Maximizing Efficiency in Smart Contract Execution Costs Techniques for Cost Optimization on Blockchain Networks

Rahul Raghavan Tharammal [a, *], Prashant Nitnaware [b]

[a] *Department of Information Technology, Pillai College of Engineering, Maharashtra, India*
E-mail: trahul22me@student.mes.ac.in ; ORCID: https://orcid.org/0009-0002-7262-8975
[b] *Department of Information Technology, Pillai College of Engineering, Maharashtra, India*
E-mail: pnitnaware@mes.ac.in;

Abstract. Blockchain is like a digital ledger that is not owned or controlled by any single organization. Instead, it's spread out over many computers all around the world. This decentralization removes the need for middlemen when people communicate with each other. Thanks to platforms like Ethereum, which use blockchain technology, we can now not only do financial transactions but also create and run programs called Smart Contracts on the network. However, using these Smart Contracts comes with a cost. When they run, they consume resources on the Ethereum network, and users have to pay for this consumption with something called gas fees. These fees depend on how complex the operations in the contracts are. So, it's important to design Smart Contracts in a way that minimizes these gas costs. In this paper, we suggest methods for organizing Smart Contract to make them more efficient in terms of gas usage. By carefully planning how these contracts are structured and what they do, we can reduce the amount of gas needed to run them, making it cheaper and more efficient to use Smart Contracts on Ethereum.

Keywords: Blockchain, Gas cost, smart contract, Polygon Blockchain.

1. Introduction

The integration of blockchain technology into cryptocurrency trading has opened up new possibilities, especially with the emergence of the Ethereum network [1]. Ethereum goes beyond just digital currency—it allows for broader applications, one of which is smart contracts [2]. These are like self-executing programs that run on the Ethereum blockchain without needing middlemen. Unlike regular software development, creating and using smart contracts on Ethereum isn't free. Each action on the blockchain comes with a cost. Smart contracts automate many tasks, from money transfers to agreements, and everything is recorded permanently on the blockchain. This change not only transforms how transactions work but also brings new concerns, especially around security and how transactions happen, which are different from regular software development practices. Using smart contracts can make processes smoother and ensure that records are transparent and can't be tampered with. However, there's a cost associated with executing them on the Ethereum network.

This paper aims to explore the limitations encountered when developing and running smart contracts, specifically focusing on the Ethereum network. It also seeks to suggest a framework for smart contracts that would minimize execution costs.

*Corresponding author. E-mail: trahul22me@student.mes.ac.in

2. Related Work

T. Chen, X. Li, J. Luo, and X. Zhang [3] focus on an important issue: the problem of smart contracts failing to work properly on blockchain, especially Ethereum They point out how the written code bad can increase the cost of gas to fulfil smart contracts, of which The result can be excessively costly for users or developers By scrutinizing Solidity, the recommended Ethereum programming language on, the authors identify seven coding practices that consume a lot of gas, classify them into two groups and to address these inefficiencies they introduce a tool called GASPER It is designed to detect them actual gas consumption measures Preliminary analysis shows that a significant number of real-world contracts have these measures, which are found to be inefficient in about 93.5% of the cases. Their work is important because it reveals optimization issues in Solidity and provides practical solutions for identifying and solving these problems. But while there may be limitations to the tool's ability to identify all gas-intensive systems and the challenges of optimizing existing contracts, their analysis provides a basis for improving the efficiency and cost-effectiveness of smart contract execution on Ethereum and possibly other blockchain platforms.

Authors Ting Chen, Yuzheng Feng, Jihao Li, and Hao Zhou [4] introduced GasChecker, a sophisticated tool designed to identify codes in smart contracts that actually caused the excess gas Their contribution is to develop a new way of teaching ten dysfunctional systems with intelli- . Uses SE to identify them in contract bytecodes To efficiently handle the analysis of millions of contracts, Map-reduce program model and feedback based load balancing were used for equalization. The methodology proposed that GasChecker shows strong scalability, as demonstrated in extensive experiments. An empirical study of deployed smart contracts reveals widespread inefficiencies in code, with manual detection showing a low false positive rate of 2.5% Although GasChecker offers improvements great for in inefficient regulatory identification though because of the complexity of some contractual analyses and the resources used It can present limitations and yet remains an important tool for will make smart contract development and implementation more efficient and reliable.

E.Albert, J.Correas, P.Gordillo, and A.Rubio [5] have collaboratively developed GASOL, a comprehensive tool for analysing and optimizing gas usage in Ethereum smart contracts. GASOL provides various cost models to accurately estimate gas consumption for different types of instructions in the Ethereum Virtual Machine (EVM). This allows users to assess both gas consumption and the frequency of bytecode instructions with precision. With its range of cost models, GASOL meets different needs, whether users want to analyse storage operations or evaluate the cost of specific lines of code. GASOL proves invaluable for developers in identifying inefficient storage patterns that can significantly impact gas usage. Additionally, it offers an optional automatic optimization feature for selected functions, simplifying the gas optimization process. Integrating GASOL into an Eclipse plugin for Solidity enhances the development environment by visually displaying gas and instruction bounds and facilitating the creation of gas-optimized Solidity functions. It's worth noting that while GASOL provides valuable insights and automation for gas optimization, its effectiveness may vary depending on the complexity and specific requirements of smart contracts. Therefore, additional validation and testing may be necessary to ensure optimal results across different contract scenarios.

In their paper Tamara Brandstatter, Stefan Schulte, Jürgen Sito, and Michael Borkowski [6] explore ways to reduce gas taxes and make Solidity smart contracts more efficient they introduce 25 optimization methods and build a model which can recognize and be an active part of this development. When more than 3,000 open-source smart contracts are analysed on etherscan.io, 471 entries are found in 204 contracts. Their work provides a systematic approach to improving Solidity smart contracts, potentially saving users a lot of money. However, some complications such as the complexity of certain processes and the risk of unintended consequences when simply changing contract law require further research to address those issues this to improve the effectiveness of the proposed methods.

3. Background

3.1. Blockchain

Blockchain is a game-changing technology that acts as a decentralized, shared digital ledger, allowing transactions to be tracked transparently across multiple computers This means that once a transaction is recorded, it cannot be changed without the consent of the entire network, due to its change logic It turns out revealed that subsequently all records are changed The idea of blockchain surfaced in 2008 when an individual or group using the name Satoshi Nakamoto published a paper on bitcoin.org. This paper introduced a novel payment system that does not rely on traditional financial intermediaries. In the case of Bitcoin, there are two key players in the blockchain network: users and miners. Users perform transactions, while miners maintain the integrity of the system by adding these transactions to the blockchain. Miners are motivated to participate because they can obtain updated cryptocurrencies by monitoring transactions. Bitcoin's blockchain was the first example of this technology, adapted for cryptocurrency transactions. However, today there are many other blockchain platforms that serve different industries and needs. Examples include Ethereum [1], Corda [7], and Quorum [8], each of which offers unique features and capabilities beyond mere currency exchange.

3.2. Smart Contract

Smart Contracts were first thought up by Szabo in 1997 [9]. These contracts are like little computer programs that run on the Ethereum network. They can carry out specific tasks all by themselves once certain conditions are met, without needing any human to step in. Because they're on the blockchain, nobody can change them or stop them once they're running. It's like they're set in stone. And once a transaction happens with a Smart Contract, it can't be undone [2].

3.3. Polygon Blockchain

Polygon [10] is a solution designed to make Ethereum, the popular platform for decentralized applications (dApps) and smart contracts, more efficient and user-friendly. It tackles issues like slow processing and high fees by working as a layer 2 scaling solution on top of Ethereum. At its heart, Polygon uses a system called Plasma, which creates additional chains connected to the main Ethereum chain. These extra chains, called "child chains," can handle transactions and smart contracts independently, easing the strain on the main Ethereum network and boosting its performance. Polygon also switches to a Proof of Stake (PoS) consensus mechanism, which is more environmentally friendly compared to Ethereum's current Proof of Work (PoW) mechanism. This change further improves scalability and sustainability. A standout feature of Polygon is its seamless integration with Ethereum. Developers can easily move their existing dApps and smart contracts to Polygon without extensive modifications, making it a convenient choice for creating scalable and efficient decentralized applications.

3.4. Deployment cost of smart contract

Usually, smart contracts are used to process financial transactions through rules defined in the code. These conventions can be written in a variety of programming languages, including those that are Turing-complete, meaning that any computation can be performed. These contracts are then converted into bytecode instructions and executed on the EVM [10]. EVM is more efficient in handling complex calculations. Solidity is a popular programming language for creating smart contracts, and each contract is compiled into bytecode instructions [11,12], or opcodes. It is important to consider that the consumption of these opcodes comes with a cost, and the Ethereum yellow-paper [13] provides detailed information on the value of each opcode Gas Tracker [5] is a tool for real-time data flow gas prices, which directly affects the speed of the transaction. (09 March 2024) Suggested fuel prices are subject to change and are detailed in the table 1 below.

Table.1 Mandatory Fees in Ethereum (Gas price in Gwei)

Price of Gas cost (Gwei)	Deployment Cost ($)	Speed of process
75	6.50	Low - 3 mins: 0 secs
75	6.50	Average - 3 mins: 15 secs
76	6.66	High - 30 secs

4. Proposed Method for Optimization Of Gas

According to many studies explained in chapter 2, it is clear that poorly designed smart contracts can consume significant amounts of gas, with financial consequences say comes Currently to improve these contracts No standards are widely agreed upon, making customization difficult. Different aspects of smart contracts should be considered for possible code optimization. These include transforming data types, variable packing, managing memory and storage, handling constants, mappings, and arrays, optimizing repeat

Optimizing smart contracts involves improving their structure and functionality, focusing on selecting suitable variable types and implementing functions efficiently. To achieve this, a systematic approach is necessary, outlined in a recommended sequence of steps or algorithm for writing smart contract source code. Figure 1 shows the process involves analyzing different application segments and performing specific optimization activities, each requiring a detailed description of the step-by-step actions to enhance smart contracts effectively active computations in loops, eliminating redundant code, and error handling effective strategies.

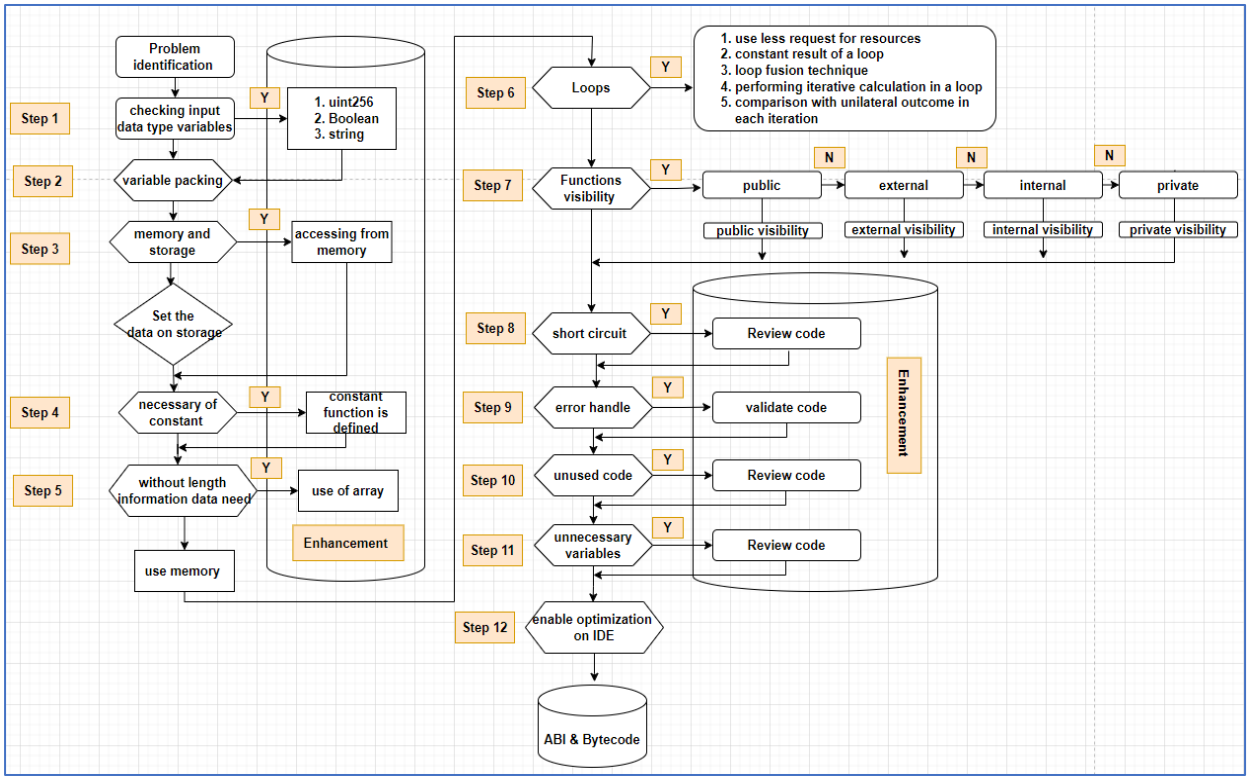

Fig. 1. Proposed Methodology for Gas Enhancement

4.1. Data Type variable step 1

4.1.1. uint256 and uint16

Choosing the right data type for smart contracts is important for optimization, especially when it comes to reducing storage usage and gas costs in EVMs EVM works with 256-bit/32-byte words, so int256, uint256 to make better use of the available space It is important to use data types such as, or bytes32. For integers, it is recommended to use uint256 to reduce gas costs. Using other types of uint requires conversion to uint256, which adds additional gas costs. Let us consider an example, we have two identical contracts, one using uint256 (Algorithm 1a) and the other using uint16 (Algorithm 1b) [2] When we compile and execute, we find that gas costs created for uint256 variables is gas 84,333. while the uint16 variables for was 85,533 gas. Increased gas costs for the uint16 variable because operations associated with this data type require additional instructions. Converting 16-bit values to the EVM's original 256-bit format requires additional instructions, increasing the dispatcher's gas bill.

Algorithm 1a

```
1. contract AdditionExample {
2. function addUint256(uint256 a, uint256 b) public pure returns (uint256) {
3. // Adding two uint256 numbers
4. return a + b;
5. }
```

Algorithm 1b

```
1. contract AdditionExample {
2. function addUint16(uint16 a, uint16 b) public pure returns (uint16) {
3. // Adding two uint16 numbers
4. return a + b;
5. }
```

4.1.2. String vs Byte32

In the first algorithm 2a, we use a string data type to hold the user's name. Each character in a string requires additional storage, and can consume a lot of gas through actions such as inserting or retrieving a username, which can be as high as 358,191 due to the dynamic nature of the string Momma we don't do it. Bytes32 is a fixed size array of 32 bytes, which means it has a fixed storage cost and can be more gas efficient than strings, which cost about 73,921 However, it is necessary that a string can be stored up to 32 characters by bytes32 Its length is limited wom. Overall, although string enables information data to be handled, using bytes32 can provide gas efficiency and storage optimization, especially for smaller strings. Your smart contract of specific needs when making gas consumption, storage efficiency, and data suitability decisions for monitoring information data.

Algorithm 2a

```
1. contract StringExample {
2. string public userName;
3. function setUserName(string memory _name) public {
4. userName = _name;
5. }
6. function getUserName() public view returns (string memory) {
7. return userName;
8. }
9. }
```

Algorithm 2b

```
1. contract Bytes32Example {
2. bytes32 public userName;
3. function setUserName(bytes32 _name) public {
4. userName = _name;
5. }
6. function getUserName() public view returns (bytes32) {
7. return userName;
8. }
9. }
```

4.1.3. Boolean

When it comes to optimizing smart contracts for gas costs and storage usage, the choice of data types also extends to simple Boolean values In Ethereum smart contracts, Boolean values are often represented by the uint8 data type, where 0 is false and 1 is true. There are a few advantages to using uint8 for Boolean values. First, it closely matches the 256-bit password-size of the Ethereum Virtual Machine (EVM), making it easy to add Boolean values to other data types in repositories Second, it's done functions of uint8 efficiently for EVM, of which The result reduces gas costs compared to using other data types or performing custom Boolean operations For example, let us assume a contract two identical ones each using the original Boolean flags, one using uint8 Algorithm 3a and the other using bool Algorithm 3b. The gas cost is lower for based Boolean variables, which occurs at 101745, compared to 82,527 for bool variables because the EVM is inherently more efficient with uint8 operations, while bool operations may require additional modifications or instructions for gas-. The cost is much higher.

Algorithm 3a

```
1. contract BooleanUint8 {
2. uint8 public flag;
3. function setFlag(bool _value) external {
4. if (_value) {
5. flag = 1;
6. } else {
7. flag = 0;
8. }
9. }
10. function getFlag() external view returns (bool) {
11. return flag == 1;
12. }
13. }
```

Algorithm 3b

```
1. contract BooleanBool {
2. bool public flag;
3. function setFlag(bool _value) external {
4. flag = _value;
5. }
6. function getFlag() external view returns (bool) {
7. return flag;
8. }
```

4.2. Variable packing step 2

In the EVM, how you arrange variables in your smart contract can have a big impact on how efficiently storage space is used. Let's say we have three variables: a large one (a *uint256*), a medium-sized one (a *bool*), and a small one (a *uint8*). If we declare these variables one after the other, starting with the

uint256, then the *bool*, and finally the *uint8*, each variable will take up its own chunk of storage, which could leave some space unused. This is shown in Algorithm 4a. But if we rearrange the order of declaration to put the smaller variables first (the *bool* and *uint8)*, and then the larger one *uint256*, we can fit them all into the same storage chunk efficiently, as shown in Algorithm 4b.

Algorithm 4a

```
1. contract StorageExample {
2. uint256 public num;
3. bool public flag;
4. uint8 public smallNum;
5. }
```

Algorithm 4b

```
1. contract StorageExample {
2. bool public flag;
3. uint256 public num;
4. uint8 public smallNum;
5. }
```

4.3. Memory and storage step 3

As shown in algorithm 5 memory and storage are important concepts for data management. Memory is temporary storage that is only available during the execution of a task. They are used to store on-demand data for a short period of time and do not need to be permanently stored on the blockchain. In contrast, storage refers to a persistent cache of data that runs between function calls and is permanently stored on the Ethereum blockchain. Changes stored in memory are deleted when a job is completed, while changes stored in storage persist even after execution. In the provided Solidity example, the contract StorageExample shows how both memory and storage are used. The addDataToMemory function specifies memory usage by creating a temporary memory array tempData, adding data, and returning a value. This data is only available during the execution of a project and is not stored permanently on the blockchain. In contrast, the addDataToStorage function defines storage functionality by adding data to a data array, which is stored permanently in the blockchain. The getDataFromStorage function accesses the data structure and retrieves data from storage. It is worth noting that storage operations are more expensive in terms of gas consumption compared to memory operations.

Algorithm 5 – Memory and storage

```
1. contract StorageExample {
2. uint256[] public data; // This array is stored in storage
3. // Function to add data to the array stored in memory
4. function addDataToMemory(uint256 _value) public pure returns(uint256) {
5. uint256[] memory tempData; // Declaring a memory array
6. tempData.push(_value); // Adding data to the memory array
7. return tempData[0]; // Returning the data (not stored permanently)
```

```
8.  }
9.  // Function to add data to the array stored in storage
10. function addDataToStorage(uint256 _value) public {
11. data.push(_value); // Adding data to the storage array
12. }
13. // Function to retrieve data from the array stored in storage
14. function getDataFromStorage(uint256 _index) public view returns(uint256) {
15. require(_index < data.length, "Index out of range");
16. return data[_index]; // Retrieving data from the storage array
17. }
18. }
```

4.4. Constant step 4

SSTORE and SLOAD are known to require a lot of resources when used in an Ethereum Virtual Machine (EVM). In design scenarios, it is often necessary to configure variables in the build or implementation phase and then make them immutable when interacting with the smart contract to address this, developers can use persistent or immutable variables has been used. This variable tells the Solidity compiler that the variable [17] will not change for the lifetime of the contract. Algorithm 6, constants are defined in the code using these variables. It is highly recommended that you use the constant keyword whenever possible to declare a constant in the code. This helps to save gas by reducing the number of times the SLOAD instruction is used, which can be expensive, costing 200 gas each time it's called.

Algorithm 6

```
1.  contract MyContract {
2.  // Declaring a constant state variable
3.  uint256 constant MY_CONSTANT_VALUE = 100;
4.  // Function to read the constant value
5.  function getConstantValue() public view returns (uint256) {
6.  return MY_CONSTANT_VALUE;
7.  }
8.  }
```

4.5. Mapping and array step 5

Mappings, which work like hash tables or dictionaries in other programming languages algorithm 7a, are excellent for storing and retrieving key-value pairs efficiently. They're particularly useful when you need to organize data based on unique identifiers, such as Ethereum addresses. They offer efficiency and flexibility, making data management streamlined and keeping gas expenses relatively low. Arrays, on the other hand, store data sequentially algorithm 7b and are great for maintaining ordered collections. However, as the size of the array increases, so do the gas costs. While arrays might be suitable for smaller data types like uint8, they can become inefficient and costly for large datasets. In general, it's recommended to prioritize mappings due to their overall efficiency and flexibility. The gas cost for mappings is approximately 228,467, while

for arrays, it's around 145,188. However, arrays should still be considered carefully, especially when sequential storage is essential, and the increase in gas costs is acceptable.

Algorithm 7a

```
1. contract MappingExample {
2. mapping(address => uint256) public balances;
3. function deposit() public payable {
4. balances[msg.sender] += msg.value;
5. }
6. function withdraw(uint256 amount) public {
7. require(balances[msg.sender] >= amount, "Insufficient balance");
8. balances[msg.sender] -= amount;
9. payable(msg.sender).transfer(amount);
10. }
11. }
```

Algorithm 7b

```
1. contract ArrayExample {
2. uint256[] public numbers;
3. function addNumber(uint256 _number) public {
4. numbers.push(_number);
5. }
6. function getNumber(uint256 _index) public view returns (uint256) {
7. require(_index < numbers.length, "Index out of bounds");
8. return numbers[_index];
9. }
10. }
```

4.6. Loops step 6

Unlike conventional software, Ethereum contracts are limited in their performance due to gas costs. If gas runs out before a transaction is completed, the whole process stops. However, payments are still made for the parts of the transaction, even if all failed. That’s why it’s important to accurately predict how much gas you’ll need before you start working. Current methods often struggle to account for gas for loop operations because the number of times a loop will run can be unpredictable. About a quarter of all contracts are loops and they tend to use more gas than any other activity [14]. That’s why you need to carefully optimize your code, especially the parts that involve loops [3].

4.6.1. Minimize requests for resource intensive loop operations.

Let’s us take example of two functions that illustrate different approaches to calculating the sum of elements in an array while minimizing the risk of expensive loop operations. The first function in algorithm 8a, *sum*, uses a typical loop structure to go through the array and accumulate the sum. While straightforward,

this method can lead to high gas costs, especially for large arrays. To tackle this issue, the second function algorithm 8b, *sumWithLimit*, takes a more cautious approach. It employs a `while` loop and explicitly checks both the iteration count and the array length against a predefined maximum threshold. This ensures that the loop terminates within a reasonable number of iterations. By doing so, the function prevents potential gas inefficiencies and reduces the risk of out-of-bounds access, thus enhancing the contract's reliability and cost-effectiveness

Algorithm 8a

```
1. contract ExpensiveLoopExample {
2. uint256 constant MAX_LOOP_ITERATIONS = 100; // Set a maximum number of iterations to prevent infinite loops
3. function sum(uint256[] memory data) public pure returns (uint256) {
4. uint256 total = 0;
5. // Instead of using a loop, directly calculate the sum
6. for (uint256 i = 0; i < data.length; i++) {
7. total += data[i];
8. }
9. return total;
10. }
11. }
```

Algorithm 8b

```
1. contract ExpensiveLoopExample {
2. uint256 constant MAX_LOOP_ITERATIONS = 100; // Set a maximum number of iterations to prevent infinite loops
3. function sumWithLimit(uint256[] memory data) public pure returns (uint256) {
4. uint256 total = 0;
5. uint256 iterations = 0;
6. // Explicitly check for the array length to avoid going beyond the array boundaries
7. while (iterations < data.length && iterations < MAX_LOOP_ITERATIONS) {
8. total += data[iterations];
9. iterations++;
10. }
11. return total;
12. }
13. }
```

4.6.2. Constant result of a loop

Taking an example given in algorithm 9 of a contract with a function called *constantOutcomeLoop*. Inside this function, there's a *for* loop that runs exactly 10 times. In each iteration, a variable called *sum* adds up the value of the loop counter *i*. Since the number of iterations is fixed, and the operations inside the loop are straightforward (just adding the loop counter to the sum), the result of this function will always be the same,

no matter when or how many times it's called. This pattern is handy when you need your smart contract to produce consistent and dependable results every time it's executed

Algorithm 9

```
1. contract ConstantLoopExample {
2. function constantOutcomeLoop() external pure returns (uint256) {
3. uint256 sum = 0;
4. // Loop for 10 iterations
5. for (uint256 i = 0; i < 10; i++) {
6. sum += i;
7. }
8. return sum;
9. }
10. }
```

4.6.3. Loop fusion technique

Loop fusion is a technique where multiple tasks are combined into a single efficient process. In the provided in Algorithm 10, we have a scenario created in the *LoopFusionExample* contract. This contract contains two arrays: *data1* and *data2*, which are filled with random values. The objective is to manipulate these arrays and store the outcomes in two separate arrays: *results1* and *results2*. Typically, one would iterate through each array separately, performing calculations in distinct loops. However, loop fusion allows us to merge these steps into a unified operation. In the *processData* function, loop fusion is implemented. This function simultaneously traverses both *data1* and *data2* arrays. During each iteration, calculations are performed on elements from both arrays, and the results are stored in their corresponding result arrays

Algorithm 10

```
1. contract LoopFusionExample {
2. uint256[] public data1;
3. uint256[] public data2;
4. uint256[] public results1;
5. uint256[] public results2;
6. constructor() {
7. // Initialize data arrays with some values
8. data1 = [1, 2, 3, 4, 5];
9. data2 = [6, 7, 8, 9, 10];
10. }
11. function processData() public {
12. // Loop fusion: Combine two loops into one
13. // Initialize result arrays with the same length as data arrays
14. results1 = new uint256[](data1.length);
15. results2 = new uint256[](data2.length);
16. // Iterate over data1 and data2 in a single loop
```

```
17. for (uint256 i = 0; i < data1.length; i++) {
18. // Perform some computation on data1 element
19. results1[i] = data1[i] * 2;
20. // Perform some computation on data2 element
21. results2[i] = data2[i] * 3;
22. }
23. }
24. }
```

4.6.4. performing iterative calculations in a loop

The provided algorithm 11 contains two functions: *calculateSum* and *calculateProduct*, both designed to operate on arrays of unsigned integers. In the *calculateSum* function, the algorithm starts by setting a variable *sum* to zero. Then, it loops through each element of the array using a *for* loop. Within this loop, each element is added to the *sum*. After processing all elements, the final *sum* value is returned. Similarly, the *calculateProduct* function initializes a variable *product* to one and iterates through each element of the array using a *for loop*. Within the loop, each element is multiplied with the *product*. After processing all elements, the final *product* value is returned

Algorithm 11

```
1.  contract CalculationLoop {
2.  function calculateSum(uint256[] memory numbers) public pure returns (uint256) {
3.  uint256 sum = 0;
4.  // Loop through the array and calculate the sum
5.  for (uint256 i = 0; i < numbers.length; i++) {
6.  sum += numbers[i];
7.  }
8.  return sum;
9.  }
10. function calculateProduct(uint256[] memory numbers) public pure returns (uint256) {
11. uint256 product = 1;
12. // Loop through the array and calculate the product
13. for (uint256 i = 0; i < numbers.length; i++) {
14. product *= numbers[i];
15. }
16. return product;
17. }
18. }
```

4.6.5. Comparison with unilateral outcome in each iteration of the loop

In Algorithm 12, named *OneSidedComparison*, illustrates a scenario where a loop is used to go through an array of numbers and count how many of them exceed a particular threshold. When the contract initializes, it sets up an array of integers. The purpose of the function *countGreaterTha* is to determine the number of elements in the array that are higher than a specified threshold. In the loop, each element of the array is compared to the threshold using the expression *numbers[i] > threshold*. This comparison is termed as one-sided because it only results in *true* when the current number being evaluated is greater than the threshold; otherwise, it automatically yields *false*. As a result, there's no explicit need to handle the situation where the comparison is *false*, as the counting mechanism solely focuses on the *true* outcomes

Algorithm 12

```
1.  contract OneSidedComparison {
2.  uint[] public numbers;

3.  constructor() {
4.  // Initialize the array with some numbers
5.  numbers.push(5);
6.  numbers.push(10);
7.  numbers.push(15);
8.  numbers.push(20);
9.  numbers.push(25);
10. }

11. function countGreaterThan(uint threshold) external view returns (uint) {
12. uint count = 0;
13. for (uint i = 0; i < numbers.length; i++) {
14. // Compare if the number is greater than the threshold
15. if (numbers[i] > threshold) {
        a. count++;
16. }
17. }
18. return count;
19. }
20. }
```

4.7. Function step 7

The number of services included in a contract significantly affects the amount of money charged to execute the contract. This is because more work usually means more warnings, which in turn leads to more gas. Programs in Solidity can have different modes of identification: external, public, internal, and private. External transactions can only be invoked from other contracts, while public transactions can be invoked both internally and externally. Calling a public function requires memory allocation, which is a resource-intensive operation. This means that calling public functions takes gas because the input parameters need to

be copied into memory automatically. When comparing external and internal functions, it is generally better to use internal functions whenever possible. This is because internal function calls are translated into simple jumps in the Ethereum Virtual Machine (EVM), which reduces gas consumption while directly referring to the rest of the internal function calls [15]

4.7. Short-circuiting step 8

Short circuit is an essential concept in logical expressions, allowing for efficient evaluation by stopping computation as soon as the outcome is clear. This is particularly useful in conditions where certain operations may be unnecessary or potentially risky if executed needlessly. In Algorithm 13 provided, named *ShortCircuitExample*, the function *divideIfBNonZero* demonstrates short-circuiting by only performing the division *a / b* if the variable *b* is non-zero. If *b* equals zero, the evaluation immediately stops without attempting the division operation. This prevents a potential division by zero error and optimizes the code's execution. Similarly, the function *doSomething* showcases short-circuiting within an OR *(||)* condition. If *a > b* is *true*, the condition *b == 0* will not be evaluated due to short-circuiting, as the overall result of the *OR* condition is already known to be *true*

Algorithm 13

```
1.  contract ShortCircuitExample {
2.  uint public a = 5;
3.  uint public b = 0;
4.  uint public c = 10;
5.  function divideIfBNonZero() public view returns (uint) {
6.  // If b is zero, no need to proceed with the division
7.  if (b != 0 && a / b > 2) {
8.  return a / b;
9.  } else {
10. return 0;
11. }
12. }
13. function doSomething() public view returns (bool) {
14. // Short-circuiting in an OR condition
15. if (a > b || b == 0) {
16. return true; // If a > b, b == 0 will not be evaluated due to short-circuiting
17. } else {
18. return false;
19. }
20. }
21. }
```

4.9. Error handling step 9

As shown in algorithm 14, *ErrorHandlingExample*, different methods are demonstrated to ensure the reliability and security of the smart contract functions. Firstly, let's look at the *withdraw* function. This

function makes use of *require* statements to set specific conditions before allowing a withdrawal. It ensures that only the contract owner can initiate withdrawals and that the withdrawal amount is greater than zero. If either condition is not met, the transaction is reversed with an appropriate error message. This helps prevent unauthorized access or invalid withdrawals. Secondly, the *divide* function shows how to prevent division by zero errors. It checks if the divisor (*b*) is not zero before performing the division operation. If *b* is zero, indicating a division by zero scenario, the transaction is reverted with a custom error message. This ensures the integrity of computations by avoiding potential arithmetic exceptions. Lastly, the *assertExample* function demonstrates an *assert* statement used for internal consistency checks within the contract. This function verifies that a specific condition (*a* being greater than or equal to *b*) holds true. If the assertion fails, indicating a critical logical error within the contract code, the transaction is reverted. This prevents potentially erroneous or unexpected behaviour, maintaining the contract's internal consistency and reliability. These error handling methods, including *require*, *revert*, and *assert*, are essential tools in Solidity programming

Algorithm 14

```
1. contract ErrorHandlingExample {
2. address public owner;

3. constructor() {
4. owner = msg.sender;
5. }

6. // Function to demonstrate require statement for input validation
7. function withdraw(uint256 amount) external {
8. require(msg.sender == owner, "Only the owner can withdraw");
9. require(amount > 0, "Withdrawal amount must be greater than zero");

10. // Transfer ETH to the caller
11. payable(msg.sender).transfer(amount);
12. }
13. // Function to demonstrate revert statement for custom error message
14. function divide(uint256 a, uint256 b) external pure returns (uint256) {
15. require(b != 0, "Division by zero error");
16. return a / b;
17. }
18. // Function to demonstrate assert statement for internal consistency checks
19. function assertExample(uint256 a, uint256 b) external pure returns (uint256) {
20. assert(a >= b); // Assertion fails if a is not greater than or equal to b

21. return a - b;
22. }
23. }
```

4.10. Unused code problem step 10

4.10.1. Dead code

In algorithm 15, dead code refers to sections of code that are essentially useless because they can never be executed. This often happens due to unreachable blocks or unused variables or functions. In the contract named *DeadCodeExample*, there's a function called *deadCodeFunction* (). Inside this function, there's a conditional statement that will always result in *false*. This means that the code inside this condition will never run, making it dead code. Additionally, the contract includes a variable named *deadCodeVariable* that is declared but never used elsewhere. This also qualifies as dead code. Identifying and removing dead code is important for keeping contracts clean and efficient. It helps reduce unnecessary gas consumption during deployment and execution, and makes the code easier to read and maintain for developer

Algorithm 15

```
1. contract DeadCodeExample {
2. address public owner;
3. uint256 public value;
4. constructor() {
5. owner = msg.sender;
6. }
7. // Function to set the value
8. function setValue(uint256 _newValue) external {
9. require(msg.sender == owner, "Only the owner can set the value");
10. value = _newValue;
11. }
12. // This function is dead code because it's unreachable
13. function deadCodeFunction() external {
14. // This condition will always evaluate to false
15. if (1 == 2) {
16. // This code block will never be executed
17. value = 100;
18. }
19. }
20. // This variable is dead code because it's declared but never used
21. uint256 public deadCodeVariable;
22. }
```

4.10.2. Opaque

In Algorithm 16, *OpaquePredicateExample*, an opaque predicate is demonstrated within the *opaquePredicate* function. Initially, the contract defines a utility function, *isPrime*, to determine whether a given number is prime. However, the key focus is on the *opaquePredicate* function, which employs an opaque predicate to obscure its behaviour. Within this function, two conditions are evaluated: whether the input number is prime (*isInputPrime*) and whether its last digit is 7 ((*_input % 10 == 7*)). However, the

function's return statement only yields *true* if both conditions are met simultaneously. This combination of conditions creates an opaque predicate, as the outcome of the function becomes less predictable without knowing the input. While such obfuscation techniques can enhance security by making code analysis and reverse engineering more challenging, excessive use of opaque predicates can complicate code maintenance and introduce potential vulnerabilities.

Algorithm 16

```
1. contract OpaquePredicateExample {
2. // Function to determine if a given number is prime
3. function isPrime(uint256 _number) public pure returns (bool) {
4. if (_number <= 1) {
5. return false;
6. }
7. if (_number <= 3) {
8. return true;
9. }
10. if (_number % 2 == 0 || _number % 3 == 0) {
11. return false;
12. }
13. for (uint256 i = 5; i * i <= _number; i += 6) {
14. if (_number % i == 0 || _number % (i + 2) == 0) {
        a. return false;
15. }
16. }
17. return true;
18. }

19. // Function demonstrating an opaque predicate
20. function opaquePredicate(uint256 _input) public pure returns (bool) {
21. // Check if the input number is prime
22. bool isInputPrime = isPrime(_input);

23. // Opaque predicate: Return true only if input is prime AND its last digit is 7
24. return isInputPrime && (_input % 10 == 7);
25. }
26. }
```

4.11. Unused variables step 11

Unnecessary or unused variables refer to variables within smart contracts or functions that don't serve any meaningful purpose or contribute to the code's functionality. These variables can lead to bloated code, higher gas costs, and confusion for developers. During the initial deployment of the smart contract, an unnecessary variable called `owner` was introduced in Algorithm 17. This resulted in a gas consumption of 541,774

units, reflecting the additional cost incurred due to the unused variable. Recognizing this redundancy, an optimization was performed by removing the `*owner*` variable in algorithm 18. As a result, the gas consumption during deployment decreased to 492,784 units. This reduction in gas usage demonstrates the impact of eliminating unnecessary variables on the efficiency and cost-effectiveness of smart contract deployment. By simplifying the code and removing extraneous components like the `owner` variable, developers not only improve readability and maintainability but also reduce unnecessary expenses associated with deploying and executing smart contracts on the Ethereum blockchain

Algorithm 17 - Unused variable present

```
1. contract MyToken {
2. string public name;
3. string public symbol;
4. uint8 public decimals;
5. uint256 public totalSupply;

6. // Unnecessary variable
7. address public owner;

8. mapping(address => uint256) public balanceOf;

9. constructor(string memory _name, string memory _symbol, uint8 _decimals, uint256 _totalSupply)
   {
10. name = _name;
11. symbol = _symbol;
12. decimals = _decimals;
13. totalSupply = _totalSupply;
14. balanceOf[msg.sender] = _totalSupply;

15. // Unnecessary variable assignment
16. owner = msg.sender;
17. }

18. function transfer(address _to, uint256 _value) public returns (bool success) {
19. require(balanceOf[msg.sender] >= _value);
20. require(balanceOf[_to] + _value >= balanceOf[_to]);

21. balanceOf[msg.sender] -= _value;
22. balanceOf[_to] += _value;

23. return true;
24. }
25. }
```

Algorithm 18 - unused variable removed

```
1. contract MyToken {
2. string public name;
3. string public symbol;
4. uint8 public decimals;
5. uint256 public totalSupply;

6. mapping(address => uint256) public balanceOf;

7. constructor(string memory _name, string memory _symbol, uint8 _decimals, uint256 _totalSupply)
   {
8. name = _name;
9. symbol = _symbol;
10. decimals = _decimals;
11. totalSupply = _totalSupply;
12. balanceOf[msg.sender] = _totalSupply;
13. }

14. function transfer(address _to, uint256 _value) public returns (bool success) {
15. require(balanceOf[msg.sender] >= _value);
16. require(balanceOf[_to] + _value >= balanceOf[_to]);

17. balanceOf[msg.sender] -= _value;
18. balanceOf[_to] += _value;

19. return true;
20. }
21. }
```

4.12. Optimization enabling step 12

Where even small improvements can significantly impact the cost of implementing smart contracts, it is important to apply compiler optimization to the source code compilation process This optimization should be coupled with the development environment used function for compilation optimization, such as the Remix IDE [16] However, it is important to note that this optimizer feature may not be automatically enabled each time it is compiled, so manual installation is required for optimal performance.

The source code for the methodology using solidity is share in GitHub[1].

[1] https://github.com/RahulTharammal/Smart-Contract-Execution-Costs-Techniques-for-Cost-Optimization-on-Blockchain-Networks

Table II gives examples of how smart contract source code can be improved by following the guidelines outlined in this paper. These improvements are explained through steps 1 to 12.

Table.2 Difference between enhanced and not enhanced gas cost with recommend contract structure

Improvement content	**Enhanced gas**	**Not Enhanced gas**	**%**
Data type variable	uint256 = 84,333 Boolean = 82527 Byte32 = 73,921	uint16 = 85,533 Boolean = 101745 String =358,191	33.23
Variable packing	Order of setting variables: bool = 82527 uint256 = 84,333 uint8 = 43761	Order of setting variables: uint256 = 152548 uint8 = 85533 bool = 101745	38.11
Memory and storage	Transaction from memory = 195036	Transaction from storage = 200612	2.78
Constant	Declared with constant = 35085	Declared without constant = 71288	50.7
Mapping and array	Function with array = 145188	Function without array = 228467	36.42
Loops	Transaction cost = 180847	Transaction cost = 231041	21.71
Short-Circuit	Execution cost = 168585	Execution cost = 170148	0.9189
Error handling	Execution cost = 344925	Execution cost = 354833	2.79
Unused code	Transaction cost = 152011	Transaction cost = 172154	11.69
Unused variable	Execution cost = 492784	Execution cost = 541774	9.94

5. Conclusions

This study focused on creating more efficient smart contract source code, particularly targeting areas that often lead to high gas costs, such as loops and data types. The main goal was to provide developers with a structured approach to building smart contracts. By examining previous research on gas usage in smart contract publishing and execution, it's clear that the efficiency of development largely depends on developers' choices. This paper aims to offer guidelines for creating efficient and cost-effective smart contracts. It emphasizes the complexity of programming smart contracts compared to traditional languages and stresses the importance of careful analysis and optimization techniques to reduce execution costs.

References

[1]. “Ethereum development documentation | ethereum.org,” ethereum.org. https://ethereum.org/en/developers/docs/ (accessed Mar. 10,2024).

[2]. “Introduction to Smart Contracts — Solidity 0.8.11 documentation.” https://docs.soliditylang.org/en/v0.8.11/introduction-to-smart-contracts.html (accessed Mar. 10,2024).

[3]. T. Chen, X. Li, L. Wang, and X. Zhang, Under-optimized smart contracts devour your money. 2017. doi: 10.1109/saner.2017.7884650.

[4]. T. Chen et al., “GasChecker: Scalable analysis for discovering Gas-Inefficient smart contracts,” IEEE Transactions on Emerging Topics in Computing, vol. 9, no. 3, pp. 1433–1448, Jul. 2021, doi: 10.1109/tetc.2020.2979019.

[5]. E. Albert, J. Fernández, P. Gordillo, G. Román-Díez, and A. Rubio, “GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts,” in Lecture Notes in Computer Science, 2020, pp. 118–125. doi: 10.1007/978-3-030-45237-7_7.

[6]. T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski, “Characterizing efficiency optimizations in solidity smart contracts,” 2020. https://www.semanticscholar.org/paper/Characterizing-Efficiency-Optimizations-in-Solidity-Brandst%C3%A4tter-Schulte/4996b77267bdac740cca27067b6af286e786a252

[7]. “Corda | The open permissioned distributed application platform,” Corda, Sep. 25, 2023. https://corda.net/ (accessed Mar. 10,2024).

[8]. [“Consensys - A complete suite of trusted products to build anything in web3,” Consensys. https://consensys.io/ (accessed Mar. 10,2024).

[9]. “Szabo, N. (1997) Formalizing and Securing Relationships on Public Networks. First Monday, 2, 1 September 1997. - References - Scientific Research Publishing.” https://www.scirp.org/reference/referencespapers?referenceid=1522947

[10]. “Web3, aggregated.” https://polygon.technology/ (accessed Mar. 10,2024)

[11]. L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, “Design patterns for gas optimization in Ethereum,” 2020. https://www.semanticscholar.org/paper/Design-Patterns-for-Gas-Optimization-in-Ethereum-Marchesi-Marchesi/7640ec77712479d9e9214bc5ccc1642786b9ea2a

[12]. “Solidity by Example — Solidity 0.8.25 documentation.” https://docs.soliditylang.org/en/latest/solidity-by-example.html (accessed Mar. 10,2024)

[13]. “ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER,” 2019. https://www.semanticscholar.org/paper/ETHEREUM%3A-A-SECURE-DECENTRALISED-GENERALISED-LEDGER/da082d8dcb56ade3c632428bfccb88ded0493214

[14]. C. Li, S. Nie, Y. Cao, Y. Yu, and Z. Hu, “Dynamic gas estimation of loops using machine learning,” in Communications in computer and information science, 2020, pp. 428–441. doi: 10.1007/978-981-15-9213-3_34.

[15]. “Expressions and Control Structures — Solidity 0.8.25 documentation.” https://docs.soliditylang.org/en/latest/control-structures.html (accessed Mar. 10,2024)

[16]. “Remix - Ethereum IDE.” https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null (accessed Mar. 10,2024)

[17]. “Contracts — Solidity 0.8.11 documentation.” https://docs.soliditylang.org/en/v0.8.11/contracts.html (accessed Mar. 10,2024)