Solidity Optimizer

Solidity Summit

Hari

April 20, 2022. `https://hrkrshnn.com/t/devconnect.pdf`

Which one is more gas efficient[1]? a() or b()?

```
contract C {
    uint[] arr = [...];

    function a() external returns (uint sum) {
        for(uint i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
    }

    function b() external returns (uint sum) {
        uint[] memory arr_copy = arr;
        uint length = arr_copy.length;
        for(uint i = 0; i < length; i++) {
            sum += arr_copy[i];
        }
    }
}
```

[1] Inspired by a post from Patrick Collins.

How does the compiler work?

1. Solidity code is parsed into an Abstract Syntax Tree (AST).
2. Perform analysis on the AST.
3. Code generation:
   3.1 Legacy code generation: translate AST into EVM bytecode directly[2].
      - Perform bytecode based optimizations.

   3.2 Intermediate representation: translate AST into Yul. (`--via-ir` or `viaIR: true`)
      - Perform Yul optimizations.
      - Translate Yul into EVM bytecode.
      - Perform bytecode based optimizations.

[2] Yul and its optimizer is partially used.

Bytecode based optimizer

- Usually works across basic blocks.
- Simple evaluation of expressions.
  - $\text{add}(A, B) \rightarrow A + B$
  - Find more on RuleList.
- Cannot perform complex optimizations.

Where does the bytecode based optimizer fail?

- Rule[3]: $\text{mul}(a, 2) \rightarrow \text{shl}(1, a)$.

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

- Basic block: `JUMPDEST`, `PUSH 2`, `MUL`.
- Because the value a is outside the basic block, this rule cannot be applied.
- Engineering decision: we want to keep the bytecode based optimizer as simple as possible.

[3] Based on a question by Alexey.

Yul Optimizer

- Can perform more complex optimizations across blocks.

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

Approximate IR:

```
function f(a) -> r {
    r := mul(x, 2)
}
```

Optimized into:

```
function f(a) -> r {
    r := shl(1, x)
}
```

- The `--via-ir` codegen can optimize `mul(x, 2)` into `shl(x, 1)`.

What can Yul do better?

- Remove redundant division by zero checks.
- More inlining.
- Better stack management.
- Packed structs are better optimized.
- Trivial things like "`x != 0` instead of `x > 0`".
- Small and independent steps that can be executed in a sequence.

Loop invariant code motion

```
function a() external returns (uint sum) {
    for(uint i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
}
```

Translated Yul excluding a certain optimization:

```
let sum := 0
for {let i := 0 } lt(i, sload(0)) {
    // overflow check for i
    if eq(i, not(0)) { panic() }
    i := add(i, 1)
}
{
    let arr_value_i := sload(add(HASH, i))
    // Overflow check for +=
    if gt(arr_value_i, not(sum)) { panic() }

    sum := add(sum, arr_value_i)
}
```

Loop invariant code motion

The length (`sload(0)`) is an invariant in the loop!
Optimized code:

```
let sum := 0
let len := sload(0)
for {let i := 0 } lt(i, len) {
    // overflow check for i
    if eq(i, not(0)) { panic() }
    i := add(i, 1)
}
{
    let arr_value_i := sload(add(HASH, i))
    // Overflow check for +=
    if gt(arr_value_i, not(sum)) { panic() }

    sum := add(sum, arr_value_i)
}
```

The other way to compute sum

Copying array into memory first. Then do all computations there. Has memory overhead.

```
function b() external returns (uint sum) {
    uint[] memory arr_copy = arr;
    uint length = arr_copy.length;
    for(uint i = 0; i < length; i++) {
        sum += arr_copy[i];
    }
}
```

In legacy codegen, this is cheaper. But `viaIR` makes the simple implementation cheaper!

A note on benchmarking

- Benchmark optimizations.
- Function dispatch can affect gas.
- Look at diffs of assembly as well as IR.

Dispatch affecting gas

```
contract C {
    function a() external {}
    function b() external {}
    function c() external {}
}
contract CByteCode {
    // pseudocode for function dispatch of C
    fallback() external {
        if (msg.sig == 0x0dbe671f)
            a();
        else if (msg.sig == 0x4df7e3d0)
            b();
        else if (msg.sig == 0xc3da42b8)
            c();
    }
    function a() internal {}
    function b() internal {}
    function c() internal {}
}
```

Dispatch affecting gas

- The order of the function in the dispatch can introduce an overhead.
- For benchmarking gas, try to have contracts with only a single function.
- Certain frameworks can display the gas without the dispatch overhead.

Diffs of assembly or IR

- The `--asm` option is more readable than the binary.
- The `--ir-optimized --optimize` is even more readable.
- godbolt.org now has Solidity support!

Future plans for the optimizer

- General idea: intuitive code is also optimal.
- Improving inlining heuristics.
- Complex optimizations that require symbolic reasoning:

```
function f() external returns (uint sum) {
    for (uint i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
}
```

- Remove redundant reverts.
- Optimize memory usage.

Safety of the optimizer

- General fuzzing and differential fuzzing by Bhargava.
- External fuzzing efforts from academia (Alex Groce, etc).
- Formally verify some steps in Z3.
- Symbolically check some optimization steps from dapptools.
- Hand written proofs, PR reviews, etc.

No known compiler bug on a deployed contract!

Slides

https://hrkrshnn.com/t/devconnect.pdf