Saving gas with a packaging framework for Ethereum transactions

Bruno Medeiros de Oliveira[1], **Pedro Antonino**[2], **Augusto Sampaio**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE) – Recife – Brazil

[2]The Blockhouse Technology Limited – Oxford, UK

bmo@cin.ufpe.br, pedro@tbtl.com, acas@cin.ufpe.br

Abstract. *Transaction costs are a barrier to the wide adoption of decentralized applications. To address this issue, this paper proposes a framework to optimize transaction fees by carrying out a conflict-avoiding packaging of multiple smart-contract calls into a single transaction. It relies on two main concepts: a way to optimize the bundling of transactions with a* conflict-avoiding packaging strategy *and a way to execute such bundles with a* package-processing methodology. *We illustrate our framework and the gas economy it can bring with a case study where we analyze a popular Ethereum smart contract.*

1. Introduction

Decentralized applications (DApps) are fast-growing domains that have the potential to disrupt conventional businesses. They aim to tackle the mistrust crisis in many security-oriented businesses. However, the intensive use of blockchains and their associated high costs hinder DApps' broader adoption. To address this issue, existing research mainly focuses on designing new protocols at the blockchain level. These protocols are designed without the legacy platform of an operational blockchain and deployed DApps in mind. In this context, this work proposes a framework to optimize transaction fees by implementing a conflict-avoiding packaging of multiple smart-contract calls into a single transaction. It relies on two main concepts: a way to optimize the bundling of transactions with a *conflict-avoiding packaging strategy* and a way to execute such bundles with a *package-processing methodology*. To motivate the approach, this work considers a typical DApp architecture, where a DApp client holding an Ethereum account sends a transaction on the Ethereum blockchain to invoke a smart-contract function there. Despite having the potential to improve the efficiency and scalability of the Ethereum network, such methods have not been widely adopted or implemented in the real-world Ethereum ecosystem. This lack of adoption is attributed to several design challenges that arise from the tradeoff among packaging's security, cost-effectiveness, and timelines; a time window needs to be used to wait for the transactions to be packaged. We illustrate the workings of our framework and the gas economy it can bring with a case study where we analyze the TetherToken smart contract: a popular Ethereum contract.

2. Background

Ethereum is a public blockchain-based distributed computing platform. Simply put, it can be considered a large computer composed of small computers spread all over the world [Buterin 2016]. It is possible to write applications, called smart contracts, that run on this

global platform that ensures availability, and resistance to censorship, fraud, or third-party interference. It also allows transfers of its native cryptocurrency, called *Ether*, between parties without the need for a central authority. Ethereum smart contracts are identified by an *address*. They have a *balance* of Ether they own which is managed by their code. Furthermore, developers can create decentralized applications that typically rely on a combination of a smart contract and an off-chain service that interacts with this contract. Dapps have been used to implement decentralized versions of traditional services such as loans, security registries, voting systems, regulatory compliance and more.

These advantages come at a cost. *Gas* in Ethereum is a unit that measures the amount of computational effort that is required to perform certain operations. To have a transaction executed, a user is required to pay a fee proportional to the amount of gas it requires. This amount-of-Ether-per-gas-unit rate is the *gas price* and it is set by the user creating the transaction. The block producers (i.e. miners) of this platform collect these transaction fees as a reward for producing blocks.

It is possible to highlight two main reasons for the existence of gas in Ethereum. Firstly, it is a financial incentive for the block producers of the network: the greater the number of miners, the more secure the blockchain. The other reason is to ensure the termination of smart contract executions. There is a limit on the amount of gas that a transaction is allowed to consume.

3. Related work

Reducing the transaction cost for blockchain applications is essential for their real-world adoption and several approaches have been proposed for that.

Layer-one protocols or base layers of a blockchain system, which define the fundamental rules and functionalities of the blockchain, such as sharding and other designs, have been proposed to reduce the cost of blockchain applications. They require launching a new blockchain network from scratch, or a hard-fork. However, it is known to be difficult to bootstrap or implement hard-forks in a large-scale blockchain [Kokoris-Kogias et al. 2018] [Luu et al. 2016].

Layer-two protocols are another approach that focuses on designing add-ons to a deployed blockchain system by designing extensions, including smart-contracts on-chain and services off-chain. Payment networks are a notable example of layer-two designs that place most of the application logic of making a series of micropayments off the blockchain while resorting to the blockchain for control operations (e.g., opening and closing a channel) and error handling. In this approach, a payment channel "batches" multiple repeated micro-payments into minimally two transactions. State channels generalize the idea to support the game-based execution of smart contracts [Antonopoulos et al. 2021] [Cheng and Zhang 2019] [Dziembowski et al. 2018][Miller et al. 2017].

Contract optimization approaches try to identify and resolve the problems with patterns, syntax, or semantics that cause smart contracts to consume an excessive amount of gas [Chen et al. 2017, Marchesi et al. 2020, Albert et al. 2020, Li 2021]. This line of work is orthogonal to the Layer-one and Layer-two Protocols.

Batching transactions is a method of combining multiple transactions into one. Our approach is orthogonal to the above protocols. For instance, Layer-2 protocols pro-

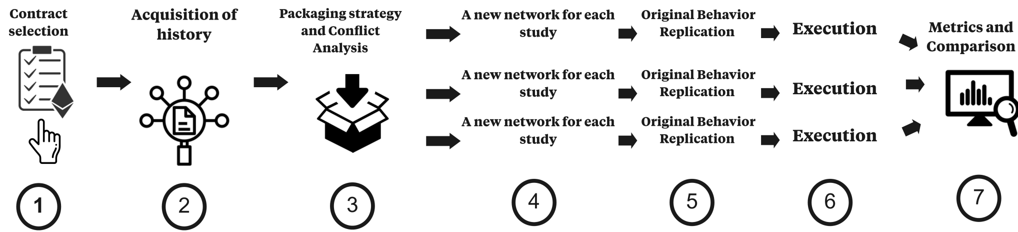

Figure 1. The seven phases of the framework proposed

vide a mechanism to execute batches of transactions but they are not concerned with methodologies on how to create these batches in an optimal way, i.e., by analysing conflicts between transactions. Users can submit multiple transactions at once by either using a smart contract that executes them together [Multisend 2023] or a tool/service that broadcasts them as a single transaction [Wang et al. 2021] [Hughes et al. 2021].

4. Proposed framework and case study

Packaging transactions on the Ethereum network is a technique used to save on gas costs by combining the execution of multiple transactions and batching them. We present a systematic framework to analyze the impact of such packaging on existing Ethereum smart contracts. Our approach analyses contracts by executing the contract in a testing environment, and then comparing the execution results to the historic execution results of the contract on the blockchain. Moreover, we present a case study where we use our framework to analyze the smart contract TetherToken which provides tangible evidence that a methodology to package transactions can reduce gas consumption.

Our framework requires an input contract instance identified by its address in Ethereum. The phases of our framework are depicted in Figure 1. The first three phases are performed only once for the input contract and the subsequent four stages are customized for each type of batch-execution approach. For our case study, we analyze the TetherToken smart contract [TetherToken website] at address `0xdac17f958d2ee523a2206206994597c13d831ec7`. Broadly speaking, it is an ERC20 contract instance implementing a stable cryptocurrency pegged to the US dollar. We chose this contract as it was the contract that spent the most gas in the period between May 2021 and October 2021. Their gas consumption (188243044506075 units) is associated, of course, to a large number of calls (26010105) that also provides evidence of its popularity.

The *history acquisition step* focuses on obtaining the transactions carried out by the contract on the Ethereum blockchain. Despite all the information being public available on the Ethereum blockchain, retrieving the history requires a costly transaction-by-transaction redemption interaction. In order to speed up the survey of a contract's history, Google's BigTable [Google Bigtable 2022] was used, which provides public tabular data on Ethereum transactions.

The *packaging strategy with conflict analysis step* attempts to create bundles of Ethereum transactions without introducing transaction conflicts. Conflicts arise when two transactions cannot both be executed in a given order. For instance, let A be an address with zero balance. A transaction t_1 that transfers some sum Y out of A cannot happen be-

fore a transaction t_2 that transfers X into A; these transactions would represent a *conflict* in this order. However, for $X \geq Y$, t_2 can happen before t_1; no conflict. Aside from conflicts, another piece of information that needs to be taken into account is the time frame used to package transactions. Users of such a system have to be aware of how long their transaction might be held up for the sake of saving gas via packaging. Our goal is to offer a complete solution for avoiding conflicts while packaging Ethereum transactions. Currently, we have analyzed conflicts for our case study manually by examining the Solidity code and deducing potential conflicts between different contract calls. In future work, our objective is to streamline and automate the process; this will involve data and control flow analysis and, possibly, will allow only partial automation.

The *network creation step* attempts to achieve a significant level of similarity in reproducing the behavior of a smart contract. To this end, we create a local Ethereum network mapping the original addresses to the new network addresses adapting the original calls to these new addresses.

The *behaviour recreation step* relies on a *package processing methodology* to execute batched transactions. This methodology consists of (i) a strategy to represent — with a way to encode and decode — a package of transaction and (ii) a strategy to execute each of these packaged transactions. Note that while the packaging strategy is in charge of deciding which transactions are to be (logically) bundled together, the package processing methodology defines the format of these packages and how they are executed. In our case study, two distinct package-processing methodologies were implemented. The first was the addition of a new function to the smart contract being analyzed that processes a sequence of function calls in the form of a Solidity array; we call this approach `commandArray`. The second approach consists of encoding calls using the `multisend` (library) methodology [Multisend 2023].

In the *execution step*, for a given recreated history of the contract according to a packaging strategy and processing methodology, our framework executes each of the created packages in the appropriate ordering. As a baseline for our analysis, we also replay the original contract with its unaltered history. The execution of these packages is implemented by test cases in the Truffle framework [Truffle] with the support of the Ganache framework[Ganache 2022]. During the execution, a total gas expense report is extracted, which is essential to the last step of the framework.

In the *metrics and comparison step*, the gas expense reports generated by the executions are compared.

4.1. Results

In our packaging strategy, we decided to limit the time frame for batching transactions to a five-minute window. We considered that this time trade-off would be reasonable for customers who are willing to wait a little longer to complete their transactions on the blockchain. All history of the contract resulted in 50793 batches of transactions from address `0x2faf487a4414fe77e2327f0bf4ae2a264a776ad2` to the TetherToken contract.

Table 1 presents some of the results we obtained with our case study. The table on the left-hand side shows examples of packages extracted from our case study execution. It illustrates the sort of gains that are expected from different package sizes (in number

Samples of batches extracted from execution		
Amount of transactions	%Gain multisend	%Gain commandArray
1	-4.69%	51.56%
2	46.09%	74.22%
4	68.02%	83.78%
6	80.14%	86.52%
8	80.66%	89.31%
10	87.02%	90.85%

Global analysis with 10050 batches		
Total Amount of transactions	Total %Gain using multisend	Total %Gain using commandArray
21352	50.47%	70.55%

Table 1. TetherToken case study results.

of transactions). We can see significant gains in gas savings, and the expected trend is that gains increase with the increase in the number of transactions in a package. It is possible to see that the approach using the `multisend` has a greater gain in relation to the `commandArray`. We assume that this advantage is due to the encoding of the arguments made off-chain, sending the data to the smart contract already in binary there is an economy. However, it also decreases as the number of bundled transactions increases. The table on the right shows the global analysis related to the 100050 first batches. The material with all analysis data can be accessed in the paper [Paper repository 2023].

4.2. Threats to the validity

We can point out two main threats to the validity of our case study. The first is the limited sample size. Since the study considered only one smart contract and one caller in this case study, it may not be representative of the broader Ethereum network. A small sample size can make it difficult to generalize the findings to other contexts. The second is the maturity of the tools used to recreate the blockchain and the different histories we are interested in.

5. Conclusion

This paper presents a systematic framework for analyzing the impact of packaging transactions on existing Ethereum smart contracts. We used the TetherToken smart contract as a case study to demonstrate the effectiveness of our approach. The framework involves recreating and analyzing a "packaged history" of the blockchain according to a packaging strategy and different package-processing methodologies. The results show significant gains in gas savings demonstrated by two different approaches.

References

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., and Rubio, A. (2020). Gasol: Gas analysis and optimization for ethereum smart contracts. In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020*, pages 118–125. Springer.

Antonopoulos, A. M., Osuntokun, O., and Pickhardt, R. (2021). *Mastering the Lightning Network.* " O'Reilly Media, Inc.".

Buterin, V. (2016). What is ethereum? *Ethereum Official webpage. Available: http://www. ethdocs. org/en/latest/introduction/what-is-ethereum. html.*

Chen, T., Li, X., Luo, X., and Zhang, X. (2017). Under-optimized smart contracts devour your money. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 442–446. IEEE.

Cheng, R. and Zhang, F. (2019). Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200. IEEE.

Dziembowski, S., Faust, S., and Hostáková, K. (2018). General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 949–966.

Ganache (2022). `https://trufflesuite.com/docs/ganache/`. Accessed: 2022-05-22.

Google Bigtable (2022). `https://cloud.google.com/bigtable/docs/overview`. Accessed: 2022-05-22.

Hughes, W., Russo, A., and Schneider, G. (2021). Multicall: A transaction-batching interpreter for ethereum. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 25–35.

Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., and Ford, B. (2018). Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE.

Li, C. (2021). Gas estimation and optimization for smart contracts on ethereum. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1082–1086. IEEE.

Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. (2016). A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 17–30.

Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., and Tigano, D. (2020). Design patterns for gas optimization in ethereum. pages 9–15. IEEE.

Miller, A., Bentov, I., Kumaresan, R., and McCorry, P. (2017). Sprites: Payment channels that go faster than lightning. *CoRR, abs/1702.05812*.

Multisend (2023). `https://github.com/gnosis/ethers-multisend`. Accessed: 2023-05-30.

Paper repository (2023). `https://github.com/medeirosbm/coloquio-sbs-2023`. 2023.

TetherToken website. `https://tether.to/`. Accessed: 2022-05-22.

Truffle. `https://trufflesuite.com/docs/truffle/`. Accessed: 2022-05-22.

Wang, Y., Zhang, Q., Li, K., Tang, Y., Chen, J., Luo, X., and Chen, T. (2021). ibatch: saving ethereum fees via secure and cost-effective batching of smart-contract invocations. pages 566–577.