VS Code Extension to Save Gas Costs in Solidity

Final Report

Wilson and Kyle Wu
301350197
301343359

Table of Contents

Note:
Since we were required to answer the same questions from our project proposal, we have included citation markers so that each question can be found throughout our report. We chose to keep the format listed above in our table of contents but still answered each question scattered in the various sections.

For example:
Question 1 states: *A clearly stated motivation. What is the problem that you are attacking? Why do you believe that this problem matters? What does addressing this problem accomplish?*

Our answer to this question can be found in the sections that have the citation marker *[Q1]*.

I. Problem/Motivation Introduction [Q1, Q2, Q3]

Deploying and interacting with Solidity smart contracts on the Ethereum blockchain costs gas fees that are equivalent to a real world amount of money.

There are three variables that can impact the deployment and usage costs of a smart contract: the cost of the blockchain's native token, how congested the blockchain network is at the time of interaction, and the code of the contract. Since the first two variables are unlikely to be controllable factors, the only reasonable option is to optimize the contract code. For our project, we have decided to optimize Solidity smart contract code through the method of *state variable packing* in order to reduce deployment and interaction costs.

There are several optimization tools such as Slither which performs static analysis in order to optimize and secure smart contracts. However, there are currently no existing tools that perform automatic variable packing optimizations as we have attempted to accomplish in this project.

II. Methodology Introduction [Q4, Q6]

We chose to implement this as a Visual Studio Code extension. Many developers already use VS Code as their editor of choice, and VS Code extensions allow the use of our project to be as easy as possible. The users are able to optimize their contracts right in the editor without having to copy and paste it out into another tool or website

Each Solidity contract is different so there is no one-size-fits-all approach to variable packing. Taking this into consideration, our VS Code extension, Solidity-Gas-Saver, has five different algorithms/strategies for variable packing that the users can choose from. The README in the repository provides an excellent introduction to the extension as well as the strategies.

In general, our extension will first generate an abstract syntax tree for any Solidity contract. From the AST it will keep track of the state variables and their sizes, as well as the order and number of occurrences for each state variable used in a function. Depending on which variable packing strategy is chosen, the extension will then generate a new order for the variables and automatically rearrange it inside the code for the user.

In this report we aim to evaluate the extension as a whole as well as the advantages and disadvantages of each strategy by conducting real world tests on popular contracts.

If you like to run the extension or take a look at our code, please visit the repository found here. The README contains instructions on how to set up and run the extension. We have also included test contracts set up already found here.

III. Ethereum Gas Costs [Q5]

Interacting with the Ethereum blockchain, such as transferring Ether (ETH) or using a smart contract to conduct a swap between two tokens all cost an amount of gas in order for the transaction to be included in the blockchain. At a high-level, the less complex a function call is and the fewer state variables used, the cheaper the overall transaction will be. The amount of gas paid per transaction is calculated by: *units of gas used * (base fee + priority fee)*. The units of gas used is the amount of computational effort required to execute a transaction. This is determined by the number of storage accesses and opcodes used in the transaction. For the purposes of this project, we will only consider the *units of gas used* in order to optimize a smart contract.

There are two costs of a smart contract that can be optimized

- Deployment Cost: A one-time cost of creating a smart contract and deploying it on the Ethereum blockchain. The total number of storage slots declared affect the gas cost of deployment.

- Function Call Costs: The cost of interacting with the deployed smart contract by executing one of the public functions. The number of variables, and therefore storage slots read or written to in a function call affects the gas cost.

The state variables we will optimize can be the following types with the sizes in bits/bytes:

- `uint` - Unsigned Integer of size 32 bytes
- `int` - Signed Integer of size 32 bytes
- `uintX` - Unsigned Integer of size X, ranging from 1 to 256 bits divisible by 8
- `intX` - Signed Integer of size X, ranging from 1 to 256 bits divisible by 8
- `bytesX` - Bytes of size X, ranging from 1 to 32 bytes
- `address` - Ethereum address of 42 hexadecimals, of size 20 bytes
- `bool` - Boolean value of size 1 byte

The developer can choose to declare these variables in any particular order. However, the size and arrangement of those variables will affect the storage used by the contract which will impact the gas cost of deployment and function calls.

In Ethereum, each storage slot is 256 bits or 32 bytes long. If a variable being packed exceeds the 256 bit limit, it is stored in a new storage slot.

Example 1- Multiple Storage Slots) The following two variables will use a single storage slot since they sum to 128 + 128 = 256 bits. If a function only needs to use `var2`, the transaction would unnecessarily load in `var3`, since `var2` and `var3` are stored in the same storage slot.

```
uint128 var2;
uint128 var3;
```

In addition, if variables are not declared in the order which maximizes these 256 bit slots, then more storage slots will be used. This will increase the cost of deploying the smart contract on the blockchain.

Example 2 - Unused Storage Slots) The following three variables will use three storage slots, one for each variable. If the variables in a storage slot do not occupy the entire 256 bits, the remaining space will be unused but affect the gas costs. In this case, `var5` could be changed to type `uint256` at no extra cost.

```
uint256 var4;
uint128 var5;
uint256 var6;
```

Example 3 - Optimizing Slot Layout) The following three variables will use three storage slots, one for each variable.

```
uint128 var7;
uint256 var8;
uint128 var9;
```

However, the following optimization will only occupy two storage slots instead, since `var7` and `var9` can be packed into a single storage slot of 256 bits.

```
uint128 var7;
uint128 var9;
uint256 var8;
```

The combination of minimizing the number of total storage slots during contract deployment and the storage slots used during a function call is known as the variable packing problem. Our project aims to automatically reorder the state variables such that the least amount of storage slots are created and used, which we hope will lower the gas costs.

IV. Optimization Methods [Q5]

Since there are two costs to consider, there are potential tradeoffs between optimizing deployment cost and function call costs. Optimizing solely in terms of the deployment cost may negatively affect the cost of function calls and vice versa. A greedy contract deployer may optimize in terms of deployment cost such that they spend less when deploying the contract, but this may negatively affect users as they incur more costs while interacting with the contract. Therefore, the contract developer must take into consideration what the purpose of the contract is for.

If a contract's functions are not meant to be called often, such as a one-time wager contract, then the deployment cost should likely be optimized. If a contract's functions are meant to be called often, such as an NFT marketplace, the function calling costs should likely be optimized. We created five different types of variable packing strategies in an attempt to optimize these two gas costs.

These are the five variable packing strategies, but the full explanation for each one can be found in our README:

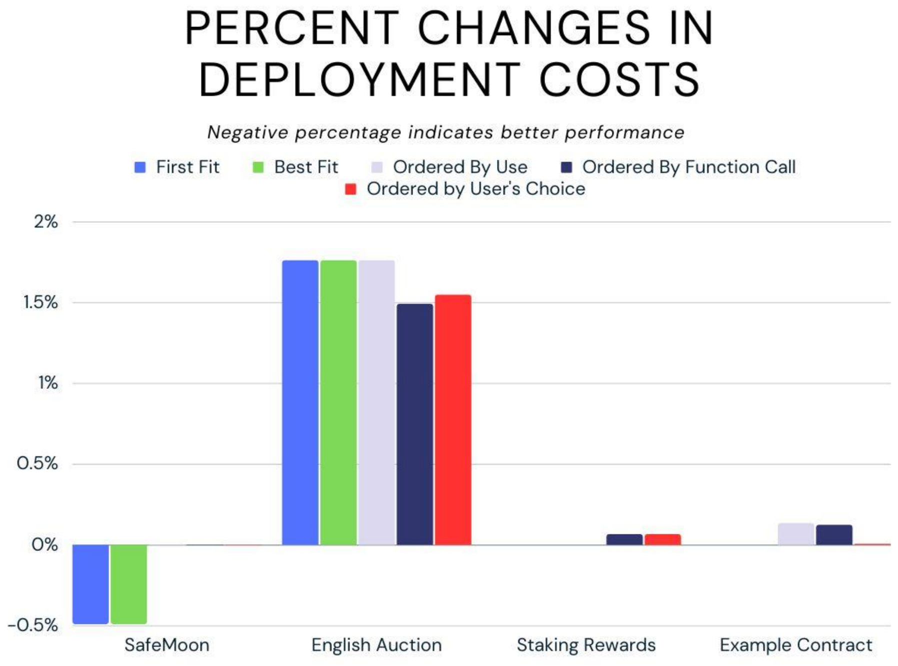
1. Bin packing: First fit (Pack Variables With First Fit)
2. Bin packing: Best fit (Pack Variables Best With Fit)
3. Packing by order of use in functions (Pack Variables By Use)
4. Grouped by function call frequency (Pack Variables By Function)
5. Grouped by functions from user input (Pack Variables By User Input Function)

We expect algorithms one and two to optimize deployment costs over function call costs because state variable usage in functions is not considered in the algorithms. On the contrary, we expect algorithm three, four, and five to optimize function call costs over deployment costs because state variable usage in functions is being considered in the algorithms.

V. Evidence and Interpretation from Comparing Gas Costs of Deploying Contracts [Q5]

Our five strategies were applied to the following smart contracts. The original contracts were deployed along with the deployment of contracts utilizing each strategy.

- SafeMoon - A popular reflection token. The contract charges a fee for transferring SafeMoon tokens, which are redistributed/reflected back to token holders
- English Auction - An non-fungible token (NFT) auction smart contract, where users can bid on a NFT. At the end of the auction, the highest bidder receives the NFT.
- Staking Rewards - A contract that distributes reward tokens over time to users who stake tokens in the contract.
- Example Contract - A worst-case scenario contract that demonstrates the potential function call cost reduction the optimizations can achieve.



The graph displays the deployment costs of the optimized contracts relative to the original un-optimized contract, where a negative percentage indicates a saving of gas costs.

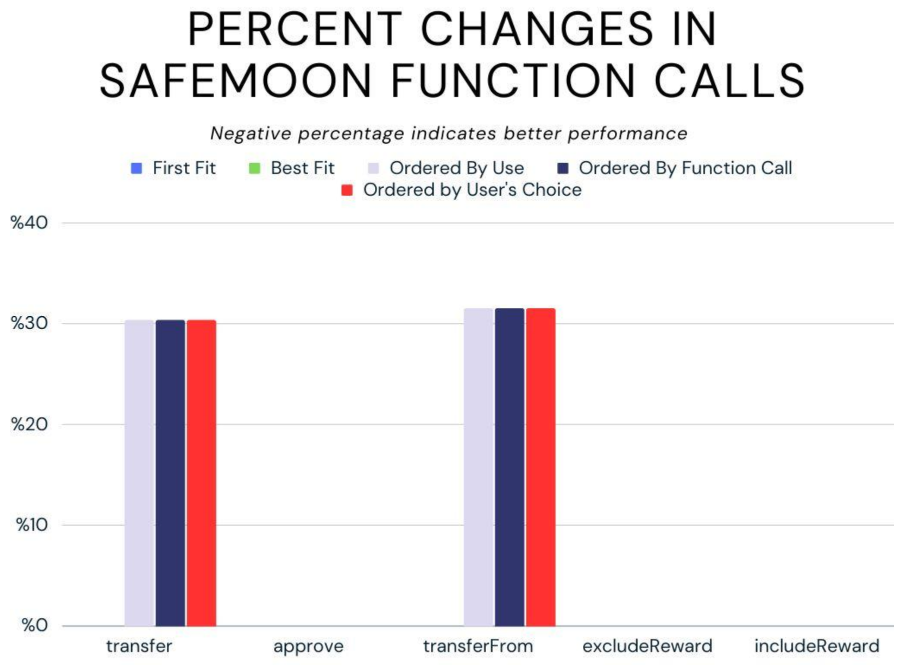
In the SafeMoon contract, the First Fit and Best Fit strategies decrease the gas costs, proving our strategies to be successful in reducing deployment costs in some contracts.

In the English Auction contract, the results show a deployment cost increase in all of the rearranging strategies. This may have been caused by data types or scenarios that were considered out of scope for this project (see VII: Reflection: Out of scope), and therefore were not rearranged by the algorithms.

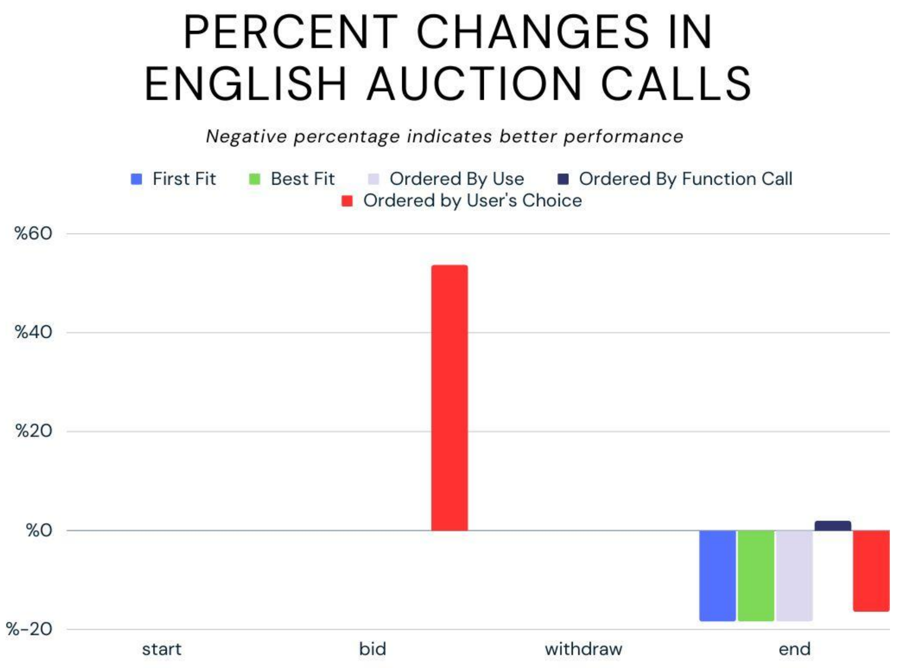
As expected, the strategies (three, four and five) that consider state variable usage in function calls have an increase in deployment costs due to the neglect of minimizing the number of storage slots.

VI. Evidence and Interpretation from Comparing Gas Costs of Function Calls [Q5]

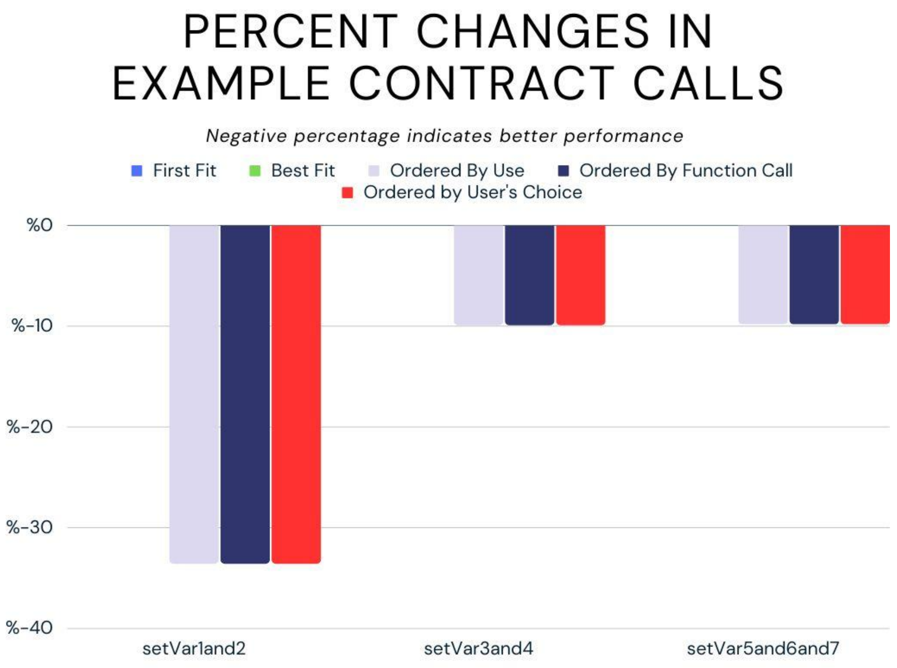
Next, we compared the function call costs. The three graphs display the function call costs of the optimized contracts relative to the original un-optimized contract, where a negative percentage indicates a saving of gas costs.



In the SafeMoon contract, the First Fit and Best Fit strategies do not change the costs of any of the main function calls, proving our strategies can reduce both the function cost and deployment cost. The three strategies expected to reduce function call costs (three, four, and five) have increased costs to call function `transfer()` and `transferFrom()` by around 30%. We suspect the main culprit is the variable `bool inSwapAndLiquify`, which was already optimally packed in the original contract. The three strategies have attempted to pack the variable in a storage slot by itself or with an irrelevant variable, increasing the execution cost.



In the English Auction contract, almost every strategy had decreased the cost of calling function `end()` by around 20%, while also maintaining a similar or same deployment cost in strategies one, two, and five. An anomaly can be seen with strategy five with function bid(). This difference can be explained by the fifth algorithm's lack of minimizing the number of storage slots within the grouping of the state variables used in `bid()`.



In the example contract, we attempted to showcase how a very inefficient contract can be optimized through our strategies. Our results show at most a 33% cost reduction in the three functions by the last three strategies. The optimized contracts loaded in fewer storage slots, and did not read unnecessary values. Again, we do not see a difference in function call costs by the first and second strategy since the algorithms were focused on reducing only deployment costs.

VII. Reflection [Q2, Q4, Q6, Q7]

Were we successful?
We believe we were successful in either strictly reducing deployment costs or function call costs. But we were unable to reduce both costs simultaneously due to the tradeoffs between the two costs. Between the two types of algorithms (optimizing deployment cost versus optimizing function call costs), there was no clear best strategy as they performed similarly in their respective groups. In the end, we were able to successfully reach the goal we set in our initial project proposal of reducing gas costs.

What we learned?
At first, we had a general idea for what we wanted to achieve in this project, and came up with many different ideas that would help us achieve the goal of optimizing Solidity contracts. Based on the excellent feedback we received for our initial proposal, we were able to see that the project we were proposing was not refined enough. We had initially failed to focus on one specific problem to solve and we may have been trying to achieve more than what was realistic in our given timeframe. Through various meetings with the professor and repeating the cycle of research and iterating, we were finally able to refine the problem statement and methodology in a realistic yet challenging project. This taught us the process and importance of planning and scoping when we are designing a project.

Although we knew we wanted to develop a VS Code extension, we did not have any experience and had to learn from scratch. Following tutorials, looking at documentation, and just trying to fail quickly was our plan for completing this project as a VS Code extension. This approach was effective for us and our existing knowledge of JavaScript was very beneficial in learning TypeScript for the first time. We learned to effectively use all the basics in the VS Code extension API such as registering commands, editors, documents, input boxes, edit builders, and many more. From implementing this project we have gained more confidence in VS Code extension development.

Our first idea for implementing the variable packing was to manually iterate through each line in the contracts and parse for state variables, the types and sizes of each variable, functions, and variables used inside each function. We used regex for this. Although we did not end up sticking with this manual method of parsing utilizing regex, it was not a wasted effort as we got to improve our regex.

Instead of manually parsing each line, we learned of the concept of abstract syntax trees from the course material as well as meetings with the professor. Abstract syntax trees is a powerful tool that would not only automatically parse the contract into an easy-to-access object, it would also ensure the contract is compilable. Learning to use the AST package *solc-typed-ast* was difficult at first as traversing the various nodes was initially overwhelming. After we had gotten used to the structure of the AST it was valuable for us to then convert our existing code to leverage the AST instead of parsing each line in the contract manually.

Through completing this project, we were able to learn more about the inner workings of the Solidity coding language. Specifically, we became familiar with how storage slots, gas costs, and optimization is handled within the Ethereum blockchain. Not only did we have to consider optimizing deployment costs, but also how it can negatively impact function call costs and vice versa. Additionally, we learned which projects can and cannot benefit from such optimizations through our research and testing on several smart contracts. For example, multiple functions that use the same variables while also separately using different variables is a difficult scenario to optimize.

Out of Scope

Our initial proposal was too optimistic, unfocussed and we mentioned a lot of additional features with the extension that ended up becoming out of scope. These include interval analysis and shrinking sizes of immutable/constant state variables. These two features became out of scope as we decided to just focus on state variable packing.

In addition, there are some assumptions that our extension makes about the user's contracts and the full list can be found here in our README. These are rules that the users should follow when using our extension such as having only one contract per file or having a contract that successfully compiles. All of these assumptions can be considered as features that are out of scope for our extension.

In addition, the README contains types that will not be specifically handled and will just be assumed to be 256 bits.

Did we hit our milestones?

Yes. Although we did a lot of refining to our project after our initial project proposal, we were able to stay on track with development and for gathering feedback for our ideas and overall direction. We met with the professor when we planned to do so, and finished development early on so that we could spend adequate time testing and evaluating the extension.

Next steps/Future directions

For our next steps, we should handle more cases with the contracts so that there are less assumptions and the usability of the extension improves. This includes adding structs, custom address types, and function modifiers to our optimization methods. Afterwards, we could implement the two features discussed in our proposal which were interval analysis, and shrinking sizes of immutable/constant state variables.