International Journal of Advanced Smart Convergence Vol.9 No.2 157-163 (2020)
http://dx.doi.org/10.7236/IJASC.2020.9.2.157

IJASC 20-2-18

Cost Minimization of Solidity Smart Contracts on Blockchain Systems

Wan Yeon Lee[1]

Professor, Dept. of Computer Science, Dongduk Women's University, Seoul 02748, South Korea,
Email: wanlee@dongduk.ac.kr

Abstract

Recently the blockchain technology has been actively studied due to its great potentiality. The smart contract is a key mechanism of the blockchain system. Due to the short history of the smart contract, many issues have not been solved yet. One main issue is vulnerability and another main issue is cost optimization. While the vulnerability of smart contract has been actively studied, the cost optimization has been rarely studied. In this paper, we propose two cost optimization methods for smart contracts running on the blockchain system. Triggering a function in a smart contract program code may require costs and it is repeated continuously. So the minimization of costs required to trigger a function of smart contract while maintaining the performance equally is very important. The proposed two methods minimize the usage of expensive permanent variables deployed on the blockchain system. We apply the proposed two methods to three prevalent blockchain platforms: Ethereum, Klaytn and Tron. Evaluation experiments verify that the proposed scheme significantly reduces the costs of functions in the smart contract written with Solidity.

Keywords: Blockchain, Smart contract, Cost minimization, Automatic convert, Solidity

1. Introduction

The blockchain is a digital ledger system duplicated and distributed across the network of computer systems. The Bitcoin cryptocurrency is the first generation blockchain system that maintains a record of transactions across several computers linked in a peer-to-peer network [1]. The Ethereum is the second generation blockchain system that maintains a record of digital contracts as well as cryptocurrency transactions in a peer-to-peer network [2]. This digital contract is made by programming codes and referred to as *smart contracts*. Because of great commercial potentiality of smart contracts, other many blockchain systems with their own designs for the smart contract have been launched: Klaytn, Tron, EOS, Qtum, Cosmos, Cardano, ICON, etc.

Because blockchain systems require voluntary participations of several separate computers, they provide some cryptocurrency as the benefit to join in their blockchain systems. If the computing resources handling a transaction is larger, the charged cryptocurrency becomes severer. So the minimization of computing resources leads to cost optimization of transactions in the blockchain system. The most expensive resource in the smart

Manuscript Received: May. 7, 2020 / Revised: May. 11, 2020 / Accepted: May. 15, 2020
Corresponding Author: wanlee@dongduk.ac.kr(Wan Yeon Lee)
Tel: +82-2-940-4685, Fax: +82-2-940-4170
Professor, Dept. of Computer Science, Dongduk Women's University, Korea

contract is to store a record permanently because all computers in the blockchain system cooperate to make consensus and store the record separately in their ledger storages.

In this paper, we propose an automatic convert scheme of the given smart contract code so as to minimize the usage of permanent variables. The proposed scheme consists of two methods: replacing permanent variables with temporary variables if possible and skipping the storing operation of permanent variables if their new values are equal to their exiting values. The proposed scheme is implemented into a software tool with JAVA programming. This tool searches for optimizable code patterns with unnecessary usages of permanent variables, and converts automatically a found code pattern into a new code pattern with the minimal usages of permanent variables. We apply sample smart contracts written with Solidity to three prevalently used blockchain platforms: Ethereum, Klaytn and Tron. Experimental results shows that the proposed scheme reduces manifestly the costs of smart contracts by up to about 95%.

The rest of this paper is organized as follows; Section 2 reviews the related previous studies and explains the system model considered in this article. Section 3 describes the proposed scheme in detail. Section 4 shows evaluation results and Section 5 provides concluding remarks.

2. Previous Studies and System Model

Most of previous studies [3, 4, 5, 6] for the blockchain system focused on vulnerability analysis of smart contracts. Some recent attacks such as DAO and Parity MultiSig Wallet and SmartMesh resulted in big money loss. So many commercial services, called *security audit* [7], that check whether there exists security problem in a smart contract have been launched. Besides of vulnerability analysis, another main issue of smart contracts is the transaction costs of smart contracts. The blockchain system requires some costs of cryptocurrency for transactions of smart contracts. If a smart contract requests expensive costs for its transactions due to its bad design, its cumulative cost loss becomes more severe as the number of called transactions grows.

Only a few recent studies [8, 9] handled the problem to minimize the cost of smart contracts. Chen et al. [8] dealt with the problem of reducing the cost of smart contract on the level of bytes codes. In contrast, the proposed scheme detects the optimizable patterns of high-level programming languages familiar to humans, instead of bytes codes. Whereas users can easily understand the difference of the original codes and the modified optimal codes and choose better one in the proposed scheme, they can't in the previous study [8]. Our previous study [9] proposed an optimization method that reduces the number of shifts of permanent array variables. This paper proposes two additional optimization methods that are different from and work together concurrently with the previous method [9]. Whereas the previous schemes [8, 9] just detect optimizable patterns but not convert automatically them into their optimized patterns, the proposed scheme detects and converts automatically the optimizable patterns into their optimized patterns.

As introduced in our previous study [9], the computing resources depends on the platform of blockchain systems and thus the cost of smart contract depends on the platform type of blockchain systems. In the Ethereum, Klaytn, Tron, Qtum, and ICON platforms, the usage of permanent variables dominates the costs of transaction, although the exact cost of the same transaction may vary depending on the type of blockchain platform. For example, in the Ethereum platform and the Klaytn platform, the cost of changing a value in persistent variables is 5000 gas from non-zero or 20,000 gas from zero, while the cost of arithmetic operation is 3 gas or 5 gas [10, 11]. In the ICON platform, the cost of changing a value in persistent variables is 320 steps per byte, while the cost of arithmetic operation is zero [12].

In this study, we focus on minimizing the usage of permanent variables. Beside of the usage of permanent variables, deploying (or updating) operation of the smart contract requires a lot of costs. But we do not consider

the cost of deploying the smart contract because it is one-time cost. The proposed scheme is applicable to most of public blockchain platforms, but not applicable to some blochchain platform such as the public EOS platform [13] and private platforms (e.g., Hyperledger Fabric [14]). In the EOS platform, the cost of changing a value in persistent variables is negligible while the foot print size and the execution speed dominate the costs of a transaction.

3. Proposed Scheme

The proposed scheme employs two code optimization methods. The first optimization method is to find the *unnecessary permanent variables* in the given smart contract and replace them with *temporary variables*. This replacement reduces the cost of smart contract, because the cost of temporary variable is much cheaper than the cost of permanent variable. The second optimization method is to find the *non-conditional modification* of permanent variables and replace them with the *conditional modification*. This replacement reduces the number of modifying the stored values of permanent variables, because the conditional modification excludes the unnecessary modification caused when the values before the modification and after the modification are same.

Figure 1 shows the flow chart of the proposed scheme. The scheme first parses the program code of the given smart contract into word tokens. Next the scheme searches the code pattern of unnecessary permanent variables. If it finds the pattern, it replaces unnecessary permanent variables with temporary variables with the same variable name. After completing the optimization of unnecessary permanent variables, the scheme searches the code pattern of non-conditional modification of permanent variables. If it finds the pattern, it replaces the non-conditional modification pattern with the conditional modification pattern by inserting the condition checking code.

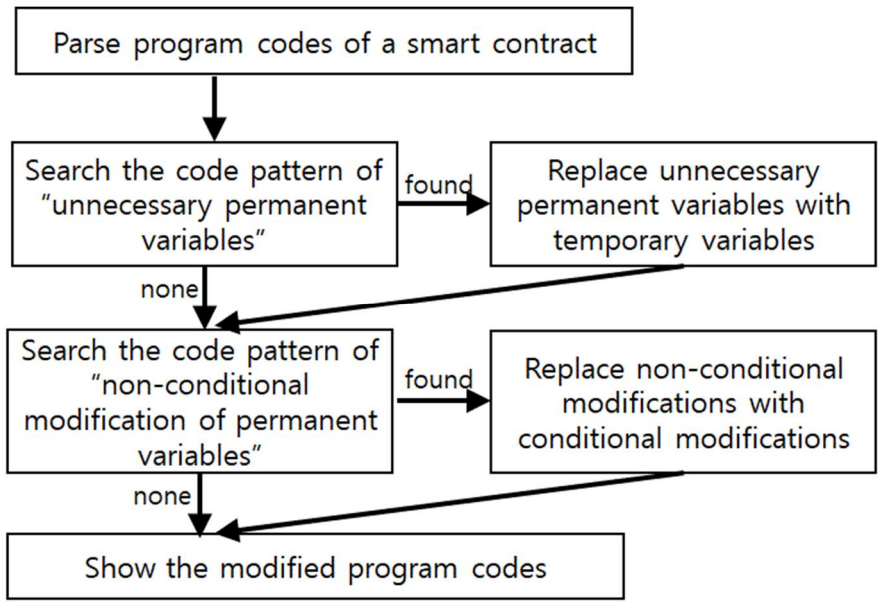

Figure 1. Flowchart of Proposed Scheme

Figure 2 shows the working example of the first proposed method to optimize unnecessary permanent variables. The given smart contract is written with solidity language the most prevalently used on the blockchain platforms. Figure 2(a) shows the given smart contract code before optimization, and Figure 2(b) shows the modified code part after optimization. The line 7 in Figure 2(a) is replaced with line 7 in Figure 2(b). In Figure 2(a), "rand_array" variable defined as a global variable variable at line 4 is a permanent and

"tmp_array" variable defined as a local variable at line 7 within the "adddTimeToRandom" function is permanent. While the "rand_array" variable is necessary and unreplaceable, the "tmp_array" variable is replaceable with temporary variable. The "storage" variable type of "tmp_array" at line 7 of Figure 2(a) is replaced with the "memory" variable type at line 7 of Figure 2(b), where the variable definition of "memory" generates a temporary variable. The variable names before the optimization and after the optimization are same with "tmp_array" and the function operations before the optimization and after the optimization are equal.

```
1: pragma  solidity 0.4.25;
2: contract sample1 {
3:      uint        constant   array_length = 10;
4:      uint[]      rand_array = new uint[]( array_length );
5:
6:      function addTimeToRandom( ) external returns( uint[] ) {
7:                  uint[]    storage   tmp_array = rand_array;
8:                  for( uint i = 0; i < array_len; i++ ) {
9:                          tmp_array[ i ] = (rand_array[ i ] + uint256(block.timestamp)) % 1000;
10:                 }
11:                 return     tmp_array;
12:     }
13: }
```

(a)

```
7:                  uint[]    memory   tmp_array = new uint[]( array_length);
```

(b)

Figure 2. The First Optimization Method

Figure 3 shows the working example of the second proposed method to optimize non-conditional modification of permanent variables. Figure 3(a) shows the given smart contract code before optimization, and Figure 3(b) shows the modified code part after optimization. The line 8 in Figure 3(a) is replaced with line 8 and 9 in Figure 3(b). In Figure 3(a), the array permanent variable "time_array" is always modified for each of 10 array elements at line 8, even when their stored values are equal to 1 before modifications. On the contrary, at line 8 and 9 in Figure 3(b), the array variable "time_array" is modified only when there stored values are different before and after modification. If the stored values of "time_array" are equal to 1 before modification, any array element is modified not at all. The function operations before the optimization and after the optimization are equal. Compared with the code size before optimization, the code size after optimization is increased by one. But its burden is negligible.

```
1: pragma  solidity 0.4.25;
2: contract sample2 {
3:      uint        constant  array_length = 10;
4:      uint[]      time_array = new uint[]( array_length );
5:
6:      function resetArray( ) external {
7:              for( uint i = 0; i < array_len; i++ ) {
8:                      time_array[ i ] = 1;
9:              }
10:     }
11: }
```

(a)

```
7:              for( uint i = 0; i < array_len; i++ ) {
8:                      if( time_array[i]  != 1 )
9:                              time_array[ i ] = 1;
10:             }
```

(b)

Figure 3. The Second Optimization Method

4. Evaluation

For evaluation of the proposed scheme, we implement the proposed scheme into a software tool with JAVA programming on Ecliple IDE Jee Neon. The implemented tool converts automatically the given smart contract code into an optimized smart contract code. We apply the two sample codes of Figure 2(a) and Figure 3(a) to the implemented tool and compare the cost of triggering the functions. For an evaluation metric, we adopt "Cost Reduction Ratio" which is defined as "(cost of original code – cost of optimized code)/(cost of original code) × 100". We run 100 times and display their average value.

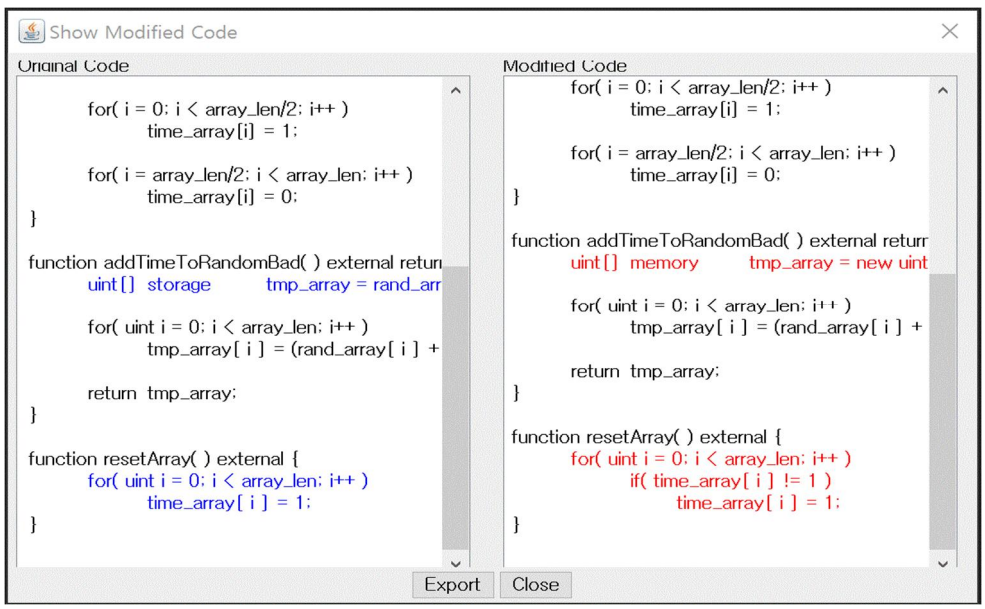

Figure 4. Implemented Tool

Figure 4 shows a running result of the implemented software tool. The left side of Figure 4 shows the given

original smart contract before optimization, and the right side shows the modified smart contract after optimization. This tool searches for optimizable code parts marked with blue color in the left side and automatically converts them into optimized codes marked with red color in the right side. If users prefer to the optimized smart contract rather than the original smart contract, they can save the optimized smart contract by clicking the "Export" button on the bottom.

Table 1. Cost Reduction Ratio of the First Proposed Method

Value of array_length	Ethereum Platform	Klaytn Platform	Tron Platform
10	65.0%	64.9%	92.2%
20	66.7%	74.9%	92.8%
40	70.9%	77.4%	93.2%
80	73.9%	83.9%	93.3%
160	74.3%	87.7%	93.4%

Table 1 shows the performance of the first optimization method (called Replacement of Unnecessary Permanent Variables). We examine the performance against various lengths of unnecessary permanent variable because the performance depends on the length of unnecessary permanent variables. We also examine the performance on three different blockchain platforms: Ethereum, Klaytn and Tron. In Table 1, we confirm that the proposed scheme reduces slightly more costs as the value of "array_length" in the code of Figure 2(a) increases. According to the used blockchain platforms, the cost reduction amounts and ratios are different because the blockchain platforms have different cost policies.

Table 2. Cost Reduction Ratio of the Second Proposed Method

Value of array_length	Ethereum Platform	Klaytn Platform	Tron Platform
10	71.0%	81.8%	94.0%
20	77.1%	88.2%	94.4%
40	80.6%	91.7%	94.6%
80	82.5%	93.6%	94.7%
160	83.9%	94.6%	94.8%

Table 2 shows the performance of the second optimization method (called Non-conditional Modification Checking of Permanent Variables). Similarly to the experiment of Table 1, we examine the performance against various numbers of non-conditional modifications on the three different blockchain platforms. In this experiment, we set 50% of the original values in "time_array" array variables to be different from the modified values (50% of the original values are set to 1 and the rest 50% are set to 0). In Table 2, we confirm that the proposed scheme reduces slightly more costs as the number of "array_length" in the code of Figure 3(a) increases. According to the used blockchain platforms, the cost reduction amounts and ratios are different because the blockchain platforms have different cost policies.

5. Conclusions

In this paper, we propose two optimization methods to reduce the cost of triggering the function of smart

contracts. The proposed two methods minimizes the usage of expensive permanent variables from the given smart contract. The first proposed method is to replace unnecessary permanent variables with temporary variables. The second proposed method is to skip the modification of permanent variables if their original stored values are equal to their modified values. We implement the proposed scheme into a software tool and apply sample smart contracts to the implemented tool. Through practical experiments on three prevalent blockchain platforms, we confirm that the proposed two optimization methods can reduce significantly the cost of triggering the function of smart contracts by up to about 95%.

In future study, we will investigate more optimizable code patterns besides the proposed two patterns. We also investigate optimizable code patterns for other languages besides of solidity.

Acknowledgement

This research was supported by the Dongduk Women's University Grant, 2019.

References

[1] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, Cryptography Mailing list at https://metzdowd.com, 2009.
[2] Gavin Wood (2014) Ethereum: A secured decentralized generalized transaction ledger, Ethereum project yellow paper 151: 1-32.
[3] Ethereum Smart Contract Security Best Practices, https://consensys.github.io/smart-contract-best-practices/
[4] EOS Smart Contract Security Best Practices, https://github.com/slowmist/eos-smart-contract-security-best-practices/blob/master/README_EN.md
[5] S. Kalra, S. Goel, M. Dhawan and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts," Annual Network and Distributed System Security Symposium, vol. 25, Feb. 2018, DOI: https://doi.org/10.14722/ndss.2018.23092
[6] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee and Hakjoo Oh, "VERISMART: A Highly Precise Safety Verifier for Ethereum Smart Contract," IEEE Symposium on Security and Privacy, vol. 41, May 2020.
[7] OpenZeppelin, https://openzeppelin.com/contracts/
[8] Ting Chen, Xiaoqi Li, Xiapu Luo, Xiaosong Zhang, "Under-optimized Smart Contracts Devour Your Money," IEEE International Conference on Software Analysis, Evolution and Reengineering, vol. 24, Feb 2017, DOI: https://doi.org/10.1109/SANER.2017.7884650
[9] Wan Yeon Lee and Yun-Seok Choi, "Vulnerability and Cost Analysis of Heterogeneous Smart Contract Programs in Blockchain Systems," Current Trends in Computer Sciences and Applications (Mar. 2020), vol. 2, no. 1, pp. 142-145, DOI: https://doi.org/10.32474/CTCSA.2020.02.000126
[10] Estimating Transaction Costs of Ethereum, http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html#estimating-transaction-costs
[11] Transaction Fees of Klaytn, https://docs.klaytn.com/klaytn/design/transaction-fees
[12] Step Estimation of ICON, https://www.icondev.io/docs/transaction-fees
[13] EOSIO Developer Portal, https://developers.eos.io/
[14] Hyperledger Fabric, https://www.hyperledger.org/projects/fabric