Gas Optimization Patterns in Move Smart Contracts on the Aptos Blockchain

Eric Keilty*, Keerthi Nelaturu*, Anastasia Kastania† and Andreas Veneris*
* *Department of Electrical & Computer Engineering, University of Toronto*
† *Athens University of Economics and Business*
{eric.keilty, keerthi.nelaturu}@mail.utoronto.ca, ank@aueb.gr, veneris@eecg.toronto.edu

***Abstract*—The Move language provides superior security and verifiability compared to existing smart contract languages such as Solidity. As the language becomes more popular, gas optimization will become an important field of research. This paper presents the first work on gas optimization in the Move language. We chose Aptos as the underlying platform for our analysis since it is the leading Move-enabled blockchain platform, and it was the first to develop a gas meter. In this paper, we describe Aptos' gas meter in detail. Then, we analyze the vast research on gas optimization in Solidity, and explore how it can be implemented in the Move language. Finally, this paper proposes 11 gas optimization patterns and principles for the Move language, presents 5 patterns that decrease the time complexity of the smart contract but have no effect on gas consumption, and implements a sample smart contract for each proposed gas optimization pattern. Our results show that the proposed patterns reduce gas consumption in a typical smart contract by $7-56\%$.**

***Index Terms*—Aptos; Move; Smart Contract; gas optimization**

I. Introduction

The *gas* of a smart contract is the cost of executing its logic on the blockchain. The *gas meter* refers to the mechanism that determines how much gas should be charged for a given smart contract. Gas optimization is a highly researched area in Solidity [1]–[13], but as of the time of writing, there is no equivalent work done for the Move language [14]. This paper presents the first work in the field of gas optimization in Move.

We have chosen Aptos as the underlying platform for studying gas optimization of the Move language. It is currently the leading blockchain platform that utilizes the Move language, and most importantly it was the first Move-enabled blockchain platform to implement a gas meter. Other platforms, such as OpenLibra [15] and StarCoin [16], have yet to implement gas meters. Sui [17] has modified core Move and integrated its own features. Thus, its gas analysis is left as future work.

The contribution of this work is summarized as follows:

- Detail the nuances of Aptos' gas meter for Move.
- Enumerate gas optimization patterns for Move.
- Enumerate patterns that reduce contract time complexity, but have no effect on the gas cost.
- Provide concrete examples which implement the proposed optimization patterns and evaluate their effectiveness in typical Move smart contracts.

The rest of the paper is organized as follows. In Section II, we detail the gas calculation of Move smart contracts in Aptos. Section III summarizes prior work on gas optimization in Solidity, and discusses how it can be applied to Move. Sections IV and V provide the proposed gas optimization and non-optimization patterns in Move. Section VI gives empirical results of sample smart contracts to validate, evaluate, and quantify the optimization patterns proposed in Section IV. Finally, Section VII concludes the work.

II. The Aptos Gas Meter

The *gas* of a smart contract is the cost of storing its items and executing its logic on the blockchain. It is necessary to pay for the use of blockchain resources and to avoid excessive consumption of resources. The *gas meter* refers to the mechanism that determines how much gas should be charged for a given smart contract deployment and invocation.

In Aptos, the native token is `APT` and the unit of gas is `Octa`. However, the Aptos gas meter operates using *internal gas units* where

$$100 \textit{ internal gas units } = 1 \texttt{ Octa } = 10^{-8} \texttt{ APT} \quad (1)$$

This gives a more fine-grain measurement of gas, which is then rounded after all calculations have been completed.

When a transaction is submitted, the user must include, among others, the following fields:

- `max_gas_amount`: The maximum number of gas units that the transaction sender is willing to spend to execute the transaction. This determines the maximum computational resources that can be consumed by the transaction.
- `gas_price`: The gas price per unit the transaction sender is willing to pay, determined by the market.

When the transaction is executed by the MoveVM, it keeps a tally of the amount of gas used according to the gas meter. The total gas charged for the transaction is

$$\textit{total gas fee} = (\textit{gas used}) \times \texttt{gas\_price} \quad (2)$$

If the *gas used* surpasses `max_gas_amount`, then the transaction is aborted. Thus, the maximum amount a user can be charged is (`max_gas_amount`) × (`gas_price`).

The *gas used* by a transaction consists of summing the gas associated with the size of its payload, the virtual machine instructions it executes, and the global storage it accesses. This is explicitly expressed as follows.

$$\textit{gas used} = (\textit{payload gas}) + (\textit{instruction gas}) + (\textit{storage gas}) \quad (3)$$

Each gas consumption type is discussed next.

A. Payload Gas

The payload gas is the cost associated with publishing a transaction to the blockchain, i.e. the *transaction size*. In Move, *modules* define functions and custom data types. Transactions are submitted via *transaction scrips*, which import modules and utilize their definitions. When publishing a module, the bytecode is stored on the blockchain. Thus, the transaction size only depends on the length of the bytecode. When publishing a transaction script, module functions and their inputs need to be stored on the blockchain. Thus, the transaction size also depends on the size of the input parameters.

Equations 4 and 5 show the payload gas calculation. Every transaction is automatically charged $1,500,000$ internal gas units ($15,000$ `Octas`). This is sometimes called *intrinsic gas*. If the bytecode is greater than 600 bytes, the transaction

is charged $2,000$ internal gas units for each of the excess bytes [18]. This is to prevent abuse of the network.

$$\textit{large tx penalty} = \max(0,\ (\textit{tx size} - 600 \text{ bytes}) \times 2,000) \quad (4)$$

$$\textit{payload gas} = 1,500,000 + \textit{large tx penalty} \quad (5)$$

B. Instruction Gas

The *instruction gas* is the gas associated with the execution of the virtual machine operations of a transaction. Each instruction of the MoveVM has been assigned a gas cost. Typically, each operation charges both a fixed *base gas* and a variable amount of gas proportional to the parameter sizes associated with the operation [19].

Since the MoveVM is a 64-bit stack-based virtual machine, instructions operate on *exactly* 64 bits. Thus, operations such as arithmetic, bitwise, boolean, and comparison charged per 64 bits. As a result, from the perspective of the MoveVM, there is no distinction between `u8` and `u64` integers. Operations on `u8` and `u64` integers will result in the same gas consumption. Conversely, operations on `u128` integers will generally require more gas, since they require at least one additional register.

Move modules are not executed when published to the blockchain. However, the instruction gas associated with a module function will be considered as the amount of gas it consumes when a transaction script executes it.

C. Storage Gas

The *storage gas* is the gas associated with accessing global storage on the blockchain. In Aptos, the global state of the blockchain consists of a set of *accounts*, containing of a set of *items*, which are *key-value pairs*. Move smart contracts can access global storage via `move_to`, `borrow_global`, `borrow_global_mut`, and `move_from`, which create, read, write, and delete items, respectively.

The gas consumed by each access type (except deletion) is calculated via Equation 6. Deletion of a resource from global storage does not consume gas [20].

$$\textit{storage gas} = \textit{items} \times (\textit{per item gas}) + \textit{bytes} \times (\textit{per byte gas}) \quad (6)$$

The first term is the base cost for accessing an item. This amount is the constant up-front cost of the access type. The second term accounts for the size of the item, charging more gas if more bytes are accessed. Note that *bytes* in Equation 6 refers to the total number of bytes in all fields of a resource, even if only once field was accessed. Table I gives the amount of gas charged for each access type [20].

TABLE I
STORAGE GAS FEES

Operation	Internal Gas Units	Octas
per-item read	$300,000$	$3,000$
per-item write	$300,000$	$3,000$
per-item create	$300,000$	$3,000$
per-byte read	300	3
per-byte write	$5,000$	50
per-byte create	$5,000$	50

III. RELATED WORK

To date, this paper is the first work that analyzes gas optimization for the Move language. The majority of the research on gas optimization is done for Solidity on the Ethereum blockchain [1]–[3].

Broadly, there are three ways to frame gas optimization. The first is in the field of vulnerability detection. An *out-of-gas* exception occurs when the smart contract uses more gas than the allowed gas limit. The authors of [4] use symbolic execution to detect specific out-of-gas exception patterns. The authors of [5] use fuzzing techniques in order to find inputs that cause a high gas output.

A second way to frame gas optimization is to abstract the problem to code optimization [6], [8]–[12]. If the smart contract code is optimized, then the virtual machine will perform fewer operations, and thus gas consumption will be reduced. The authors of [6] developed the static analyzer tool, GASPER, which applies parallelized symbolic execution to Solidity bytecode to identify specific patterns such as dead code, opaque predicates, and expensive operations in loops. The authors of [12] identify three loop patterns to reduce the number of virtual machine operations and global storage accesses.

Finally, gas optimization can be viewed as its own unique subject. The gas meter is not isomorphic to "the number of virtual machine operations". For example, addition and division are often given the same gas price per operation, even though division typically requires more virtual machine operations to compute. Some techniques will make the smart contract less efficient in the strict sense of virtual machine operations but nonetheless, reduce gas consumption. The authors of GASPER extended their work to develop GASREDUCER [7], which identifies 24 different optimization patterns in bytecode. The authors of [13] created a tool called GASOL, which targets optimizing gas consumption associated with the usage of storage operations by replacing multiple accesses to global memory with local variable operations.

In this paper, we wish to see how this large body of work can be applied to the Move language. Generally, code optimization techniques that operate on the source-code level can be directly applied. However, code optimization techniques that operate on the bytecode level cannot be directly applied, since the EVM and MoveVM are fundamentally different. Finally, some techniques related to reducing the number of global storage accesses can be modified for Move.

IV. GAS OPTIMIZATION PATTERNS

Generally, gas optimization stands very close to time and space complexity optimization. The aim is to minimize the *number of virtual machine operations* and the *amount of global storage accessed* during a transaction. However, there is often a trade-off between time and space complexity with many ways to implement the same specification. Decreasing memory use may result in more virtual machine operations, and decreasing virtual machine operations may require an increase in memory use; in which case, it is not clear how to minimize gas consumption.

This section gives both general principles and concrete design patterns for optimizing the gas consumption of Move smart contracts on Aptos. For each gas optimization, an example smart contract has been constructed which demonstrates how it can be applied (see Section VI).

A. Payload Gas

The gas associated with the payload is typically much less than the gas associated with instructions and global storage. Thus, for most applications, its contribution is negligible. However, if the payload greatly exceeds 600 bytes, then it may cause a noticeable increase due to Aptos' large transaction penalty. We give the following general principles for payload gas optimization.

1) Minimize the Length of Modules: The code of a published module is stored on the blockchain, which consumes gas. Minimizing the length of the module, i.e. the number of bytes required to store its bytecode, reduces the total gas

cost. Some instances include removing dead or unnecessary code, reducing redundant code, avoiding unnecessary additional variables, using standard libraries, and separating out the module into multiple smaller modules. Note that comments do not have an effect on this calculation, since the blockchain stores the bytecode and not the source code.

2) Minimize the Size of Parameters in Transaction Scripts: When executing a transaction script, the payload may contain the values of the parameters given by the user, which are stored on the blockchain and thus consumes gas. Minimizing the number and size of these parameters will reduce the total gas cost. For example, combining many small functions that require a lot of parameters into one larger function, and also avoiding passing resources as parameters into functions.

B. Instruction Gas

The gas associated with virtual machine instructions is akin to the time complexity of the smart contract. In general, less virtual machine operations mean less gas consumption. However, this is not always possible without sacrificing the necessary functionality. We give the following general principles for instruction gas optimization.

1) Limit Function Calls: One of the most expensive instruction gas operations are function calls [19]. The gas saved from the lack of a function call is always larger than the gas gained from a larger module size. Therefore, abstracting smart contract functionality into helper functions should be avoided as much as possible. For instance, it is very common for programmers to write getter functions which are a single line or very few lines of code. Removing these is a small change, which saves a large percentage of gas (see Table II). However, having large and complicated functions makes testing more difficult. It is up to the developer to find an acceptable balance.

2) Minimize Vector Element Operations: Vector operations charge gas on a per-element basis, and are more expensive than operations on local variables. Thus, accessing vectors can be treated like accessing the global state, allowing us to apply principles 1) and 3) from Section IV-C analogously for vectors. If one wishes to operate on an element from a vector more than once, then it should be copied to a local variable and then updated after all calculations are performed. Lastly, a vector element should be directly updated, rather than deleted and recreated.

3) Short Circuit: When using the logical connective `AND` (`&&`), if the first expression evaluates to `false`, then the second expression will not be evaluated. Likewise, when using the logical connective `OR` (`||`), if the first expression evaluates to `true`, then the second expression will not be evaluated. Thus, continued expressions in if-statements and while-loops should be ordered by increasing gas cost. If a cheap expression short-circuits the condition check, then we save on evaluating the more expensive expressions.

4) Write Values Explicitly: Since virtual machine operations consume gas, any constant values should be written explicitly rather than implicitly computed via the smart contract.

5) Avoid Redundant Operations: Since virtual machine operations consume gas, redundant operations should be avoided. For example, Move has a bytecode verifier that checks for common vulnerabilities such as integer overflow/underflow. Thus, checking for this in a smart contract is redundant and unnecessary.

C. Storage Gas

The gas associated with global storage is akin to the space complexity of the smart contract. In general, less accesses to global storage will result in less gas consumption. Moreover, storage gas will typically dominate both payload and instruction gas. Thus, it should be given the most attention when optimizing smart contract gas. The following general principles are given for storage gas optimization.

1) Operate on Local Variables: Operating directly on resources and resource fields consumes significantly more gas than operating on local variables. Whenever a smart contract is operating on the values of a resource, its ownership should be borrowed by a local variable. If necessary, those values can be transferred back to the resource at the end of the function.

This pattern can be implemented when accessing a resource field value in a loop. One should first store its field value in an intermediate local variable and do all loop operations on this local variable. At the end of the function, the resource is updated. This limits the number of accesses to the resource to a maximum of two, rather than the number of loop iterations.

2) Variable Packing: There are two facts about Move's gas meter that this pattern utilizes. First, global storage access consumes the most gas of any operation. Thus, we should aim to make as few as possible. Second, when accessing a resource, the per-byte charge consists of all fields in the resource, not just the ones that were accessed.

Variable packing refers to representing many variables of data using a single resource field. For example, consider variables `x8`, `x32`, and `x24` that will only ever store 8, 32, and 24 bits of information, respectively. The naive way of storing these is to separate them each into their own field. However, we can save on storage by packing these variables into a single `u64` integer, and use bitwise masking to unpack the variables. Thus, saving on per-item global accesses.

3) Resource Update: There is currently no incentive to deallocate global storage. Thus, in order to minimize gas consumption, unused resources should be overwritten rather than deallocating and creating new resources.

4) Read Instead of Write: Writing to a resource is more expensive per byte than reading. Thus, a resource should only be written to if necessary.

V. Non-optimization

In addition to giving principles that will minimize gas consumption, it is equally useful to know what does not affect gas consumption.

1) Operation Types: Basic arithmetic operations (*add*, *sub*, *mul*, *div*, *mod*) cost the same amount of gas, even though division, for example, typically requires more computation than addition. Bit-wise operations (*and*, *or*, *xor*, *left shift*, *right shift*) cost the same amount of gas. Similarly, the comparison operations ($<, >, \leq, \geq$) cost the same amount of gas, and the operations ($=, \neq$) both cost slightly less.

2) Reads/Writes are Never Partial: Reading or writing only one field from a resource may save a little gas with respect to the instruction gas, but it does not save any gas with respect to storage gas. When `borrow_global_mut` is called and a resource field is updated, the per-byte cost of the update is for the entire resource, not just the updated field.

3) `u8` Integers: There is no difference between doing operations with `u8` integers and `u64` integers, both locally and globally. This is because MoveVM has 64-bit registers. However, doing operations with `u128` integers will cause an increase in gas usage.

4) Ordering Fields in Resources: The order of the fields of the resource does not matter. All fields of a resource occupy their own space in storage. One can pack variables within a field, but not between fields.

5) Deallocation of Resources: Currently, Aptos is lacking any mechanism for rewarding the destruction of resources via `move_from`. Although, they have expressed interest in adding this in the future.

VI. Experiments

This section presents the results of an experimental evaluation of the gas optimization patterns that were identified in Section IV. Through this, we validate, evaluate, and quantify the gas savings of each gas optimization pattern. While Move is rapidly gaining popularity, it has yet to become a standard in decentralized application development. As a result, there are fewer examples of smart contracts deployed to Aptos as well as a lack of developer tools and standardized benchmarks. Thus, we have created a set of sample smart contracts[1] designed to isolate each gas optimization pattern. Table II compares the gas consumption of the original and optimized smart contracts measured in `Octa`. The rightmost column is the percent decrease of gas in the optimized contract. The patterns "Minimizing the Length of Modules", "Minimizing the Size of Parameters in Transaction Scripts", and "Avoiding Redundant Operations" were omitted as these are general principles rather than concrete design patterns.

TABLE II
Gas savings comparison of optimization patterns

Gas Optimization Pattern	Original Cost	Optimized Cost	Gas Savings Percent
Limit Function Calls	47	26	44.7
Minimize Vec Element Ops	41	30	26.8
Short Circuit	2372	2	99.9
Write Values Explicitly	410	2	99.5
Operate on Local Variables	62	27	56.5
Variable Packing	746	630	15.5
Resource Update	130	120	7.7
Read Instead of Write	3663	56	98.5

The gas optimization patterns "Minimize Vector Element Operations", "Short Circuit", "Write Values Explicitly", and "Read Instead of Write" heavily depend on the particular smart contract. The gas optimization patterns "Variable Packing" and "Resource Update" depends on the size of the global storage that is being accessed. The sample smart contracts use moderately sized resources, and we see the percent decrease is small. For applications with large global variables, these patterns are effective at reducing gas consumption. However, for applications with small global storage values, the impact may be negligible. Lastly, the gas optimization patterns "Limit Function Calls" and "Operate on Local Variables" are the most stable with respect to changes in the smart contract. Both patterns result in a substantial percent gas decrease.

VII. Conclusion

Move is a new smart contract language that offers superior security and verifiability compared to existing smart contract languages. This paper is the first to apply the vast research on gas optimization in Solidity to the Move language using Aptos as the underlying platform. We proposed 11 gas optimization patterns and identified 5 patterns that decrease the time complexity of a smart contract but have no effect on gas consumption. We implemented sample contracts for each proposed gas optimization pattern. Our results showed that our gas optimization patterns reduce gas consumption on a typical smart contract by $7 - 56\%$.

As Move becomes more popular, future work includes surveying deployed Move smart contracts to analyze the frequency of these design patterns, and empirically determine their impact on gas optimization. Subsequently, tools for detecting and optimizing these design patterns can be developed. Another avenue for future work includes an analysis of gas optimization on Move bytecode. This work focused purely on optimizations at the source-code level; however, there is a great deal of research on Solidity bytecode gas optimization to pull from. A final prospect for future work includes performing a similar analysis on Sui's version of Move and doing a comparison with Aptos. Likewise can be done with StarCoin and OpenLibra provided they develop their own gas meters.

[1] https://github.com/Veneris-Group/Move-Gas-Optimization-Patterns

References

[1] B. Severin, M. Hesenius, F. Blum, M. Hettmer, and V. Gruhn, "Smart money wasting: Analyzing gas cost drivers of ethereum smart contracts," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp. 293–304.

[2] C. Signer, "Gas cost analysis for ethereum smart contracts," Master's thesis, ETH Zürich, Department of Computer Science, 2018.

[3] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020, pp. 9–15.

[4] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, oct 2018. [Online]. Available: https://doi.org/10.1145/3276486

[5] F. Ma, Y. Fu, M. Ren, W. Sun, H. Song, Y. Jiang, J. Sun, and J. Sun, "V-gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability," 2021.

[6] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 442–446.

[7] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, 2018, pp. 81–84.

[8] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, may 2019. [Online]. Available: https://doi.org/10.1109%2Fwetseb.2019.00008

[9] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, "Characterizing efficiency optimizations in solidity smart contracts," in *2020 IEEE International Conference on Blockchain (Blockchain)*, 2020, pp. 281–290.

[10] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of super-optimized smart contracts using max-smt," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. cham: Springer International Publishing, 2020, pp. 177–200.

[11] J. Nagele and M. A. Schett, "Blockchain superoptimizer," 2020.

[12] K. Nelaturu, S. M. Beillahi, F. Long, and A. Veneris, "Smart contracts refinement for gas optimization," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, 2021, pp. 229–236.

[13] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," 2019.

[14] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. R. Rain, S. Sezer *et al.*, "Move: A language with programmable resources," *Libra Assoc.*, 2019.

[15] 0L Network, https://0l.network/, 2023, accessed on 04/13/2023.

[16] The StarCoin Team, "Starcoin: A Hierarchical Smart Contract Based Decentralized Finance Network," StarCoin, Tech. Rep.

[17] The MystenLabs Team, "The Sui Smart Contracts Platform," Sui, Tech. Rep., 2023.

[18] "Aptos labs," GitHub repository, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/blob/cdb1f27868890a49075356d626e91d73f8ee3170/aptos-move/aptos-gas-meter/src/meter.rs

[19] A. Labs, GitHub repository, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/blob/3791dc07ec457496c96e5069c494d46c1ff49b41/aptos-move/aptos-gas-schedule/src/gas_schedule/instr.rs

[20] ——, "Computing transaction gas," GitHub repository, 2023. [Online]. Available: https://github.com/aptos-labs/aptos-core/blob/3791dc07ec457496c96e5069c494d46c1ff49b41/developer-docs-site/docs/concepts/base-gas.md