The compiler's and the optimizer's effect on writing smart contracts

Intro

Have you ever wondered why you don't have to add a `` `require(msg.value == 0)` `` check to a non-payable function? What's going on when your smart contracts get compiled? It feels like magic, which is good because you should only worry about your Solidity source code, right? But what if your source code looks good, and you get unexpected results? How do you get to the bottom of code misbehavior?

The following citation is from the 6 parts series about Deconstructing a Solidity Contract on OpenZeppelin's blog:

> *You're on the road, driving fast in your rare, fully restored 1969 Mustang Mach 1. The sunlight shimmers on the all-original, gorgeous plated rims. It's just you, the road, the desert, and the never-ending chase of the horizon. Perfection!*
>
> *In the blink of an eye, your 335 hp beast is engulfed in white smoke, as if transformed into a steam locomotive, and you're forced to stop on the side of the road. With determination, you pop the hood, only to realize that you have absolutely no idea what you're looking at. How does this damn machine work? You grab your phone and discover that you have no signal.*
>
> *Could this perhaps be an analogy for your current knowledge of dApp development?*

A great effort has already been done explaining the bytecode of a compiled Solidity smart contract in this series, which will help understand how Solidity source code is transformed into EVM bytecode and how it can be debugged.

1. The compiler

The Solidity-commandline-compiler `` `solc` `` is a program that takes in Solidity code as input, parses it into an Abstract Syntax Tree (AST) and generates EVM bytecode out of it.

A big part of the Solidity-commandline-compiler is essentially powered by the optimizer, one of the most misunderstood features of the compiler, especially in its usage. More precisely, the optimizer consists of various optimizers and sub-optimizers that run in different stages of the compilation. The opcode-based optimizer applies a set of simplification rules to opcodes. It also combines equal code sets and removes unused code. The Yul-based optimizer is much more powerful because it can work across function calls. The Yul-based optimizer is also so powerful that it can be easily audited compared to its compiled EVM opcodes version. But first, let's take a step back and follow the Solidity source code to its EVM-bytecode version through the multiple optimizers.

In the following sections, we will discuss the internals of the different optimizers and how they affect the writing of code in Solidity.

Here is why a non-payable function is said to be more expensive than is payable counterpart. For example

```
Unset
function transfer(address to, uint256 amount) public { … }
```

turns into

```
Unset
switch FIRST_4_BYTES_OF_CALLDATA
case 0xc610bb07 { // transfer(address,uint256) function signature
    if callvalue() { revert(0, 0) } // check added by the compiler
```

Here the compiler adds an extra step to revert if `callvalue` returns a value greater than 0.

2. The bytecode optimizer

The compiler produces an internal data structure corresponding to a stream of assembly items. These assembly items are passed as input to the bytecode optimizer, which reconstructs them and returns an optimized version.

2.1. Under the hood of the bytecode optimizer

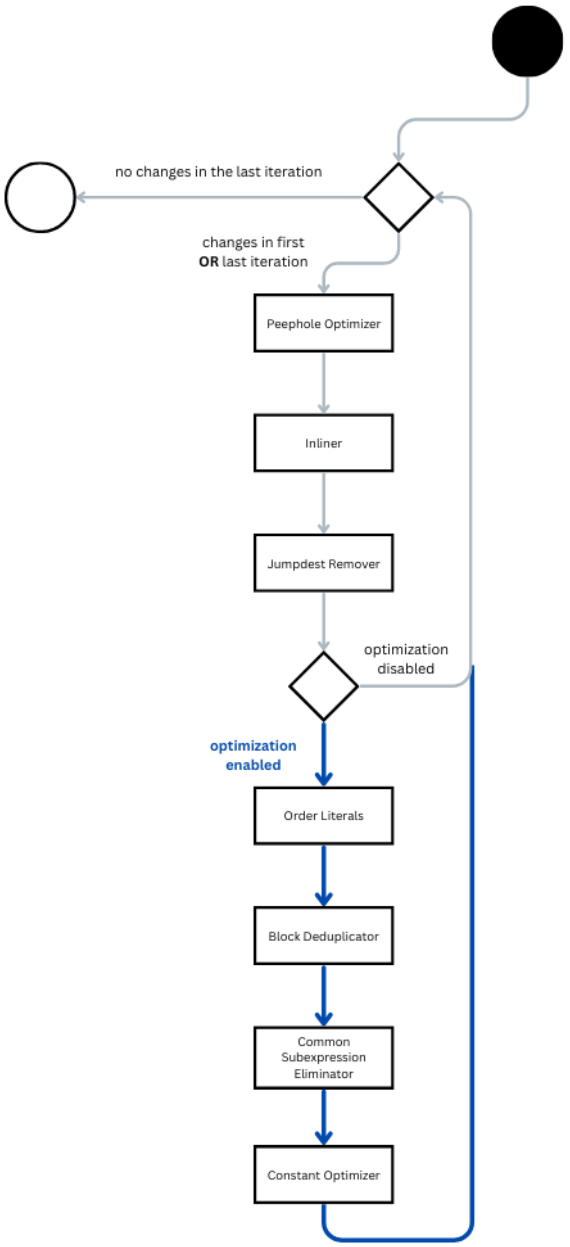
The bytecode optimizer is essentially a loop where all optimization steps are executed in order. This sequence will keep being repeated until no modifications to the bytecode are made during the last iteration.

The bytecode-optimizer takes 4 essential parameters, which consist of the `` `--optimize` `` flag, the version of the EVM if the bytecode is a deployment-bytecode of runtime-bytecode, and the expected execution per deployment also known as `` `--optimize-runs` ``, which defaults to 200 if the `` `--optimize` `` flag is set.

The last option causes a lot of confusion since many developers assume this is the number of runs the loop will make to optimize the bytecode. The number passed to `` `--optimize-runs` `` specifies an estimate on how often each opcode in this assembly will be executed, i.e., use a small value to optimize for size and a large value to optimize for runtime gas usage. Also, note that this parameter is not used much in the optimizer because almost all optimization steps improve both runtime and deployment costs. The main component in the optimizer that makes use of that parameter is the one that encodes constants and string literals.

2.2. Bytecode Optimization Steps

As discussed, bytecode optimization is enabled by setting the `--optimize` flag. Nevertheless, some optimization always happens even when this flag is not set. In this illustration, the first three optimization steps involving **Peephole Optimizer**, **Inliner**, and **Jumpdest Remover** are always executed in the optimization loop. The loop will keep reiterating as long as optimizations are still possible. The compiler will end the optimization if no further optimizations were found in the last iteration.



In the subsequent sections, the functionality of the individual optimization steps will be

explained in more detail.

2.2.1. Peephole Optimizer

Peephole optimization is an optimization technique performed on a small set of compiler-generated instructions; the small set is known as the peephole or window.

Peephole optimization involves changing the small set of instructions to an equivalent set with better performance. (https://en.wikipedia.org/wiki/Peephole_optimization)

For example:
1. instead of multiplying `` `x * 2**n` `` (21011 gas)

```
Unset
PUSH1 0x2
PUSH1 0xa
MUL
```

a peephole optimization might do a left shift by `n` bits (21009 gas)

```
Unset
PUSH1 0xa
PUSH1 0x1
SHL
```

This effect is even bigger with exponents. Instead of calculating `` `2**n` `` (21066 gas)

```
Unset
PUSH1 0xa
PUSH1 0x2
EXP
```

we can do (21009 gas)

```
Unset
PUSH1 0x2
PUSH1 0x9
SHL
```

2. if the optimizer finds a `SWAP` upcode executing a swap on two identical stack values, the `SWAP` will be omitted. The following sequence (21009 gas)

```
Unset
PUSH1 0xa
PUSH1 0xa
SWAP1
```

is the same as (21006 gas)

```
Unset
PUSH1 0xa
PUSH1 0xa
```

2.2.2. Inliner

Function inliner optimization is a technique that involves replacing a function call with the function itself. Inlining can also increase the code size, therefore inlining optimization uses heuristics to determine which functions are good candidates for inlining and which should be left as function calls.

2.2.3. Jumpdest Remover

The compiler often introduces generated jumpdests that are never used by the bytecode. **Jumpdest remover** optimization is a step that finds and removes redundant jump destinations (`JUMDEST`s) in the bytecode.

2.2.4. Order Literals

Order literals is an optimization technique that optimizes the order in which variables are evaluated. Certain combination of variables may lead to a faster evaluation of the expression.

2.2.5. Block Deduplicator

Block deduplicatior is a technique for eliminating redundant code sequences and replaces them with a single shared combination of opcodes.

2.2.6. CSE (common subexpression eliminator)

Common Subexpression Slimination is a method used by the optimizer that searches for identical expressions that evaluate to the same value and replaces and checks if it's cheaper to add store the value of these expressions in reusable variable.

For example:

```
Unset
a = x * y + w;
b = x * y + z;
```

can be replaced by

```
Unset
c = x * y;
a = c + w;
b = c + z;
```

only if storing `c` is more gas efficient than recalculating `x * y` every single time.

2.2.7. Constant Optimizer

Constant optimizer is one of the few techniques that are effected by the `--optimize-runs` option's value. If set to a high enough value, usually defaults to 200, the constant optimizer will make use of less opcodes and store constants in the runtime bytecode. If set to a low value, for example, to 1, the constant optimizer will try to minimize the size of the deployment and runtime bytecode, which will end up with a smaller contract size, e.g. cheaper deployment cost, but may be more expensive when executing functions.

For example, we have a constant calldata that looks like this `0xa9059cbb00000000000000000000000000000000000000000000000000000000000000000` and we want to store it to memory. This can be written as:

```
Unset
PUSH32
0xa9059cbb0000000000000000000000000000000000000000000000000000000000000000
PUSH1  0x00
MSTORE
```

(21012 gas)

but can also be written as:

```
Unset
PUSH4 0xa9059cbb
PUSH1 0xe0
SHL
PUSH1 0x00
```

```
MSTORE
```

(21018 gas)

Although both sequences produce the same result, the first one executes less opcodes, and is thus cheaper to execute, while the second one uses more opcodes and is less gas efficient using 6 more gas in total. But the first example is represented by the following `0x7fa9059cbb0000000000000000000000000000000000000000000000000000000000600052` bytecode, whereas the second one is represented by this `0x63a9059cbb60e01b600052` bytecode, and thus the second is cheaper to deploy because of the smaller size.

2.3. Limitations

The bytecode optimizer is limited to basic blocks. For example, there is a rule to replace a multiplication by 2 with a left shift. But this only works if the variable being multiplied is inside the block. The bytecode optimizer will fail to transform the following function to a left shift:

```
Unset
function f(uint256 a) public pure returns (uint256) {
      unchecked {
            return a * 2;
      }
}
```

The basic block for this function will be `JUMPDEST`, `PUSH 2`, `MUL`. Since the value of `a` is outside the basic block, this rule cannot be applied. Although there are workarounds around similar problems, the Solidity team decided to keep the bytecode based optimizer as simple as possible, and use the Yul IR-optmizer for more complicated optimizations.

3. The Yul IR-optimizer

In Solidity v.0.4.0, the Yul IR-optimizer was introduced as an experimental feature, where IR stands for intermediate representation. On the 22. March 2022, the Solidity Team announced that with the release of Solidity v.0.8.13 the Yul IR-optimizer is now considered production ready. In contrast to the bytecode optimizer, the Yul-optimizer does not work with assembly, but with the Yul intermediate-language.

The IR-based code generator was introduced with an aim to not only allow code generation to be more transparent and auditable but also to enable more powerful optimization passes that span across functions.
By default the optimizer applies its predefined sequence of optimization steps to the generated assembly. You can override this sequence and supply your own using the `--yul-optimizations` option:
`solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul:fDnTOc'`

The order of steps is significant and affects the quality of the output. Moreover, applying a step may uncover new optimization opportunities for others that were already applied, so repeating steps is often beneficial.

The sequence inside `[...]` will be applied multiple times in a loop until the Yul code remains unchanged or until the maximum number of rounds (currently 12) has been reached. Brackets (`[]`) may be used multiple times in a sequence, but can not be nested.

An important thing to note, is that there are some hardcoded steps that are always run before and after the user-supplied sequence, or the default sequence if one was not supplied by the user.

Here is a full table of the optimizer steps.

4. Debugging compiler and optimizer code

4.1. Storing variables

It is recommended, when declaring storage variables, to pack variables which storages add up to 32 bytes or less after one another to save storage slots, for example:

```
Unset
contract MyContract {
    bool foo;
    uint16 bar;
    uint256 baz;
}
```

Turns out the IR-optimizer also makes use of this by concatening adjacent variables that add up to 32 bytes or less and then storing both using only one `SSTORE` upcode.

For example in the following contract:

```
Unset
// SPDX-License-Identifier: unlicensed
pragma solidity ^0.8.19;

contract Storage {
    uint256 foo;
    bool bar;
    uint16 baz;

    constructor(uint256 _foo, bool _bar, uint16 _baz) {
       // notice how the `baz` gets assigned first
       baz = _baz;
```

```
        foo = _foo;
        bar = _bar;
    }
}
```

In the constructor we see that the assignment of the storage variables is different than the order of their declaration. Since there are no effects happening between the variables declaration, the optimizer can assign `foo` first, and then concatenate `bar` and `baz` and use only twice the `SSTORE` opcode in the constructor.

Example can be found in the repo in the `out/Storage_opt.yul` file.

4.2. Reassigning storage variables in the constructor

If we declare a storage variable and assign it a value, and re-assign it another value in the constructor, the optimizer will ignore the first assignment. For example:

```
Unset
// SPDX-License-Identifier: unlicensed
pragma solidity ^0.8.19;

contract Storage {
    uint256 foo = 42; // the optimizer will remove this assignment
    bool bar;
    uint16 baz;

    constructor(uint256 _foo, bool _bar, uint16 _baz) {
        foo = _foo;
        bar = _bar;
        baz = _baz;
    }
}
```

Example can be found in the repo in the `out/Storage_reassignment_opt.yul` file.

4.3. ForLoopConditionIntoBody and ForLoopInitRewriter

```
Unset
for { Init... } C { Post... } {
      Body...
}
```

is transformed to

```
Unset
for { Init... } 1 { Post... } {
      if iszero(C) { break }
      Body...
}
```

then to:

```
Unset
Init...
for {} 1 { Post... } {
    if iszero(C) { break }
      Body...
}
```

Example can be found in the repo in the `out/Loop_opt.yul` file.

4.3. Why is comparing to boolean value more expensive?

Question: Why doesn't Yul IR-optimizer convert the `if (condition == true)` into `if (condition)`?
Answer: `true` is read as a number, e.g. `0x01` and thus must be compared to `condition`.

4.5. ExpressionSimplifier

The expression simplifier executes simplifications similar to the Peephole Optimizer

```
Unset
x + 0
```

becomes

```
Unset
x
```

Example can be found in the repo in the `out/BoolComparison_opt.yul` file.

Same thing goes for

```
Unset
x * 2**n
```

which can be written as

```
Unset
shl(n, x)
```

5. Benefits of optimizing the code

Overall, the optimizer tries to simplify complicated expressions, which reduces both code size and execution cost, i.e., it can reduce gas needed for contract deployment as well as for external calls made to the contract. It also specializes on inlining functions. Especially function inlining is an operation that can cause much bigger code, but it is often done because it results in opportunities for more simplifications.

6. Security concerns

Vulnerabilities come in different shapes and forms. A vulnerability can be an error in the code, discrepancies between the code's behavior and the specifications, or design flaws. The Solidity compiler is developed by humans, and mistakes have already been made, found and fixed. It is also safe to say that mistakes will also be made in the future, and there are also probably mistakes in the compiler's code or design waiting to be discovered.

For developers – bugs within the Solidity compiler are difficult to detect with tools like vulnerability detectors, since tools which operate on source code or AST-representations do not detect flaws that are introduced only into the compiled bytecode.

Compiler bugs can have wildly varying consequences depending on the program control flow, but we expect that this is more likely to lead to malfunction than exploitability. Nevertheless, there has already been a decent amount of critical errors reported over the years.

The best way to protect against flaws in the compiler is to have rigorous end-to-end tests for smart contracts, covering all possible code paths, since bugs in a compiler very likely are not “silent” and instead manifest in invalid data.

For auditors – it is important to be aware of optimizer/compiler bugs, because we don't always have tests. Never assume that the compiler is perfect. A good source for keeping up to date with Solidity changes is to follow the Solidity team on twitter, read the published Solidity blogs on the official Solidity blog website and most importantly the security alerts.

For a sneak peek on what vulnerabilities a compiler might introduce, make sure to read the Solidity Compiler Audit done by OpenZeppelin for the commit e67f0147998a9e3835ed3ce8bf6a0a0c634216c5 (tag v0.4.24).