Lesson 8 - Gas Optimisation

Gas Optimisation Introduction

“The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming. - Donald Knuth

Optimisation Process

Be clear what circumstances you are optimising for, and decide upon an acceptable level of performance. You may have to make trade offs between different scenarios.
The correctness of the code is always the priority.

Assuming your code is correct, and you have a suite of unit tests to confirm that.

Optimisation steps

1. Measure the performance (context is important here)
2. Pick the change that will have the most impact
3. Refactor the code
4. Confirm the code correctness using the unit tests
5. Go to 1

Basic Techniques

Remove redundant functionality and code

One of the easiest gains is to remove functionality that is excessive, or no longer needed. Similarly removing redundant code will reduce deployment cost.

Take shortcuts

1. Break out of loops as early as possible
2. For 2 functions as follows
   `a(x)` is low cost
   `b(x)` is expensive

   The ordering should be

```
    a(x) || b(x)
    a(x) && b(x)
```

Storage

Since the SSTORE operation is one of the most expensive, there is significant potential for gas savings when using storage.

Saving one slot that is a word of 256 bits to Storage (SSTORE) is 22,100/20,000 gas when you initially set it from zero to non-zero, depending on whether it is an initial access.

Storage slots are either warm or cold.

- Cold: storage slot hasn't been accessed during this transaction.
- Warm: storage slot has been accessed during this transaction.

5,000/2,900 gas is spent when an already used Storage slot is rewritten.
Reading a storage slot using SLOAD takes 100 gas.

Consider

1. Storing data elsewhere
2. Not storing intermediate steps in storage, just the final result.

Memory versus Storage

Memory is generally cheaper than storage but

Copying between the memory and storage will cost some gas, so don't copy arrays from storage to memory, use a storage pointer
(but beware of subtle bugs when doing this, a Defi project was recently rekt by this)

Obviously some data needs to persist between function calls
The cost of memory is complicated, you "buy" it in chunks, the cost of which will go up

quadratically after the first 724 bytes.

Avoid repetitive checks

By being sure of the logic of your code, you may be able to remove redundant checks, having a test suite with good coverage will help this.

Refunds

Free Storage slots by zeroing corresponding variables as soon as you don't need them anymore. This will refund 15000 gas.

Some opcodes can trigger gas refunds, which reduces the gas cost of a transaction.
However the gas refund is applied at the end of a transaction, meaning that a transaction always need enough gas to run as if there was no refunds.
The amount of gas that can be refunded is also limited, to half of the total transaction cost before the hardfork London, otherwise to a fifth.
Starting from the hardfork London also, only SSTORE may trigger refunds. Before that, SELFDESTRUCT could also trigger refunds.

Data Types and Packing

- Use `bytes32` whenever possible, because it is the most optimized storage type.
- Type `bytes` should be used over `byte[]`.
- If the length of bytes can be limited, use the lowest amount possible from `bytes1` to `bytes32`.

Strings

Using `bytes32` is cheaper than using the `string` type.

Mapping vs. Array

Most of the time it will be better to use a mapping instead of an array because of its cheaper operations.
However, an array can be the correct choice when using smaller data types. Array elements are packed like other storage variables and the reduced storage space can outweigh the cost of an array's more expensive operations. This is most useful when working with large arrays.

Packing variables

Pack several blocks of information into one Storage slot if they are smaller than a word of 32 bytes. This can give significant savings. Specifically if according to the application logic, they are usually updated and accessed together. For example, a structure of 2 uint128 can be stored in one slot in a mapping instead of storing them separately.

For example take this code

```
Struct Data {
uint64 a;
uint64 b;
uint128 c;
uint256 d;
}

Data public data

constructor (uint64 _a,uint64 _b, uint128 _c, uint256 _d) public {
    Data.a = _a;
    Data.b = _b;
    Data.c = _c;
    Data.d = _d;
}
```

The SStore instruction is performed twice, once to store a,b,c and a second time to store d (the solc optimiser is able to work out this optimisation)

```
[         a       ] [        b        ] [         c          ]
[8 bytes / 64 bits] [8 bytes / 64 bits] [16 bytes / 128 bits ]

[                                  d                         ]
[                        32 bytes / 256 bits                 ]
```

So generally is modifying a uint8 cheaper than modifying a uint256 ? ....

no...
storing a small number in a uint8 variable is not cheaper than storing it into a uint256 variable, because for storing, any smaller data is padded with zeros to fill the 32 bytes, requiring additional operations from the EVM and additional gas.

Storage and Inheritance

When we extend a contract, the variables in the child can be packed with the variables in the parent.
The order of variables is determined by C3 linearization. For most applications, all you need to know is that child variables come after parent variables.

Memory versus Storage

Memory is generally cheaper than storage but

Copying between the memory and storage will cost some gas, so don't copy arrays from storage to memory, use a storage pointer
(but beware of subtle bugs when doing this, a Defi project was recently rekt by this)

Obviously some data needs to persist between function calls
The cost of memory is .. complicated, you "buy" it in chunks, the cost of which will go up quadratically after a while. See lesson 3 notes.

Try adjusting the location of your variables playing with the keywords "storage" and "memory". Depending on the size and number of copying operations between Storage and Memory, switching to memory may or may not give improvements. All this is because of varying memory costs. So optimising here is not that obvious, and every case has to be considered individually.

Language Features

Variables

- Use events rather than storing data
- Avoid public variables
- It may be good to avoid using storage, by employing memory arrays. If the size of the array is exactly known, fixed size memory arrays can be used to save gas.
- A simple optimisation in Solidity consists of naming the return value of a function. It is not needed to create a local variable then.

Functions

Calling functions is relatively cheap (it is just a jump instruction), but it can degrade the compiler's attempts at storage optimisation.

Memory, calldata and function parameters

Storing the input parameters in memory costs gas. For all public functions, the input parameters are copied to memory automatically.
If a function is only called externally, it should be explicitly marked as external, in a way that these parameters are not stored into memory but are read from call data directly.
This can save gas when the function input parameters are huge.

Function order

See Function order article

Each position will have an extra 22 gas, so

- Reduce public variables
- Put often called functions earlier (the order depends on the hash of the function name)

Tool to optimise function name

Compress Input Data

See the example in Compress Input Data Article

they go from these function parameters

```
  uint256 amountSell,
uint256 amountBuy,
address tokenSell,
address tokenBuy,
address user,
uint256 nonce,
uint256 gasFee,
uint256 takerFee,
uint256 makerFee,
```

```
uint256 joyPrice,
bool isBuy,
uint8 v,
byte32 r,
byte32 s
```

to these

```
  uint256 amountSell,
uint256 amountBuy,
uint256 data,
uint256 gasFee,
byte32 r,
byte32 s
```

without losing functionality by packing many of the parameters in the data field

View Functions

You are not paying for view functions that aren't transactions. But this doesn't mean they aren't consuming gas, they do. It is just that it is free when executed on the local EVM. However, if a view function is called in a transaction, all the gas matters.

Modifiers

Modifiers Increase Code Size you can make them functions instead
When using modifiers, the code of the modifiers is inserted at the start of the function at compile time, which can massively increase code size.
So making a modifier a function call instead can help, as only the function call will be inserted at the start of the function.

Loops

Due to the expensive SLOAD and SSTORE opcodes, managing a variable in storage is much more expensive than managing variables in memory. For this reason, storage variables should not be used in loops.

For example

```language-none
uint num = 0;
function expensiveLoop(uint x) public {
  for(uint i = 0; i < x; i++) {
    num += 1;
  }
}
```

do this instead

```language-none
uint num = 0;
function lessExpensiveLoop(uint x) public {
  uint temp = num;
```

```
    for(uint i = 0; i < x; i++) {
      temp += 1;
    }
    num = temp;
  }
```

- Optimise loops to minimise the number and cost of instructions within the loop.
- Take unnecessary values out of the loop
- Predict values if possible
- Reduce the number of iterations by for example breaking out of loop as soon as possible
- Try to avoid unbounded loops

Custom Errors

Custom Errors Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., revert("Insufficient funds.");), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.
Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries).

language-js

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
error Unauthorized();
contract VendingMachine {
        address payable owner = payable(msg.sender);
        function withdraw() public {
                if (msg.sender != owner){
                        revert Unauthorized();
                }
        owner.transfer(address(this).balance);
        }
}
```

Events

Here's the formula for a LOG gas cost:

k + unindexedBytes *a* + *indexedTopics* b
where

k = 375
a = 8
b = 375

(Note also that if you use a bigger than 256 bit type for an indexed event topic, like bytes[1000] or something, then you still only pay 375, because in this case, only the Keccak hash of the value is actually indexed.)

Miscellaneous Optimisations

Use the optimiser

Make sure the optimiser is turned on when compiling your code, you can also adjust the number of runs it will use.

- Remove dead code
- Solidity version, later versions are probably better.
- Use optimization and set the counter to high values or leave the default 200. Setting it to 1 can be useful in a rare case when it's important to optimize contract deployment, but not subsequent functions call.
- Use Libraries (wisely)
  When a public function of a library is called, the bytecode of that function is not made part of a client contract. Thus, complex logic should be put in libraries for keeping the contract size small. But there is a cost for calling the library function.
- Require and Assert
  Use "require" for all runtime conditions validations that can't be prevalidated on the compile time. And "assert" should be used only for static validations that normally never fail in a properly functioning code.
  Reducing error messages text will decrease gas used by the function.

Hash functions
keccak256: 30 gas + 6 gas for each word of input data
sha256: 60 gas + 12 gas for each word of input data
ripemd160: 600 gas + 120 gas for each word of input data
So if you don't have specific reasons to select another hash function, just use keccak256

Tools and Measurement

Web3 :
`web3.eth.estimateGas(callObject [, callback])`
can estimate the gas required for a transaction

There are gas reporter plug ins for Hardhat and Foundry

Sol2UML Visualisation Tool

See repo
It provides visualisation of storage, plus UML diagrams for the contract.