Advanced Solidity: Understanding and Optimizing Gas cost

Gas cost are determined by four things: transaction data that was sent, amount of memory that was used, state changes that were executed and opcodes that were used.

Whenever you transfer Ethereum it always cost 21k gas. This never changes. However, the gas price varies (how much you pay the miners to get the transaction executed)

Transaction cost in dollars = ((Gas used by the trans x Gas price / 1 000 000 000)) * [price of ETH in dollar]

Gas = how many CPU cycles a certain computation requires

The same transaction on the same smart contract will not always have the same gas costs. The gas price increases when more people are using the blockchain.

Bitcoin historically has limited block size to 1 MB. Ethereum does not explicitly set a byte limit. Instead ETH limits the amount of computations per block (or gas). Why? If a transaction requires to much computation it could bring down the blockchain. At the time of this course, block limit is 30 million gas. ETH transfer cost 21k gas ⇒ one block in theory: 1 428 transactions.

A new block is generated every 15 seconds. In the extreme case of Tornado trans (1 million gas) ⇒ 2 trans per second (95 if standard ETH). In reality there are 13 tps at this time of recording.

Gas wars occur when more than 1428 trans are trying to hop on the same block (at least 21k per trans).

cost of storing a variable ~2100 gas

mstore looks at the top two items in the stack and stores the penultimate value in the slot indicated by.

5 places to save gas:

1. Deployment: the smaller the smart contract, the lass gas you pay.
2. During computation. Using fewer and/or cheaper OP-codes saves gas
3. Transaction data. The larger your transaction data, the more non-zero bytes you have in it, the more gas it costs
4. Memory. The more memory you allocate, even if you don’t use it, you pay more gas
5. Storage.

Payable functions are typically cheaper in gas than non-payable. By definition it will have fewer op-codes since a non-payable function must include OP-codes for revering if tokens are sent to the function.

In a lot of cases, you don’t want to put code in unchecked blocks. However, sometime it makes sense because you will save gas. Solidity 0.8.0> requires that a + b > a. Tip: put in unchecked when hunting for bugs (less OP-codes).

21, 000 comes from the blockchain has to “prepare” for the transaction

gas price per gwei ≤max_fee

BASEFEE = how much Gwei every transaction must burn in a certain block. If current block is full then next block is gonna have higher BASEFEE. Very roughly it increases by 12% and decreases by 12%.

BASEFEE = amount burned. Determined by network.

Max Fee = most you are willing to pay for the transaction independent of the priority fee. This is the upper bound of the gas price your trans. will pay.

Max Priority Fee = the most you are willing to give to miner.

Miner tip/priority fee = actual amount miner receives.

Smaller number of iterations in solidity optimizer = cheaper to deploy but more expensive for users in the long run

GAS COSTS

Setting storage 0 to >0 → 20 000 gas

Setting storage non-zero to non-zero → 5 000 gas

Setting storage from non-zero to zero → refund

pay additional 2 100 gas if first time accessing a variable in a transaction

pay additional 100 gas if the variable has already been touched

Doing a storage read and the a storage write costs similar to only doing a storage write.

The read + write case is :

2,100 (read + cold access) + 20,100 (write + warm access) = 22,200

Doing a write without read : 22, 100

cost(read + write) ~~ cost(write)

Ethereum always treats uint as 32 bytes. No matter 256 or 8 or whatever.

1 → 0 will save you gas. EVM wants to reward you for setting state variables to zero.

I. Setting to zero can cost between 200 and 5000 gas.

II. Deleting an array (or setting many values to zero) can be expensive. Beware of the 20% RULE

III. Setting one variable to zero is OK. Setting several to zero is like doing several non-zero to non-zero operations.

IIII. Counting down is more efficient than counting up.

Storing large arrays of memory will be linear in terms of gas but at a certain point it will blow up exponentially.

<AND> are always for efficient than ≤AND≥

because ≤AND≥ requires two OP codes: LT/GT and ISZERO

<AND> only requires one opcode: LT or GT

It-s often cheaper to revert earlier than later

X && Y : if X is false, Y will never be evaluated. Put the cheaper one first. Example:

require(block.timestamp > 1649604154 || allowed[msg.sender], “invalid”)