Storage Strategies for Gas Reduction

jistro.eth 0xVato.stark

Last update: January 13, 2024

ABSTRACT

The objective of this research is to efficiently store any number of native or non-native tokens for an account while minimizing the gas consumption for each transaction. This goal is pursued without accounting for the deployment cost of the contract.

Keywords: EVM, Blockchain

CONTENTS

1 First Iteration: Implementation of Mappings 1
2 Second Iteration: Implementation of Modifiers 2
3 Third Iteration: Reduction of Nested Mappings 2
4 Fourth Iteration: Implementation of `payable` in Functions 3
4.1 Concerns about `payable` in Functions . . . 3
5 Fifth Iteration: Implementation of Linked Lists 3
6 Conclusion 3
References 5

1 FIRST ITERATION: IMPLEMENTATION OF MAPPINGS

During this phase, a smart contract is implemented employing a specific strategy. The use of a `mapping` is more gas-efficient than implementing with dimensional or multi-dimensional arrays [1]. The implementation utilizes a `public` nested `mapping`, where the first key (`address`) represents the account, and the subsequent key (`address`) corresponds to the smart contract of the token. This structure allows us to retrieve the holdings for a specific account in this token, represented by `uint256`.

> It's crucial to note that, for each smart contract implemented, the compiler version used is 0.8.20, utilizing the Shanghai EVM version. All tests are conducted using an account with 10,000 registered addresses. This choice is based on the observation of a linear pattern in gas consumption when testing with 100, 1000, and 10,000 mock token addresses. As there were no drastic changes in the pattern, we have chosen to keep the number of registered addresses at 10,000 for consistency.

Based on the code from `https://github.com/Roll-a-Mate/Research/blob/main/0004-Storage%20Strategies%20for%20Gas%20Reduction/code/src/FirstApproach.sol`, using the `entryInit`, `entryAdd`, and `entrySub` functions, the gas consumption results are shown in Table 1.

Name	min	avg	median	max
entryInit	22752	22752	22752	22752
entryAdd	945	945	945	945
entrySub	923	923	923	923
mean				8207

Table 1. Gas Consumption Results for the First Iteration

This serves as the foundation for subsequent iterations in the implementation of new strategies, eliminating the need for comparisons.

2 SECOND ITERATION: IMPLEMENTATION OF MODIFIERS

In this iteration, we introduce different modifiers to optimize transaction costs. For example, minimizing the use of `public` variables can significantly reduce gas expenses. `Public` variables implicitly generate a getter function, contributing to increased contract size and gas usage. To enhance efficiency, functions should be marked as `external` whenever possible. `External` functions are more gas-efficient than public ones, as they expect arguments to be passed from the external call, ultimately saving gas [2].
Based on the code from `https://github.com/Roll-a-Mate/Research/blob/main/0004-Storage%20Strategies%20for%20Gas%20Reduction/code/src/SecondApproach.sol`, using the `private` keyword for variables and `external` for functions, the gas consumption results are shown in Table 2.

Name	min	avg	median	max
entryInit	22730	22730	22730	22730
entryAdd	923	923	923	923
entrySub	901	901	901	901
mean				8185

Table 2. Gas Consumption Results for the Second Iteration

Based on the mean total cost of the transaction, a change of -0.27% (22 gas) is observed.

3 THIRD ITERATION: REDUCTION OF NESTED MAPPINGS

In this iteration, we aim to reduce the cost associated with using nested mappings by implementing a theoretical reduction in nested storage. To achieve this, we introduce a `bytes32` key to retrieve the balance of a specific token. This is accomplished by hashing the `address` of the account with the `address` of the token, the use of `bytes32` as the key is because this type of variables is the most optimized storage type [2, 3].
Based on the code from `https://github.com/Roll-a-Mate/Research/blob/main/0004-Storage%20Strategies%20for%20Gas%20Reduction/code/src/ThirdApproach.sol`, using the `sha256` keyword for hashing, the gas consumption results are shown in Table 3.

Name	min	avg	median	max
entryInit	23455	23455	23455	23455
entryAdd	1621	1621	1621	1621
entrySub	1599	1599	1599	1599
mean				8891

Table 3. Gas Consumption Results for the Third Iteration

Compared to the first iteration, this iteration shows a 8.33% increase in gas consumption (684 gas units) in the mean total cost. This indicates a more expensive implementation, leading us to discard this approach.

4 FOURTH ITERATION: IMPLEMENTATION OF `PAYABLE` IN FUNCTIONS

In this iteration, we explore the introduction of the `payable` modifier in every function, aiming to achieve a slight improvement in gas efficiency compared to non-payable ones. The advantage lies in the fact that the compiler doesn't need to check for the transfer of Ether in payable functions, contributing to potential gas savings [2]. Based on the code from https://github.com/Roll-a-Mate/Research/blob/main/0004-Storage%20Strategies%20for%20Gas%20Reduction/code/src/FourthApproach.sol, the gas consumption results are shown in Table 4.

Name	min	avg	median	max
entryInit	22706	22706	22706	22706
entryAdd	899	899	899	899
entrySub	877	877	877	877
mean				8161

Table 4. Gas Consumption Results for the Fourth Iteration

Compared to the first iteration, this iteration shows a decrease of 0.56% (46 gas units).

4.1 Concerns about `payable` in Functions

The application of the `payable` modifier in functions raises substantial security concerns. Enabling primary token transfers within a function introduces potential risks, particularly when the specific actions or possibilities within this transaction are not fully comprehended. A meticulous evaluation of the security implications is imperative before integrating the `payable` modifier into any function. In this section, we will explore the criticality of these concerns.

Upon consulting the Slither Detector wiki [4] and reviewing pertinent security articles [5, 6], along with security audit documents [7, 8, 9], it becomes evident that issues associated with the `payable` modifier primarily revolve around the transfer of native tokens into the contract, such as reentrancy or replay attacks using loops, etc. Notably, there is no documented evidence regarding specific payload executions.

5 FIFTH ITERATION: IMPLEMENTATION OF LINKED LISTS

In this iteration, we explore the use of linked lists to retrieve token data for each transaction, aiming to achieve more gas-efficient transactions.

Based on the code from SoliChain [10], we implement this approach in our code available at https://github.com/Roll-a-Mate/Research/blob/main/0004-Storage%20Strategies%20for%20Gas%20Reduction/code/src/FifthApproach.sol. The gas consumption results are shown in Table 5.

Name	min	avg	median	max
entryInit	25369	45267	45269	45269
entryAdd	1045	1045	1045	1045
entrySub	1067	1067	1067	1067
mean				15794

Table 5. Gas Consumption Results for the Fifth Iteration

Compared to the first iteration, this approach results in a 92.78% increase in gas consumption (7587 gas units) in the mean total cost. This indicates a very expensive implementation, leading us to discard this approach.

6 CONCLUSION

The investigation into optimizing gas consumption for transactions on Ethereum or compatible EVM chains has led to several iterations of smart contract implementations. Each iteration introduced specific strategies, and the gas consumption results were analyzed. Here's a summary of the conclusions drawn from each iteration:

First Iteration: Implementation of Mappings

Utilizing a `mapping` for storage proves to be more gas-efficient than using dimensional or multi-dimensional arrays.

The use of a `public` nested `mapping` provides a foundation for subsequent iterations.

Second Iteration: Implementation of Modifiers

Introduction of different modifiers, such as marking variables as `private` and functions as `external`, results in a decrease in gas consumption (0.27%).

Third Iteration: Reduction of Nested Mappings

Attempting to reduce the cost of nested mappings by introducing a `bytes32` key increases gas consumption by 8.33%. This approach is deemed more expensive and is discarded.

Fourth Iteration: Implementation of `payable` in Functions

Introducing the `payable` modifier in every function results in a small decrease in gas consumption (0.56%).

Fifth Iteration: Implementation of Linked Lists

Implementing linked lists to retrieve token data significantly increases gas consumption (92.78%), rendering this approach too expensive and unsuitable for optimization. Overall, while some strategies led to marginal improvements, others proved to be more expensive. The choice of gas optimization strategy should be carefully considered based on the specific requirements and trade-offs in the context of the smart contract application.

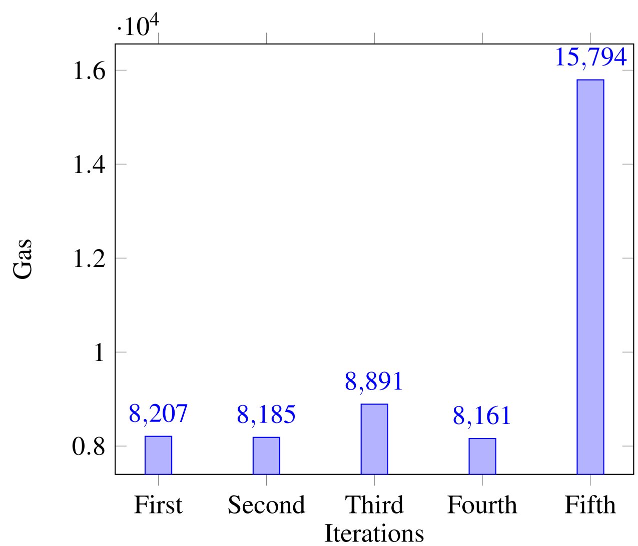

Figure 1. Graph comparing Gas Consumption Results for each Iteration

REFERENCES

[1] natewelch_, "Answer to "Array or mapping, which costs more gas?"," Jan. 2018. [Online]. Available: https://ethereum.stackexchange.com/a/37594

[2] R. T. Malanii, Oleh, "Solidity Gas Optimization : Best Practices & Expert Tips," Nov. 2023. [Online]. Available: https://hacken.io/discover/solidity-gas-optimization/

[3] Y. Riady, "How to Write Gas Efficient Contracts in Solidity," May 2021. [Online]. Available: https://yos.io/2021/05/17/gas-efficient-solidity/

[4] Trail of Bits, "Detector Documentation." [Online]. Available: https://github.com/crytic/slither/wiki/Detector-Documentation

[5] "Smart Contract Security: The Ultimate Guide," May 2023. [Online]. Available: https://www.rareskills.io/post/smart-contract-security

[6] "Recommendations for Smart Contract Security in Solidity - Ethereum Smart Contract Best Practices." [Online]. Available: https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/recommendations/

[7] WatchPug, "[H-04] EIP712MetaTransaction.executeMetaTransaction() failed txs are open to replay attacks." [Online]. Available: https://solodit.xyz/issues/h-04-eip712metatransactionexecutemetatransaction-failed-txs-are-open-to-replay-attacks-code4rena-rolla-rolla-contest-git

[8] berndartmueller, Dravee, antonttc, cccz, and WatchPug, "[M-01] ‘call()‘ should be used instead of ‘transfer()‘ on an ‘address payable‘." [Online]. Available: https://solodit.xyz/issues/m-01-call-should-be-used-instead-of-transfer-on-an-address-payable-code4rena-backd-backd-contest-git

[9] N. Chin and M. Krüger, "Use of delegatecall in a payable function inside a loop." [Online]. Available: https://solodit.xyz/issues/use-of-delegatecall-in-a-payable-function-inside-a-loop-trailofbits-yield-v2-pdf

[10] A. Boudjemaa, "Elevate Your Smart Contracts with Linked Lists," Nov. 2023. [Online]. Available: https://blog.solichain.com/elevate-your-smart-contracts-with-linked-lists-8ff8ab2c197a