 \bowtie R_4$, share a sub-query $R_2 \bowtie R_3$. By materializing this sub-query, one of the query can use the materialized results directly instead of recomputing the sub-query.

In Figure 1b, a smart contract is invoked by two transactions in the same block. Both transactions invoke the same method giving it the same address X. They both read and write to the same storage memory at *balances[X]*. If an EVM executor were able to cache the results of data access operations, the second transaction tx_2 would be able to re-use some results from tx_1. In particular, after tx_1 is executed, the value of *balances[X]*, whether it was updated, would be cached and be re-used for the conditional check in tx_2, thus saving the cost of reading it from the ledger. Furthermore, the write from tx_1 could be buffered and merged with that of tx_2, thereby saving the cost of writing to the ledger.

To understand the opportunities for sharing gas consumption across transactions in the same block, we collect and analyze detailed execution traces of Ethereum transactions. We find that data access accounts for over 50% of total gas per block, and up to 25% of data access operations involve the same storage addresses. We estimate the benefits of multi-query optimization by extending the EVM's data caching mechanism, which works within a single transaction, to support sharing across multiple transactions. In particular, we cache the results of read and write operations of transactions in the same block. A cache hit means the data access operation can be served directly from the cache, and therefore is charged a small amount of gas, for example, that of a *warm access*. A cache miss means the data is not present in the cache or the cached data is invalid, the data access operation is equivalent to a *cold access* and is charged the corresponding amount of gas. Our analysis on Ethereum data shows up to 23% gas reduction on average, most of which come from popular Decentralized Finance applications. We explain the access patterns in real smart contracts that benefit the most from caching.

We then discuss challenges that need to be addressed in order to reap these opportunities. The first is how to build an efficient data-sharing mechanism across transactions in a block. This involves exploring the cache design, and understanding the trade-off between the performance overhead and the amount of gas saved. As the benefit of caching depends on transactions in a block, how to determine if caching is useful for executing that block? The second is to ensure correctness, security, and compatibility when implementing this new caching layer. The next question is how to design a new gas fee structure that accounts for shared data access. In particular, one challenge is to distribute the gas-saving benefits fairly among transactions. The final question is to quantify the incentives for validators to adopt data access sharing, i.e., are they collecting more fees as a result of being able to pack more transactions per block?

In summary, we make the following contributions:

(1) We present a vision of using multi-query optimization to reduce transaction gas consumption in Ethereum, by enabling the sharing of data access operations.
(2) We demonstrate that our vision is practical, by analyzing real Ethereum execution traces and showing up to 35% of storage operations are accessing the same storage address. We estimate that simple caching of read and write operations can reduce up to 50% the amount of gas consumed per block.
(3) We discuss research questions opened up by our vision. These questions concerns the design of the cache, the end-to-end integration to EVM, and the incentives structure.

The rest of the paper is structured as follows. Section 2 provides background and related work on the Ethereum gas mechanism and multi-query optimizations. Section 3 describes our analysis of the cross-transaction sharing opportunities in Ethereum. Section 4 presents the estimates of gas reduction via storage caching, and shows examples of transactions that observe large amounts of saving. Section 5 discusses challenges before Section 6 concludes.

2 BACKGROUND

2.1 Ethereum Virtual Machine

Smart contracts are compiled into bytecodes and executed by EVM clients. The EVM is stack-based, and it charges a certain amount of *gas* for each opcode executed [18]. An Ethereum transaction specifies the amount of gas it is willing to pay, and the EVM clients keep track of the number of remaining gas when it executes the transaction. When the transaction runs out of gas, it is aborted. Table 1 shows a trace of a transaction execution, including the states being tracked by the EVM. In particular, *PC* refers to the position in the bytecode, *Operation* to the opcode being executed, *Gas* to the remaining gas avaialbe for the transaction, and *GasCost* to the gas cost of the opcode. The other states include *Stack*, which is the content of the current stack, *Memory* which is values in memory, and *Depth* which is the depth of the call stack.

Step	PC	Operation	Gas	GasCost	Stack	Memory	Depth
[1]	0	PUSH1	56321	3	[]	[]	1
[2]	2	PUSH1	56318	3	$[0x80]$	[]	1
[3]	4	MSTORE	58315	12	$[0x80, 0x40]$	[]	1
[4]	5	CALLVALUE	58303	2	[]	[0....0080]	1
...	...	...	...	...	...	...	...

Table 1: An example of the transaction execution trace.

Storage Keys. Each smart contract is given a storage memory space. Any update to the contract storage is made persistent in the blockchain. The storage is essentially a large array of 32-byte slots. Each contract state is stored at an index of array [11, 18]. We refer to this index as *storage key*. A combination of contract address and storage key uniquely identifies the state. We refer to this combination as *storage address*. During the execution of a read or write operation, EVM uses the storage key pushed on the stack to identify the corresponding state.

2.2 Gas Mechanism

Gas Fee. Each transaction in Ethereum consumes an amount of gas units, which is the sum of the gas costs of all the executed opcodes. Ethereum enforces a hard limit of 30 million gas units per block. The transaction owner pays a *gas fee* f to the validators. In the original version of Ethereum, the gas fee is the product of the number of gas units g and a gas price p, i.e. $f = g \times p$. The latter is specified by the transaction owner, which creates incentives for the validators to include transactions with the highest fees in the block.

The latest version of Ethereum implements a major change in gas fee proposed in EIP-1559 [10, 14]. Specifically, the gas price p includes a base price b that captures network condition, and a priority price $prio$. The base fee $b \times g$ is burnt, and the remaining $b \times prio$ is paid to the validators. Our goal of reducing g per transaction is orthogonal to the gas price mechanism because lower g directly leads to lower gas fees given any fixed gas price.

Storage Operation Cost. When executing a transaction, the first read or write access to a storage address is considered a *cold access* and is charged a large amount of gas (e.g., 21000 gas for write). Subsequent read or write to the same address is considered a warm access which is charged a smaller amount of gas (e.g., 100 gas for warm write). EIP-2930 [16] proposes an *access list* parameter containing a list of addresses and storage keys that a transaction plans to access. When the EVM executes the transaction, any access to a storage address in the list is considered warm access. Empirical analysis of EIP-2930 demonstrates that access lists are not widely used, and that they can lead to sub-optimal gas consumption [6].

2.3 Multi-Query Optimization

Multi-query optimization was used in relational databases to reduce the execution time of a batch of queries. Early works focus on join queries and propose algorithms that quickly find shared components of the query plans, whose results are materialized and re-used by multiple queries [8, 12, 13]. Figure 1a shows an example of two query plans with $(R_2 \bowtie R_3)$. More recent works focus on batch processing systems such as MapReduce, where multi-query optimization are in the form of sharing of I/O scanning and intermediate outputs such as those of map functions [7, 17].

Multi-query optimization (MQO) assumes that queries are processed in batch, which makes it unsuitable for applications requiring low latency such as traditional OLTP applications. Ethereum executes transactions in batches, making it a nature fit for MQO. Realizing the full potentials of MQO requires careful design choices. In the example in Figure 1a, storing and re-using $R_2 \bowtie R_3$ can be more costly than recomputing it when the results are large or I/O contention is high. In our settings, using MQO to reduce transaction gas may lead to the validators earning lower fees from executing the block, which gives the validators no incentives to adopt MQO.

3 CROSS-TRANSACTION SHARING OPPORTUNITIES

3.1 Dataset

We build a dataset containing detailed execution traces of Ethereum transactions. We use the Go-Ethereum (Geth) client [4] to download 3000 blocks, from four distinct periods, covering blocks from number 19519860 to number 20050000. We then use Geth to replay them ($0.7M$ transactions) and call the `bug_traceTransaction` function to collect execution traces. Table 1 shows an example of the traces. The compressed trace is of 165GB in size.

We release the dataset and analysis code at https://github.com/dangtv/Ethereum-Transaction-Trace.

3.2 Gas Cost Analysis

We first analyze how frequently different opcodes are used in real transactions, and how much they account for the transaction gas consumption. Figure 2[a,b] shows the distributions. The most popular opcodes are stack-related, which reflects the fact that EVM is a stack-based machine. Although storage operations, namely SLOAD and SSTORE, have a low frequency of 0.7% of all the opcodes, they account for over 70% of the total transaction gas. This finding is aligned with EIP-2929 [15] that have different costs for warm and for cold accesses.

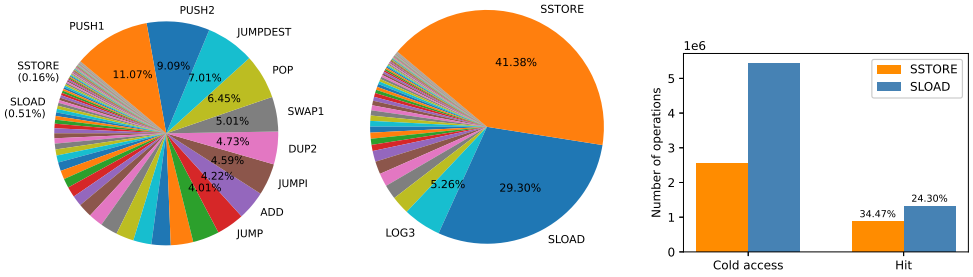
We next examine the storage operations across transactions in the same block. Figure 2[c] shows how many operations access the *same* storage address. In particular, 25% of reads and 34% of writes are to the same address. This finding suggests the opportunities of sharing storage accesses across transactions.

4 GAS SAVING VIA CACHING

Motivated by the opportunities of sharing storage access, we analyze the collected traces to estimate the benefits of caching storage access. We consider a per-block cache for storing the results of SLOAD and SSTORE. It works in the same way as current caching mechanisms in Ethereum, i.e. EIP-2930, except that the cache content persists for the duration of the block, as opposed to the duration of a transaction.

4.1 Estimating Gas Cost

Given the execution trace of a block, we compute transaction gas using a block-level cache by identifying the cold and warm accesses.



(a) Frequency **(b) Gas cost** **(c) Hit rate**

Figure 2: Gas cost analysis, and the opportunities for storage sharing.

We extract all SLOAD and SSTORE opcodes from the trace, and group them by storage address, i.e., by the contract address and storage key. Each storage address then consists of a list of operations sorted by their indices in the original trace.

Transaction	Contract	StorageKey	Opcode	StorageValue	GasCost	NewCost
19519861-0	0x...c78ba3	0xc0c2a4..	SLOAD	0x10662a8..	2100	2100
19519861-0	0x...c78ba3	0xc0c2a4..	SSTORE	0x1	100	100
19519861-0	0x...c78ba3	0xc0c2a4..	SLOAD	0x1	100	100
19519861-6	0x...c78ba3	0xc0c2a4..	SLOAD	0x1	2100	100
19519861-8	0x...c78ba3	0xc0c2a4..	SLOAD	0x1	2100	100

Table 2: Example of estimated gas cost for SLOAD.

Transaction	Contract	StorageKey	Opcode	StorageValue	GasCost	NewCost
19519861-1	0x...c78ba3	0xc0c2a4..	SLOAD	0x0	2100	2100
19519861-1	0x...c78ba3	0xc0c2a4..	SSTORE	0x106a8..	22100	100
19519861-1	0x...c78ba3	0xc0c2a4..	SLOAD	0x106a8..	100	100
19519861-1	0x...c78ba3	0xc0c2a4..	SLOAD	0x106a8..	100	100
19519861-1	0x...c78ba3	0xc0c2a4..	SSTORE	0x0	100	100

Table 3: Example of estimated gas cost for SSTORE.

SLOAD. If the first operation of a given storage address is a SLOAD, then the first SLOAD is considered cold access (2100 gas units), and all subsequent ones are warm accesses (100 gas units). If the first operation is an SSTORE, the remaining SLOADs of the same addresses are warm accesses. Table 2 shows an example where two SLOADs become warm because of the SLOAD of an early transaction in the block.

SSTORE. We exploit the fact that EVM only commit changes to the ledger after all transactions in the block have been executed, therefore writes to the same address can be batched and only the latest value need to be written to the ledger. As a result, when there are multiple SSTOREs that accessing the same storage address in a block, at most one of them need to be a cold access. There are even cases when no cold accesses are necessary, as illustrated in Table 3. In this example, the final SSTORE updates the storage address with the same value as at the beginning of the block, meaning that the address does not change and no update to the ledger is needed.

4.2 Analysis

We evaluate the amount of gas saved over all the storage operations, and over the total gas consumption. We also examine if different operations lead to different saving.

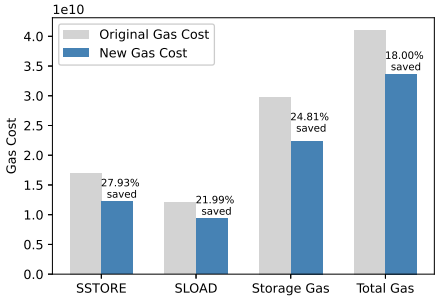


Figure 3: Gas saving analysis.

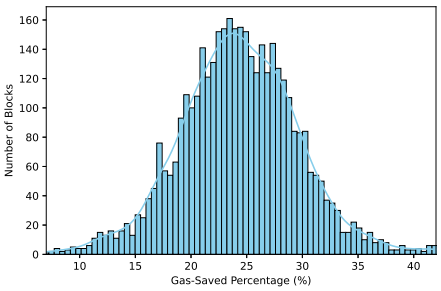


Figure 4: Distribution of per-block gas saving.

Figure 3 shows up to 24% gas reduction across all storage operations, and average of 18% reduction over the original cost. The results confirms the benefit of caching. Figure 3 also compares the saving of SLOAD versus that of SSTORE. Both the saving percentage and the number of gas unit saved by SSTORE are higher than those of SLOAD. This is because of the hit rate shown in Figure 2[c], and because turning a cold SSTORE to a warm one saves 20900 units of gas, whereas turning a cold SLOAD to a warm one saves only 2000 units.

Figure 4 shows the distribution of the number of gas units saved per block. The peak is at 25%, and we observe that some blocks save up to 50%. We discusses these cases in the next section.

Contract	SSTORE			SLOAD			Original Gas	Saved Gas	Saved Percentage
	Hit rate	Cold	Warm	Hit rate	Cold	Warm			
0x...99e14	100	4	4	50	4	12	90000	83600	92.88
0x...4a320	0	0	0	92	744	0	15622400	1414000	90.50
0x...9b347	100	3	3	0	3	0	66600	59700	89.63
0x...4693f	100	2	2	0	2	0	44400	39800	89.63
0x...e2906	100	46	46	4.17	48	48	1030200	919400	89.24

Table 4: Top 5 Contracts with Highest Percentage of Gas Saved

4.3 Smart Contract Analysis

We analyze contracts that benefit the most from caching, in terms of the percentage and number of gas units saved.

4.3.1 Saving percentage. Table 4 lists the contracts with the highest saving percentage. These contracts are not popular and do not come with a source code. Therefore, we only list their deployed addresses. Some contracts have over 90% saving, suggesting that they contain repetitive patterns which are highly amenable to caching.

Contract 0x...99e14 - 100% SSTORE hit rate This contract has two SSTOREs appearing after a SLOAD, and the second SSTORE restores the original value. We reverse-engineer the bytecode and find that the contract updates a variable to some value and then reset back to 0. Our caching strategy considers these SSTORE as warm access, thus the entire contract execution has no cold SSTORE.

Contract 0x...4a320 - 92% SLOAD hit rate. This contract has a single state variable with a static storage key (0x4). Since SLOAD operations are cached across transactions within a block, more transaction calls to this contract in the same block result in higher cache hits, leading to significant gas savings.

4.3.2 Saving amount. Table 5 lists the top contracts that have the most saving in terms of the number of gas units. These contracts are popular, well-known decentralized finance applications. In particular, we find contracts such as Wrapped Ether and *Uniswap: Universal Router* in this list, with saving of upto 81.87%. We observe high hit rates for both SSTORE and SLOAD in the Universal Router contract (91.41% and 92.48%), suggesting that the contract contains access patterns amenable to caching.

4.3.3 Wrapped Ether (WETH). This contract turns native Ethereum currency (Ether) into ERC-20 tokens. It has a market cap of over $\$10B$ as of November 2024. We observe 63.24% gas saving with this contract. To explain this significant saving, we outline below its main data structures and its most frequently invoked functions.

```
uint256[] array_0; // STORAGE[0x0]
uint256[] array_1; // STORAGE[0x1]
mapping (uint256 => uint256) _balanceOf; // STORAGE[0x3]
mapping (uint256 => mapping (uint256 => uint256)) _allowance; // [0x4]
uint8 _decimals; // STORAGE[0x2] bytes 0 to 0

function withdraw(uint wad) public {
    require(balanceOf[msg.sender] >= wad);
    balanceOf[msg.sender] -= wad;
    msg.sender.transfer(wad);
    Withdrawal(msg.sender, wad);
}

function transfer(address dst, uint wad) public returns (bool) {
    return transferFrom(msg.sender, dst, wad);
}
```

```
function transferFrom(address src, address dst, uint wad) {...} {
    require(balanceOf[src] >= wad);
    if (src != msg.sender && allowance[src][msg.sender] != uint(-1)) {
        require(allowance[src][msg.sender] >= wad);
        allowance[src][msg.sender] -= wad;
    }
    balanceOf[src] -= wad;
    balanceOf[dst] += wad;
    Transfer(src, dst, wad);
    return true;
}
```

This contract can be called from many other contracts. Figure 5 shows the function call trace of one transaction invoking WETH multiple times. On line 7, the Uniswap V2 contract transfers an amount `wad = 31,286,...` of WETH to the MEV Bot contract at address `0xd129...ecd`. This transfer results in an update of the MEV Bot's WETH balance: `balanceOf[MEV Bot] += wad`. Immediately after that, on line 13, the `withdraw` function is invoked with `msg.sender = MEV Bot`, using the same `wad` value to convert the MEV Bot's WETH into ETH, that is: `balanceOf[MEV Bot] -= wad`. During this process, the WETH balance of the MEV Bot is essentially restored to its original value. This pattern of temporarily increasing and then restoring the balance is common in Ethereum transactions, especially during conversions between WETH and ETH. This is because conversion to and from WETH is necessary in applications or contracts that only supports ERC-20 tokens as inputs. One possible explanation for the trace in Figure 5 is that a user wants to exchange an amount of USDT for Ether, where the former is a ERC-20 token and the latter is not. The transaction invokes Uniswap, which returns an amount of WETH to the user's account, which user then converts immediately back to Ether.

We note that these repeated state changes, in which the same storage address is updated multiple times in a block, present an opportunity to save on cold SSTORE access. Our caching mechanism (Section 4.1) goes even further, i.e. it removes all cold SSTORE accesses, when multiple updates restore the original values.

5 OPEN RESEARCH QUESTIONS

The results presented in the previous sections confirm the viability of our vision. In particular, they demonstrate that a simple caching strategy can significantly reduce the gas consumption of the entire block, leading to potentially large savings for users. In the next steps, we plan to realize our vision by implementing this proposal and integrating it into existing EVM clients. We discuss below four research and technical challenges that need to be overcome before our vision becomes reality.

The first challenge is how to implement an efficient and robust caching mechanism for data access operations that can be integrated into existing EVM clients. This requires a detailed analysis

Contract	SSTORE			SLOAD			Original Gas	Saved Percentage
	Hit rate	Cold	Warm	Hit rate	Cold	Warm		
Wrapped Ether	35	327196	154409	41.71	302497	725888	3.47×10^9	63.24
Stable Coin: USDT	3.95	163805	19635	55.69	527235	165786	2.25×10^9	30.12
USDC Token	8.17	84748	16207	51.80	177758	387738	9.98×10^8	27.88
Uniswap: Universal Router	91.41	53706	50394	92.48	52386	127078	2.86×10^8	81.87
MAGA Token	11.20	19520	4617	53.13	72144	115988	3.06×10^8	37.75

Table 5: Top 5 Contracts with Highest Saving Amount.

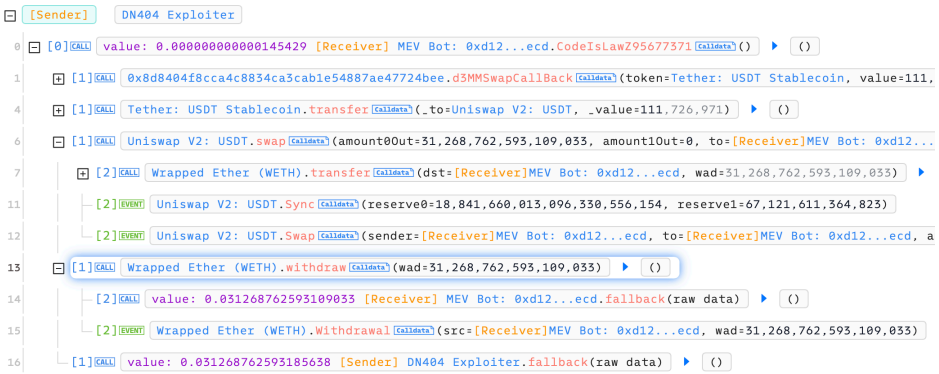


Figure 5: An example transaction of WETH to ETH Conversion (https://tinyurl.com/blockbeef)

StorageKey	Opcode	Transaction	GasCost	StorageValue	NewCost
0x4	SLOAD	19720551-69	2100	0x1	2100
0x4	SLOAD	19720586-70	2100	0x1	2100
0x4	SLOAD	19955947-50	2100	0x1	2100
0x4	SLOAD	19955947-51	2100	0x1	100
0x4	SLOAD	19955947-52	2100	0x1	100
0x4	SLOAD	19955947-53	2100	0x1	100

Table 6: SLOAD pattern in contract $0x...4a320$

of the EVM architecture and the implementation of the caching layer to ensure that it both can run fast and save as much gas as possible while incurring low overhead. Since caching consumes resources, an interesting question is how to predetermine if there are benefits in caching, before actually executing the block. There are several potential solutions to address this question. For example, one could perform static analysis on the smart contract source code and bytecode to identify the shared data access patterns. The other approach is to dynamically execute the transactions in a sandbox environment to identify the commonly accessed storage addresses. More advanced methods can involve combining static analysis, dynamic execution, and other methods in program analysis, compiler optimization, and database query optimization to identify the caching opportunities for each block.

The second challenge is to ensure correctness, security, and compatibility when implementing this caching layer. More specifically, the caching algorithm must be correct, that is, it must produce the same results as the original EVM clients when executing transactions without caching. A potential issue is that transactions can be aborted, thus any uncommitted updates must not be shared. The implementation must also be secure, ensuring that it does not introduce any vulnerabilities or security risks. This is important given the tremendous amount of money being locked on Ethereum. Furthermore, the implementation needs to be compatible with the existing EVM clients, so that it can be integrated into existing EVM-based networks.

The third challenge is to design a new gas fee structure that accounts for shared access. In the new execution model, the early transactions in the blocks incur cold accesses, whereas the latter can enjoy warm accesses. Therefore, both the cost and benefits of caching must be distributed fairly among the transactions. One potential solution is to charge all the transactions the average cost of all the warm and cold accesses. The transaction still includes a payment for gas, but any gas surplus due to sharing is returned back to the transaction owner.

The final challenge is to quantify the benefits of caching for validators. While sharing storage access can potentially save gas for users, it introduces overhead for validators to maintain the cache. Furthermore, since it can reduce the gas usage of each transaction, the fee earned by the validators can be lower. Although the validators can pack more transactions per block, thereby earning more fees on average, the actual benefits need to be validated. We envision that initial validation can be based on game theoretic models.

6 CONCLUSION

We presented a vision of exploiting multi-query optimization for reducing Ethereum gas consumption. The key idea is to re-use results of data access operations across transactions in the same block. We estimated the benefit of sharing via simple cross-transaction caching mechanism. We analyzed Ethereum execution trace and found that the caching technique is effective. We observed up to 20% gas reduction on average, and many popular decentralized finance applications enjoyed significant savings.

REFERENCES

[1] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiaopu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2020. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2020), 1433–1448.

[2] CoinMarketCap. 2024. CoinMarketCap: Cryptocurrency Prices, Charts And Market Capitalizations. https://coinmarketcap.com/.

[3] DefiLlama. 2024. DefiLlama - DeFi Dashboard, TVL Aggregator. https://defillama.com/.

[4] The go-ethereum Authors. 2024. Go-Ethereum: Official Go implementation of the Ethereum protocol. https://geth.ethereum.org.

[5] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[6] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, Roger Wattenhofer, and Patrick Züst. 2023. Dissecting the eip-2930 optional access lists. *arXiv preprint arXiv:2312.06574* (2023).

[7] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[8] Tarun Kathuria and S Sudarshan. 2017. Efficient and provable multi-query optimization. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 53–67.

[9] L2BEAT. 2024. The state of layer 2 ecosystem. https://l2beat.com/scaling/summary.

[10] Yulin Liu, Yuxuan Lu, Kartik Nayak, Fan Zhang, Luyao Zhang, and Yinhong Zhao. 2022. Empirical Analysis of EIP-1559: Transaction Fees, Waiting Times, and Consensus Security. In *CCS*.

[11] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2024. Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. In *Netw. Distrib. Syst. Security Symp.*

[12] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.

[13] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. 2001. Cache-on-demand: Recycling with certainty. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 633–640.

[14] Vitalik Buterin and Eric Conner and Rick Dudley and Matthew Slipper and Ian Norden and Abdelhamid Bakhta. [n.d.]. EIP-1559: Fee market change for ETH 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559. *Ethereum Improvement Proposals* ([n. d.]).

[15] Vitalik Buterin and Martin Swende. [n.d.]. EIP-2929: Gas cost increases for state access opcodes. https://eips.ethereum.org/EIPS/eip-2929. *Ethereum Improvement Proposals* ([n. d.]).

[16] Vitalik Buterin and Martin Swende. [n.d.]. EIP-2930: Optional access lists. https://eips.ethereum.org/EIPS/eip-2930. *Ethereum Improvement Proposals* ([n. d.]).

[17] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *Proceedings of the VLDB Endowment* 7, 3 (2013), 145–156.

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

[19] Abdullah A Zarir, Gustavo A Oliva, Zhen M Jiang, and Ahmed E Hassan. 2021. Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–38.