Solidity Best Practices / Tips & Tricks

Solidity Gas golf

https://g.solidity.cc/

Third Party Lists

Consensys Best Practices

GENERAL

- Prepare for Failure
- Stay up to Date
- Keep it Simple
- Rolling out
- Blockchain Properties
- Simplicity vs. Complexity

PRECAUTIONS

- General
- Upgradeability
- Circuit Breakers
- Speed Bumps
- Rate Limiting
- Deployment
- Safe Haven

SOLIDITY SPECIFIC

- Assert, Require, Revert
- Modifiers as Guards
- Integer Division
- Abstract vs Interfaces
- Fallback Functions
- Payability
- Visibility
- Locking Pragmas
- Event Monitoring
- Shadowing
- tx.origin
- Timestamp Dependence
- Complex Inheritance

- Interface Types
- EXTCODESIZE Checks

TOKEN SPECIFIC

- Standardization
- Frontrunning
- Zero Address
- Contract Address

DOCUMENTATION

- General
- Specification
- Status
- Procedures
- Known Issues
- History
- Contact

ATTACKS

- Reentrancy
- Oracle Manipulation
- Frontrunning
- Timestamp Dependence
- Insecure Arithmetic
- Denial of Service
- Griefing
- Force Feeding

From @controlcthenv

Dont Initialise Zero Values

Before Using Yul, Verify YOUR assembly is better than the compiler's

Overwrite new values onto old ones you're not using when Possible

Solidity doesn't garbage collect, and this is just cheaper in Yul, but write new values onto old unused ones to conserve memory and storage used saving gas!

Keep Data in Calldata where possible

Since Calldata has already been paid for with the transaction, if you don't modify a parameter to a function, then don't bother copying the function to memory and just read

the value from calldata.

View Solidity Compiler Yul Output

If you want to see what your Solidity is doing under the hood, just add -yul and -ir to your solc compiler options. Very helpful to see how your code is working, see if you order operations unsafely, or just see how Solidity is beating your Yul in gas usage.

Solc Compiler Options

Using Vanity Addresses with lots of leading zeroes

Why? Well if you have 2 addresses - 0x000000a4323... and 0x0000000000f38210 because of the leading zeroes you can pack them both into the same storage slot, then just prepend the necessary amount of zeroes when using them. This saves you storage when doing things such as checking the owner of a contract.

Using Sub 32 Byte values doesn't always save gas

Sub 32 byte values can save gas in the event of packing, but note that they require extra gas to decode and should be used on a case-by-case basis.

Writing to an existing Storage Slot is cheaper than using a new one

Credit - @libevm

EIP - 2200 changed a lot with gas, and now if you hold 1 Wei of a token it's cheaper to use the token than if you hold 0. There is a lot to unpack here so just google EIP 2200 and learn if you want, but in general, if you need to use a storage slot, don't empty it if you plan to re-fill it later. Goes for all Yul+ and Yul contracts when managing memory.

Tip - Int's are more expensive the leading bits

Credit - @transmissions11

Explained here

Using iszero() in a lot of places because the compiler is smart

Credit - @transmissions11

He explains it very well here but because the compiler knows how to optimize, putting it before some pieces of logic can end up reducing overall gas costs, so test out inserting it before JUMP opcodes.

`if iszero(x)` is actually cheaper than `if x` because of how JUMPI works!

For more info: `if (X > 0)` is translated into `iszero(gt(X, 0))`. The optimizer translates that into `iszero(iszero(iszero(X))`

Also see here

Use Gas() when using call() in Yul

Credit - @libevm

When using call() in Yul you can avoid manually counting out all the gas you need to perform the call, and just forward all available gas via using gas() for the gas parameter.

Store Storage in Code

Credit - @boredGenius

So Zefram's blog explains this well, but you can save gas by deploying what you want to store in a new contract, and deploying that contract, then reading from that address. This adds a lot of complexity to code but if you need to cut costs and use SLOAD a lot, I recommend looking into SLOAD2.

Half of the Zero Address Checks in the NFT spec aren't necessary

Credit - @transmissions11

Launching a new NFT collection and looking to cut minting and usage costs? Use Solmate's NFT contracts, as the standard needlessly prevents transfers to the void, unless someone can call a contract from the 0 address, and the 0 address has unique permissions, you don't need to check that the caller isn't the 0 address 90% of the time.

If it can't overflow without uint256(-1) calls, you don't need to check for overflow

Save gas and avoid safemath with unchecked {} , this one is Soldity only but I wanted to include it, I was tired of seeing counters using Safemath, it is cost-prohibitive enough to call a contract billions of times to prevent an attack.

If you are testing in Production, use Self-Destruct and Factory patterns for an Upgradeable contract

Credit - @libevm

Using a technique explained in this Twitter thread you can make it easily upgradeable and testable contracts with re-init and self-destruct. This mostly applied to MEV but if you are doing some cool factory-based programming it's worth trying out.

Fallback Function Calls are cheaper than regular function calls

The Fallback function (and Sig-less functions in Yul) save gas because they don't require a function signature to call, for an example implementation I recommend looking at @libevm's subway which utilize's a sig-less function

Pack Structs in Solidity

Struct packing article here

A basic optimization but important to know, structs should be organized so that they sequentially add up to multiples of 256 bits in size. So uint112 uint112 uint256 vs uint112 uint256 uint112

Saves read operations needed to get a value

Making Solidity Values Constant Where Possible

They are replaced with literals at compile time and will prevent you from having to a read a value from memory. For writing Yul - replace all known values and constant values with literals to save gas and comment what they are.

Solidity Modifiers Increase Code Size, So sometimes make them functions

Credit - The Smart Contract Programmer on Youtube

When using modifiers, the code of the modifiers is inserted at the start of the function at compile time, which can massively ballon code size. So sometimes it makes sense to make a modifier a function call instead, as only the function call will be inserted at the start of the function.

Trustless calls from L2 to L2 exist, and can be very useful for L2 based DAO's

Credit - Optimism and Arbitrum teams

The OVM and ArbOS have built-in functions on contract calls from L1 to L2 to verify msg.sender and vice versa. Therefore if you make an L1 contract that can only be called by a trusted party on one L2 before calling another L2, you can create a trustless bridge. Recommend reading about Hop for this, but a cool design choice for DAO building.

Low level call example

```
    assembly {
        let x := mload(0x40)   //Find empty storage location using "free
memory pointer"
        mstore(x,sig) //Place signature at begining of empty storage
        mstore(add(x,0x04),a) //Place first argument directly next to
signature
        mstore(add(x,0x24),b) //Place second argument next to first, padded
to 32 bytes

        let success := call(      //This is the critical change (Pop the
top stack value)
                            5000, //5k gas
                            addr, //To addr
```

```
                                0,    //No value
                                x,    /Inputs are stored at location x
                                0x44, //Inputs are 68 bytes long
                                x,    //Store output over input (saves space)
                                0x20) //Outputs are 32 bytes long

        c := mload(x) //Assign output value to c
        mstore(0x40,add(x,0x44)) // Set storage pointer to empty space
    }
```

Further gas optimisation tips

Use bitmaps, From bitmaps

For example, rather than

```
// people who showed up: 1, 3, 4, 8, 9
uint8[] memory a = new uint8[](1, 0, 1, 1, 0, 0, 0, 1, 1, 0);
```

we can do

```
// people who showed up: 1, 3, 4, 8, 9
uint16 a = 397; // equals 0110001101 in binary
```

We can read the bits with

```
uint16 a = 397; // equals 0110001101 in binary

// Read bits at positions 3 and 7.
// Note that bits are 0-indexed, thus bit 1 is at position 0, bit 2 is at
position 1, etc.
uint8 bit3 = a & (1 << 2)
uint8 bit7 = a & (1 << 6)
```

Gas Refunds

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.

Details about the optimiser

https://docs.soliditylang.org/en/latest/internals/optimizer.html#optimizer

Solidity has 2 optimiser modules

- at opcode level
  - Tries to simplfy expressions
  - Can inline functions, which may help it simplify bytecode
  - uses a rule list : https://github.com/ethereum/solidity/blob/develop/libevmasm/RuleList.h
  - removes duplicates
  - removes dead code
- Yul IR code level - more powerful as it can operate across function calls.
  - Functions have fewer side effects so easier to judge if independent
  - Can remove functions that are multiplied by zero
  - Can reorder functions

OPTIMISER - NUMBER OF RUNS

From the docs
"The number of runs (`--optimize-runs`) specifies roughly how often each opcode of the deployed code will be executed across the life-time of the contract.
This means it is a trade-off parameter between code size (deploy cost) and code execution cost (cost after deployment). A “runs” parameter of “1” will produce short but expensive code. In contrast, a larger “runs” parameter will produce longer but more gas efficient code. The maximum value of the parameter is `2**32-1`."

Dev Tools

Foundry
