"ALEXANDRU-IOAN CUZA" UNIVERSITY, IAȘI

FACULTY OF COMPUTER SCIENCE

MASTER'S THESIS

Solidity Optimization with Control Flow Graphs

proposed by

Sergiu Iacob

Session: july, 2022

Scientific Coordinator

Dr. Arusoaie Andrei

"ALEXANDRU-IOAN CUZA" UNIVERSITY, IAȘI

FACULTY OF COMPUTER SCIENCE

Solidity Optimization with Control Flow Graphs

Sergiu Iacob

Session: july, 2022

Coordinator

Dr. Arusoaie Andrei

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Static analysis has been at the core of **optimizing** code for many years now, doubled by its sibling, dynamic analysis. Very well known programming languages such as C++, Java, Golang etc. generate **bytecode**, by the usage of **compilers**, that is eventually interpreted by various virtual machines, such as LLVM, on various architectures — x86, x86_64, AMD64, arm64 etc.

While the virtual machines that run bytecode and the cpu architectures are shared across programming languages, since they ultimately revolve around the same assembly instructions, each programming language has its own compiler while, of course, interpreted languages such as Python have their own interpreter.

Zooming in on how a compiler works, we see that the optimization step can be seen as a level in the Maslow pyramid, but in the software engineering world. While any compiled code may run just fine without it being optimized, it will more than likely miss on various performance improvements that will greatly speed up the code's run time and resources' usage.

Writing high level code often means that one cannot always invest in the needed resources (time and knowledge) to perfectly optimize its code, nor can he/she make a cpu's frequency higher by writing a line of code. What the compiler can do, instead, is less. Running less instructions, in a better order, using registers more efficiently and moving less memory blocks around is what gets us the fastest computation. This is ultimately the optimizer's job in a compiler — find opportunities to generate more efficient bytecode.

1.2 Objective

The purpose of this thesis is to research the techniques of applying static analysis on high level code, in order to obtain efficient bytecode, regardless of the virtual machine it will execute on. These techniques are usually backed up by **intermediate representations** of the initial code, which can be either another high level code (with different syntax, but the same semantics), or data structures meant to provide an overview (or in-depth view) over the program execution.

In this thesis, we will see how intermediate representations help, why they're useful, and how **Control Flow Graphs** determine which optimizations can be applied while doing static analysis. The programming language we will focus on is **Solidity** [1].

1.3 Motivation

Solidity is a new programming language on the market, backing up the development of Smart Contracts deployed on the Ethereum blockchain. Since it is a relatively new technology, there is a lot of ongoing work to still build the basis of the eco-system and to enhance the compiler by basic features or well-known optimizations.

An important aspect is that Solidity is an open-source technology, which facilitates **external contributions**, some of which will be provided by the usage of this thesis. Optimization sits at the heart of the compiler, and as the official documentation states, [2, the optimizer is under heavy development]. This is sufficiently encouraging to research on how Control Flow Graphs, intermediate representations and static analysis techniques could be combined in order to improve the bytecode generated from Solidity.

1.4 Thesis structure

Bringing external contributions to a technology usually follows a logical timeline, around which this thesis was also structured. Specifically in our case, we'll go through understanding Solidity and its usecases, understanding what a compiler and what a

[1] Solidity is a highly volatile language at this point, therefore it is probable that some of the details presented in this thesis will lose veridicity in the near future. However, the high-level schemas, approaches and structure of the Solidity eco-system will likely remain the same.

optimizer is, how Solidity's compiler works, what are its optimizer's drawbacks and how we should solve these. Finally, we go through integrating our improvements into the actual Solidity codebase.

In the first chapters, we set the knowledge base needed to improve the optimizer. We go through Solidity and the (relatively) new Yul IR, static analysis, control flow graphs (CFG), abstract syntax trees (AST) and the Solidity optimizer structure.

In the latter chapters, we do a deep dive into a few optimization steps and identify edge cases not optimized by these and do a few benchmarks to see how much gas is saved through these improvements. Finally, we conclude with some experiments on Etherscan, where we try to optimize already optimized verified smart

Chapter 2

Contributions

2.1 A way to contribute

The most challenging part was building smart contracts that were not fully optimized by Solidity's compiler, i.e. the generated bytecode contained redundant bytecode. This mostly meant finding situations, edge cases in which the compiler was "fooled" by the high level code, making it impossible for various optimizations to be applied or simply ignored.

The first approach here was to experiment with Solidity and compile smart contracts until such an edge case was found. The challenge here is that it is very difficult and time consuming to analyze the generated bytecode and try to find an optimization opportunity, since assembly instructions are not "human readable".

The second approach was to reverse engineer Solidity's open-source codebase, get familiar with the internal optimizer structure and brain storm on edge cases that are not treated by the optimizer. While this is definitely a good approach, it requires a bit of experience with optimizers and some time to get familiar with Solidity's codebase.

What helped by a great margin was the usage of **YUL Intermediate Representation**, which can be analyzed in its optimized form, since it is the intermediate code used to generate the final bytecode. This way, it was much faster to analyse whether the generated bytecode would be optimal or not.

2.2 Enhancement of optimization steps

Solidity's optimizer has 32 documented optimization steps, out of which this thesis focuses on two of them: **UnusedPruner** and **UnusedAssignmentEliminator**. One of the most straightforward way to speed up code computation is to run less code, which comes from generating the proper bytecode and from pruning "dead code", i.e. high level code that the user wrote but is unreachable. We'll also take a look at **StructuralSimplifier** and **DataFlowAnalyzer**, the latter being a core component of the optimizer that enhances static analysis.

In this thesis, I've managed to identify several edge cases that fool the optimizer, and by treating the respective edge cases, I've improved the functionality of the two optimization steps mentioned above. A few other observations are made throughout the thesis, which can easily transform into external contributions to ultimately generate a more efficient smart contract.

2.3 Reducing gas usage of smart contracts

The quality of Solidity's compiler and optimizer is evaluated by the **gas usage** of the smart contract, both deployment on the network and the cost of running a function by its signature. With the optimizations mentioned above, the optimizer **generated code that requires less gas**, which means less ethereum, which means smaller costs for the user.

Multiply any small, seemly insignificant gas improvement by the millions of smart contracts on the Ethereum blockchain and you end up with a pretty significant cost reduction in code execution. Therefore, external contributions as these will sum up to bring a real difference in code quality[1].

[1]We will dive deeper into which smart contracts will benefit from these optimizations and how optimization steps give more opportunities to other optimization steps in the latter chapters.

Chapter 3

Solidity

3.1 Overview

Solidity is a relatively new programming language, its first proposal coming in 2014 from the co-founder of Ethereum, Gavin Wood [1]. Its purpose is to serve development of smart contracts to be deployed on various blockchain platforms, especially focusing on the Ethereum blockchain.

Although it's been around for around 8 years now, Solidity has just started to peak interest in the blockchain development community. Dune Analytics reported an increase of 75% in smart contracts deployment in March, 2020, summing up to a total of 2 million smart contracts deployed [3]. Since then, hundreds of thousands of smart contracts have been deployed monthly, with around 370 thousands just in May 2022 3.2.

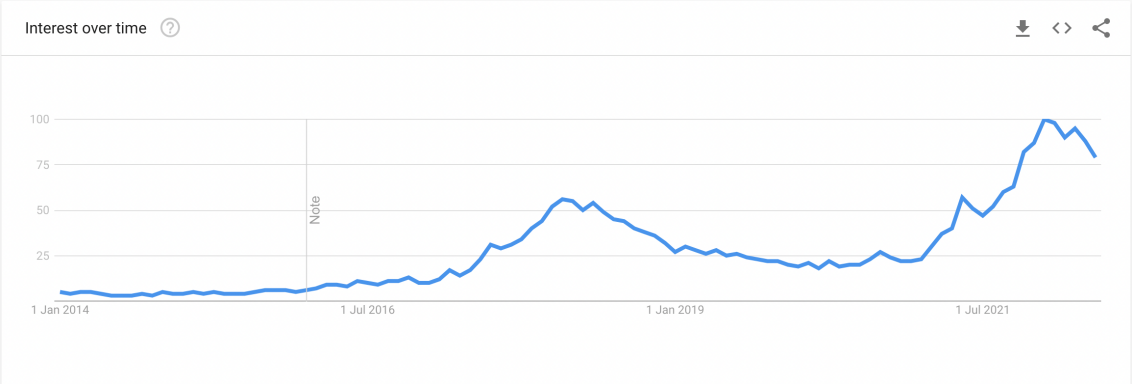

Figure 3.1: Solidity interest according to Google web searches

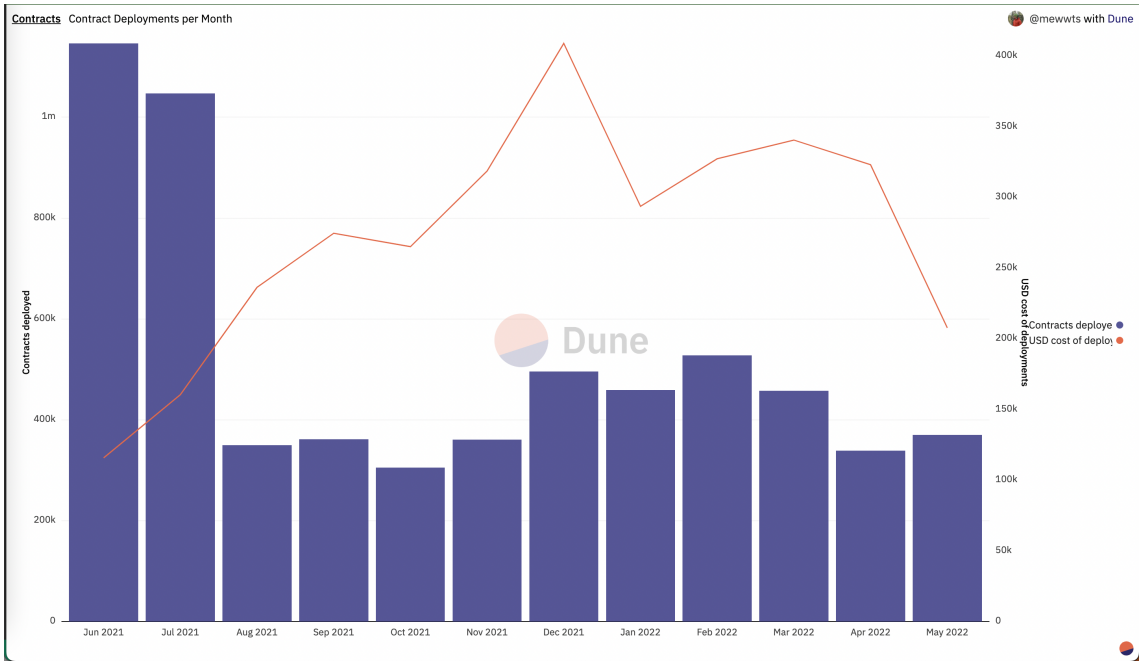

Figure 3.2: Contract deployment per month in the past year, according to Dune Analytics

The properties that interests the reader are that this programming language is **statically typed** and is converted into bytecode to run inside the Ethereum Virtual Machine (EVM). Being statically typed helps the compiler with static analysis, making possible the development of an optimization suite that performs both general and specific optimizations for EVM.

3.2 Solidity's Compiler and Optimizer

High level solidity code is sent to `solc` binary, which stands for "solidity compiler". The user may or may not specify the usage of the optimizer with the `--optimize` flag, which is enabled by default. Upon analyzing the compilation flow 3.3, we notice that there are some optional (yet recommended) steps to follow while compiling, which take advantage of the Yul IR and Abstract Syntax Trees, the latter being built for both the initial Solidity code and for the Yul IR. If we choose to optimize the code through Yul, then the `--via-ir` must be specified, which is also enabled by default.

When optimizing through Yul, the compilation flow will actually use **two optimizers**: the "new" one (Yul optimizer) and the "legacy" one (EVM bytecode optimizer). The reasoning behind this is that bytecode optimization should be as sim-

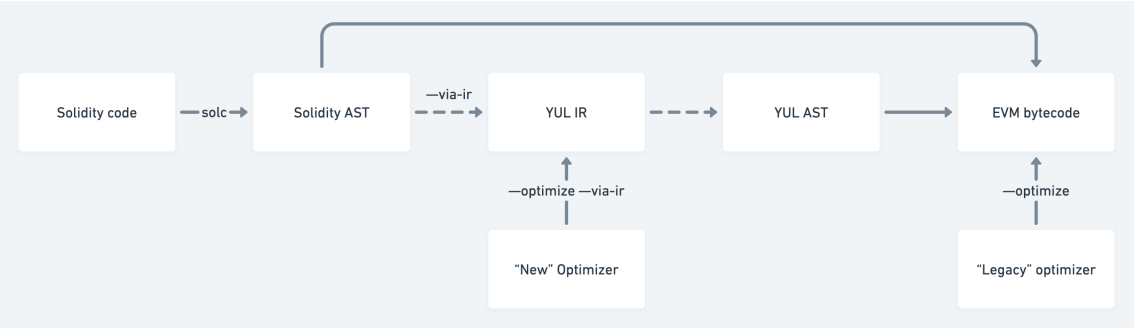

Figure 3.3: Solidity compilation flow

ple and straightforward as possible, this also being an engineering decision made by Solidity's team [4]. `JUMP` instructions are especially tricky to handle while optimizing bytecode, making high level, semantic optimization (for example simplifying `for` loops) more difficult.

The advantage of having two optimizers is that it decouples work by a great deal. Low level assembly optimizations are done by the legacy optimizer [1], and more complex optimizations, that require data flow analysis for example, are done directly on the Yul IR, which is later used to generate the efficient bytecode. This also helps by visually inspecting the generated Yul IR and making sure that its output is the expected one. Specifically for the team, it means that it makes unit testing and development much easier.

3.3 Intermediate Representations

Abstract Syntax Tree

Abstract Syntax Trees (AST) are often used to do a first analysis of the code instructions by walking the tree. This is often coupled with the Visitor Design Pattern, where the user implements what the `visit` function should do for each type of node. These are [7, "widely used in compilers to represent the structure of program code"].

On Solidity's territory, ASTs appear in two phases. First, an AST is generated from the unaltered Solidity code, while another AST is later generated from the Yul IR code. The AST nodes provide sufficient granularity to make a structural analysis simple enough, thanks to recursively analysing the children node of a root.

[1] Future work might also consist in unifying EVM and LLVM, bringing all of the work and optimization done to LLVM on Solidity's ground as well.

Figure 3.4: Solidity AST Example. The body of the function is an AST node itself, in which the statements are other AST nodes. Generated using AST Explorer.

ASTs are very common in static analysis and is often times coupled with a grammar (or dialect) that enforces different node types and node structure in the tree. There are a variety of tools that use ASTs to do code analysis of their own.

Yul IR

Yul, former name Julia, is an intermediate representation for Solidity Code. It has been publicly announced as "production ready" in March, 2022, in version 0.8.13 of Solidity. Since then, the engineering team has been encouraging people to compile smart contracts via the usage of Yul, since the optimization focus has shifted on this pseudocode language.

Yul was designed around four main principles, as described by its official documentation [8]: readability, easy manual inspection of control flow, simple and straightforward generation of bytecode from Yul and whole-program optimization. As compared to bytecode, this IR provides high-level syntax for instructions that introduce jump instructions (`JUMP`, `JUMPDEST`, `JUMPI`), the reasoning behind this being that it is much easier to analyze data flow and control flow in the code.

```
{
    function power(base, exponent) -> result
    {
        switch exponent
        case 0 { result := 1 }
```

```
6           case 1 { result := base }
7           default
8           {
9               result := power(mul(base, base), div(exponent, 2))
10              switch mod(exponent, 2)
11                  case 1 { result := mul(base, result) }
12          }
13      }
14  }
```

Listing 3.1: Example of Yul code which computes exponentiation recursively

This was designed as an actual programming language, making it possible to generate valid EVM bytecode from Yul code. Of course, it is not recommended to write smart contracts in Yul, but one can do so to get more familiar with the dialect. It also takes on the property of its ”parent” code, and is also statically typed as Solidity. This further eases the static analysis done by the compiler.

3.4 Other tools performing static analysis

The compiler has the option of outputting the Solidity AST in JSON format, which means that external tools can be built to alter the AST in any desired way. Since the compiler also supports receiving an AST as input, and generate bytecode from that one, that means that in theory one could build a better optimizer than the official one. While that hasn't happened yet, there are a few tools that perform code analysis of their own.

3.4.1 Solidity Instrumentation Framework (SIF)

Built by Chao Peng, Sefa Akca and Ajitha Rajan, from University of Edinburgh, SIF is a tool built in C that gravitates around the Visitor Design Pattern. The main objective is to build an internal C representation of Solidity code, through intermediary data structures, upon which SIF can orchestrate various optimizations / refactoring functions. As the authors describe it, it's a tool [5, to easily and effectively understand, manipulate and analyse Solidity code].

The drawbacks that this toolkit currently has is that it's outdated and that it does

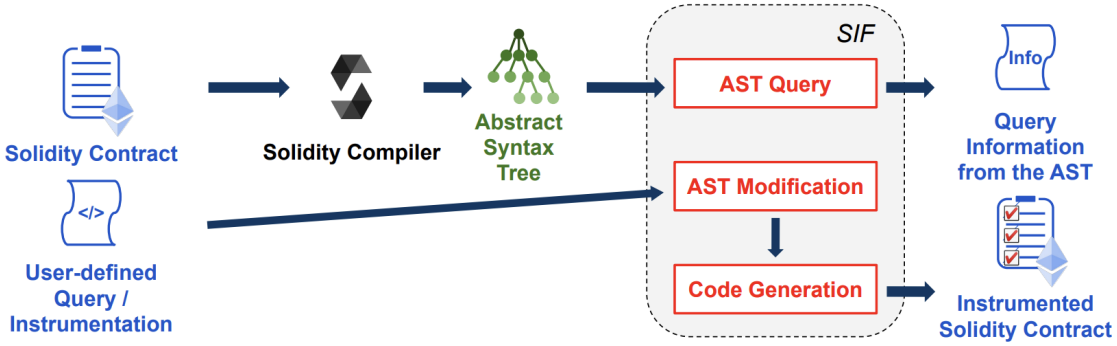

Figure 3.5: SIF workflow, captured from SIF: A Framework for Solidity Contract Instrumentation and Analysis[5]

not allow for external users to ”plug-in” their own intermediate representations of Solidity code. It acts more as a middle-man, allowing users to implement, through the *visit* method, specific optimization / refactoring items that run against isolated pieces of code. The tool does build a Control Flow Graph for the code, the one that this thesis will focus on, but it does not specifically use it for any kind of in-house optimization - it is just available to the user for usage. However, there are issues of Control Flow Graph completeness. ¡- TODO aici paper-ul cu completeness

3.4.2 Slither

Slither is a more abstract, high level static analysis framework, which gives a broader overview over the internals of an Ethereum smart contract. Its main focus is centered around 4 areas: security (vulnerability detection), automated code optimization, code analysis and assisted code review, as described in Section 4.1 by the author Margherita Renieri [6].

The tool takes advantage of the Abstract Syntax Tree (AST) built by the Solidity Compiler, then passes that through a series of optimization steps: Information Recovery, SlithIR Conversion and Code Analysis.

What’s common in both of the tools we’ve seen is the usage of Abstract Syntax Trees and Control Flow Graphs, as well as building ”proprietary” intermediate representation of the input code.

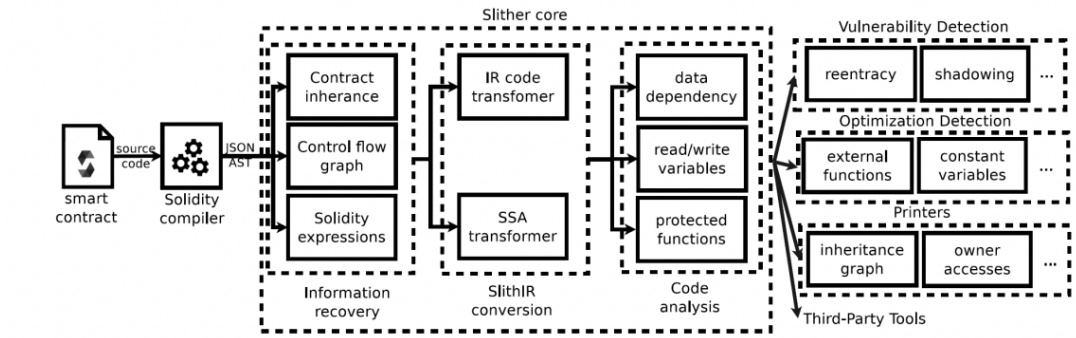

Figure 3.6: Slither Architecture

Why enhancing the official optimizer is better

The issue with such tools as SIF or Slither is that they get rapidly deprecated, since Solidity ground is so volatile, and they don't provide better optimization flows than Solidity's compiler does. Furthermore, integration is a bit more difficult, since it requires a separate optimization pipeline to be run through this intermediary tools, which just adds an additional burden to generating efficient bytecode.

What is certain is that the official compiler and optimizer will stay at the core of generating the Solidity bytecode. Therefore, any improvements to that will reflect itself in the next Solidity release - given it passes the team's rigurous reviews, of course.

Chapter 4

Contributions

Solidity is a relatively new technology, when compared to already existing programming languages widely used in the market. As stated by the Solidity team itself, "the optimizer is under heavy development" [2], which gives plenty of room for contributors to bring their own performance enhancements.

I plan on being one of those contributors, starting from the Solidity code and the AST built by the Solidity compiler itself. I will then evaluate the given inputs and focus on building a specific, low resolution Control Flow Graph, by which memory allocation and cpu cycles can be thoroughly inspected. This would allow for a technical deep dive into the insides of an Ethereum smart contract, which in return can provide useful insights to the user as to how the code can be optimized.

As compared to the other existing tools, this thesis will focus on the latest stable version of Solidity (0.8) and will be built around a set principles focused on scalable, maintainable code analysis tools. Security will also not be the main concern, as there is already plenty of interest in that area.

YUL Intermediate Representation

Introduction

YUL IR is relatively new in the Solidity compiler.

Control Flow Graphs

Solidity Optimizer

A way to compute the optimizer's efficiency

Daddy Yankee – gasolina.

Control Flow Graphs Study Case

Termination flows * stop() == return(0, 0) * return(p, s) – end execution, return data mem[p…(p+s)) * revert(p, s) – end execution, revert state changes, return data mem[p…(p+s))

Benchmarking

Conclusions

Solidity is still a place where "construction is in progress". What's common across optimizing all of the versions of this programming language is common software techniques, known as static analysis. Control Flow Graphs are a center piece of this mechanism and represents a good starting point, even more valuable when augmented by ASTs and other intermediate representations of the code, such as op code or YUL.

State of the art in Smart Contract Optimization is still receiving inputs, modifications, enhancements from many parts as parties, as the programming language itself is still under heavy development. https://www.overleaf.com/project/627ab6f7ebec3330916a293b

Bibliography

[1] Solidity Language, Solidity Team, 2022

[2] Solidity v0.8.13 Official Optimizer Documentation, Solidity Team, 2022

[3] Ethereum Smart Contracts up 75% to Almost 2M in March, Coin Telegraph, 2020

[4] The Solidity Optimizer @ Solidity Summit, Hari Mulackal, 2022

[5] Solidity Instrumentation Framework (SIF), Chao Peng, Sefa Akca, Ajitha Rajan, University of Edinburgh, 2019

[6] Slither: A Static Analysis Framework For Smart Contracts, Josselin Feist, Gustavo Grieco, Alex Groce, 2019

[7] Abstract Syntax Trees, Wikipedia, 2022

[8] YUL Motivation and High-level Description, Solidity Team, 2022