OPTIMIZING COMPILATION OF ARRAY ACCESSES IN SOLIDITY SMART CONTRACTS
COMPILACIÓN OPTIMIZANTE DE ACCESOS A ARRAYS EN CONTRATOS INTELIGENTES EN SOLIDITY

Trabajo de Fin de Grado
Curso 2022–2023

Autor
Javier Sande Ríos

Director
Jesús Correas Fernández

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

OPTIMIZING COMPILATION OF ARRAY ACCESSES IN SOLIDITY SMART CONTRACTS
COMPILACIÓN OPTIMIZANTE DE ACCESOS A ARRAYS EN CONTRATOS INTELIGENTES EN SOLIDITY

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Javier Sande Ríos

Director
Jesús Correas Fernández

Convocatoria: *Junio 2023*

Grado en Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

May 28, 2023

Agradecimientos

A mis padres por brindarme la oportunidad de estudiar. Junto a ellos, a mi hermana y a toda mi familia por el constante cariño y apoyo incondicional que siempre me han brindado.

A mis compañeros y amigos por todos los momentos inolvidables que hemos compartido a lo largo de este camino.

A la Universidad y a todos sus profesores por su dedicación e incansable esfuerzo, especialmente durante los difíciles años de la pandemia, por en enseñarnos y proporcionarnos los recursos necesarios para convertirnos en excelentes profesionales.

Por último, pero no menos importante, quiero expresar mi más profundo agradecimiento a mi director en este trabajo, Jesús Correas Fernández, por su guía durante todo el proceso y por su esfuerzo para adaptarse a las dificultades que implicó la diferencia horaria

Abstract

On Ethereum, smart contracts must have two key characteristics: efficiency, an essential feature in smart contracts with a direct economic impact on the user, and security. The compiler of Solidity, the most widely used language for programming Ethereum smart contracts, automatically incorporates various security checks to avoid programming errors. This is the case of bounds checks on array accesses. Nevertheless, these checks introduce some computational overhead, which might be unnecessary in some scenarios.

In this project, two approaches are proposed to reduce the costs associated with these controls and, consequently, array accesses. First, we introduce a new Solidity code construct that allows you to disable bounds checks in those sections of code that programmers deem unnecessary. Secondly, we propose a new compiler optimization phase that takes advantage of the high-level language structures of the program in order to optimize low-level accesses to arrays and relax the applicability conditions of current optimizations. Finally, we confirm through experimental results the effectiveness of both solutions in reducing the computational cost of array accesses.

Keywords

Ethereum, blockchain, smart contract, Solidity, optimizing compiler, runtime check.

Resumen

En Ethereum, los contratos inteligentes deben tener dos características clave: eficiencia, característica esencial con un impacto económico directo en el usuario, y la seguridad. El compilador de Solidity, el lenguaje más utilizado para programar contratos inteligentes en Ethereum, incorpora automáticamente varios controles de seguridad para evitar errores de programación. Este es el caso de los controles de límites en los accesos a arrays. No obstante, estas comprobaciones introducen cierta sobrecarga computacional, que puede ser innecesaria en algunos escenarios.

En este proyecto, se proponen dos enfoques para reducir los costos asociados con estos controles y, en consecuencia, con el acceso a arrays. Primero, presentamos una nueva construcción de código de Solidity que permite deshabilitar las comprobaciones de límites en aquellas secciones de código en las que los programadores las consideran innecesarias. En segundo lugar, proponemos una nueva fase de optimización en el compilador, que aprovecha las estructuras del lenguaje de alto nivel del programa para optimizar los accesos de bajo nivel a los arrays y relajar las condiciones de aplicabilidad de las optimizaciones actuales. Finalmente, confirmamos a través de resultados experimentales la efectividad de ambas soluciones para reducir el costo computacional de los accesos mediante índice a arrays.

Palabras clave

Ethereum, blockchain, contratos inteligentes, Solidity, compilador optimizador, comprobaciones en tiempo de ejecución.

Contents

List of figures

List of tables

Introduction

Ethereum is one of the most important blockchain networks in the current landscape. Its launch altered the blockchain world due to its smart contracts, programs whose code and state can be stored in the blockchain network to be executed by all users.

Smart contracts, commonly developed using the Solidity language, are used by network users through transactions and executed in the nodes. These nodes, known as miners, are economically rewarded proportionally to the computational cost of the executed transaction. This compensation, paid by the user, makes the efficiency of the contracts essential, encouraging responsible use of the blockchain and avoiding attacks that block network resources.

Another essential feature of smart contracts is security. These programs often manage and store digital assets of great value, exposed to all network users. Therefore, contracts must be as secure as possible. However, sometimes security and efficiency clash, as in the case of array accesses on Solidity smart contracts.

The use of arrays is a widespread practice in programming. Arrays are a flexible data type, extremely useful for storing and structuring data in programs, and smart contracts are no exception. However, the cost of accessing arrays is considerably high in Solidity smart contracts. In order to avoid programming errors, the Solidity compiler automatically generates bounds checks on each index access. Therefore, each access to an index involves two memory loads to access the length and the value, significantly increasing the cost. This high cost implies a limitation in their use, especially when storing non-volatile data, one of the most common uses of arrays. Therefore, the optimization of array accesses is of great interest.

In this project, we propose two possible solutions for this problem. On the one hand, we will propose a Solidity language construct that allows programmers to eliminate bounds checks they consider unnecessary. On the other hand, we will propose a new compiler optimization to reduce the array length loads when possible and, consequently, reduce the cost of accessing arrays.

Goals

The main goal of this project is to design, implement and test two optimization proposals to reduce the gas consumption associated with array accesses in Solidity smart contracts. In order to achieve it, we initially set the following goals:

- Understand how the current production compiler works, its internal and intermediate representations of the code, and its compilation phases.
- Understand current approaches of the Solidity language and its compiler to generate optimized code, their applicability, and limitations.
- Implement an optimization at the language level that allows developers to disable bounds checks on array index accesses.
- Implement a new optimization able to reduce array access gas cost in situations the current optimization modules cannot.

Planning

This project was divided into two main phases developed between September 2022 and May 2023. Each project phase focused on one of the two array optimization proposals.

Language optimization block. During the first phase, between September and December, we focused on implementing a new language construct in Solidity, called `uncheckedArray` block, to provide developers with a tool to disable bounds checks in array accesses to reduce gas consumption. In order to implement this language extension, we first performed a deep study of the Solidity language, its compiler, and a similar solution that the language provides to reduce gas consumption on arithmetic operations. Then, we decided how to name and structure this new block, its possible use, and its limitations. Finally, we implemented the block to test and analyze the real performance of this new optimization tool on smart contracts.

Compiler optimization. In the second phase of the project, we implemented a new optimization at compile time to reduce the cost of array accesses inside loops. During this phase, we used all the knowledge about the compiler acquired in the first phase. Nevertheless, we needed to perform some research and experiments in order to understand the optimizations currently performed by the compiler, its performance, and its limitations. Once we had a clear idea of how we could improve

the optimizations, we focused on establishing the constraints needed to guarantee the soundness of our new optimization. Finally, we implemented an experimental version of the proposed optimization and integrated it into a production version of the official compiler. We measured the potential impact on the gas consumption of real smart contracts by performing several tests with non-trivial benchmark programs.

During both phases, we maintained weekly meetings to discuss the difficulties found, decisions taken, implementation details, and experimental evaluation, as well as the results obtained with benchmark programs and conclusions outlining some lines of future work.

Code repositories

The developed optimizations are available in the following GitHub repository:

`https://github.com/javierSande/solidity`

This repository is a fork from the original Solidity repository[1], and it is publicly available. The repository is composed of four different branches:

- `develop` branch: contains the version of the official `develop` branch we used as a base for our development, corresponding to version v0.8.19[2] of the compiler as of February 23, 2023.
- `uncheckedArray` branch: contains the basic implementation of the `uncheckedArray` block as presented in Section 2 of Chapter 3.
- `targetedUncheckedArray` branch: contains the final implementation of the `uncheckedArray` block as presented in Section 3 of Chapter 3.
- `arrayLoopOptimization` branch: contains the implementation of the compiler optimization as presented in Chapter 4.

Additionally, the benchmarks developed for the evaluation of the optimization presented in Chapter 4 are available in the following GitHub repository:

`https://github.com/javierSande/solidity-benchmarks`

The `README` page of the repository contains the setup instructions to perform gas cost evaluations of code generated by an official or experimental version of the Solidity compiler.

[1] The official Solidity repository can be found at `https://github.com/ethereum/solidity`.
[2] Commit 983407762c3423e9c301d5ae56ac7b6d951655df

Structure of the document

This memory contains the background, implementation details, and conclusion of the proposed optimizations. The document is structured as follows:

- **Chapter 1:** introduces Ethereum, the Ethereum Virtual Machine, and Solidity language. This chapter provides the theoretical and technical background to understand the necessity and possibilities for creating optimizations.
- **Chapter 2:** introduces the Solidity compiler, its compilation process, and state of the art on Solidity code optimizations.
- **Chapter 3:** presents the `uncheckedArray` block, a new Solidity structure to allow developers to disable safety checks in favor of efficiency.
- **Chapter 4:** presents a new compiler optimization phase that takes advantage of a higher level of code representation to optimize array accesses inside loops.
- **Conclusions and Future Work:** presents the conclusions of the optimizations developed in our project and proposes lines of work for future development of the proposed optimizations.

Chapter 1

Ethereum

Ethereum is an open-source, decentralized blockchain network conceived by Vitalik Buterin in 2013 and officially launched in 2015 [25]. It is founded on the basis of the blockchain protocol, first proposed by David Chaum in 1982, and first implemented by *Satoshi Nakamoto*[1] in 2008 with the creation of Bitcoin [19].

As a blockchain, Ethereum is a distributed ledger where transactions are recorded into blocks linked using cryptographic hashes. Like many other blockchain networks, it has its own digital currency, *ether*, used to store value, perform exchanges, pay fees, and reward participants of the network, such as the mining nodes.

What makes Ethereum special is that it was the first programmable blockchain. This means that the network can be used to execute programs. Those programs are known as *smart contracts*, a concept first coined in the 1990s by Nick Szabo, who defined them as:

> *A set of promises, specified in digital form, including protocols within which the parties perform on these promises.* [22]

Smart contracts are programs whose code and state are stored on the network at a specific address, allowing anyone to interact with them through function calls executed in the *Ethereum Virtual Machine*(EVM). When a blockchain user wants to communicate with the smart contract, it sends a transaction to its address specifying the function of the contract it wants to execute. Then, this transaction is validated by the network nodes, which execute the corresponding function using the EVM on its local machine.

The possibility of storing and executing programs in a decentralized environment in a secure manner has opened the door to a large number of new technologies, such

[1] Pseudonymous person or group of persons who developed Bitcoin.

as decentralized apps, known as *dApps*, *Non-fungible tokens* (NFTs) and the *Web3*. All of them are based on the smart contract technology introduced by Ethereum.

Ethereum is, at the time of this writing, a system with a market capitalization of $200B, which performs more than 1 million transactions and has half a million active addresses daily[2]. It is a network in continuous growth, laying the foundations for the future of decentralized systems.

1.1. Ethereum Virtual Machine

The *Ethereum Virtual Machine* (EVM) is a virtual machine capable of Turing-complete computation. It is a runtime environment executed by the blockchain nodes to process the blockchain transactions, including the function calls to smart contracts. The EVM can be seen as the state function of the Ethereum network [25]. It takes an input (state of the blockchain, its contracts, and code to be executed), performs some computations, and outputs a new state.

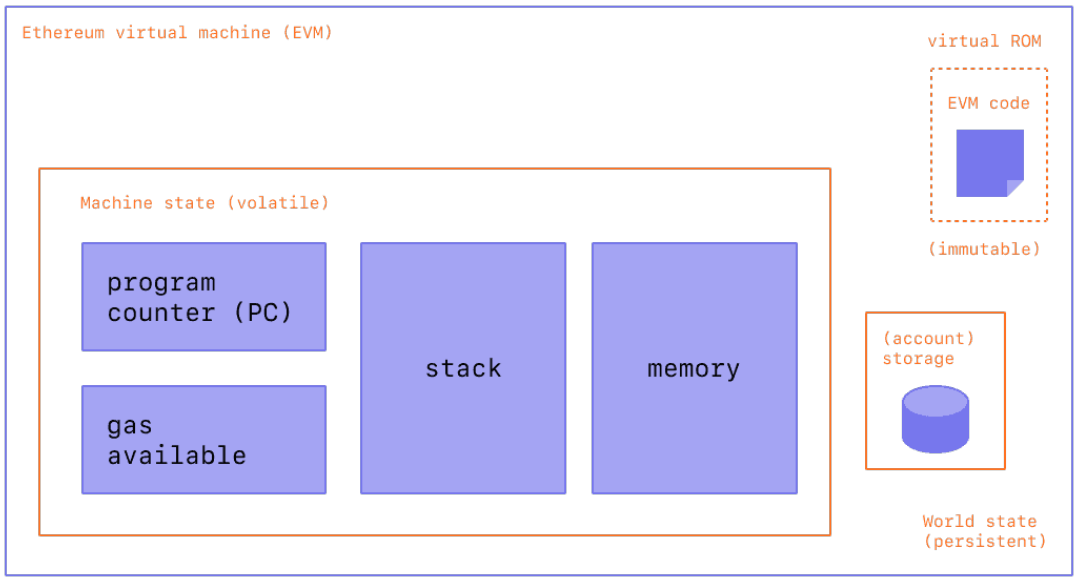

Figure 1.1: Representation of the EVM from *EVM Illustrated* [23]

The EVM is a simple stack-based architecture with a 1024-element stack. The word size is 256 bits in the stack, and in all its memory regions[3], in order to facilitate cryptographic operations such as hashing and elliptic curves. As illustrated in Figure 1.1, the EVM is composed of the following elements:

[2]These and more statistics about Ethereum can be consulted at `https://etherscan.io`.

[3]Volatile memory is both word and byte-addressable. We consider the same word size for all regions for simplicity.

- The program counter (PC).
- A ROM memory containing the code to execute.
- The gas available to perform the operations, a parameter that limits the computation done in a transaction.
- The stack that contains the local values used on the operations.
- A volatile memory region to store data during the execution.
- A persistent memory region called storage, where the contract state is stored.

The EVM executes the bytecode of the compiled contract interpreted as a sequence of opcodes[4], instructions that perform stack-bases operations (e.g., `PUSH`, `ADD`, `PUSH`) and other blockchain and cryptographic related operations such as `BLOCKHASH`, `BALANCE` and `KECCAK256`.

1.2. Gas

The EVM can execute Turing-complete programs. However, it is a quasi-Turing-complete machine since its computation is bounded through a parameter called gas, which limits the amount of computation performed in a single transaction. This computation limit has a security reason. It limits the execution on the EVM and, consequently, in the blockchain, preventing buggy or malicious contracts from executing indefinitely, hanging the network (Denial of Service).

Moreover, this cost model has another essential duty: rewarding the miners. Each unit of gas has an associated price set in *ether*[5] determined by the users that issue the transaction. With this amount of *ether*, miners are rewarded for the computational effort of executing the operation and are incentivized to prioritize certain transactions. Users can set higher gas prices to prioritize transactions or lower prices if they do not mind the transaction taking longer to process. The average gas price of each mined block can be consulted in real-time on platforms such as *Etherscan*.

Along with specifying the gas price, the user that initializes the transaction or contract call specifies the amount of gas that can be destinated to its execution on the EVM. Each EVM instruction has a predefined gas cost[6] withdrawn from the

[4]Read more about EVM opcode at `https://ethereum.org/en/developers/docs/evm/opcodes/`

[5]Usally, gas price is specified in *gwei* that corresponds to 10^{-9} *ether*.

[6]The cas cost of each EVM instruction can be consulted at `https://ethereum.org/en/developers/docs/evm/opcodes/`.

total amount of gas when it is executed. The non-consumed gas is returned to the caller after the function call. However, if the EVM runs out of gas while executing the function call or the execution fails, the contract state reverts, but the gas is not refunded to the caller.

The gas cost mechanism is a fundamental feature of the Ethereum blockchain. It ensures that the network remains secure and economically incentivized for miners. Nevertheless, this mechanism urges the contracts in the network to be as efficient as possible since they have a direct monetary cost on their users.

1.3. Programming Languages

Developers can use several programming languages to develop smart contracts for the Ethereum blockchain. Different languages can target the EVM, such as LLL (Lisp Like Language) [18], one of the first languages developed for Ethereum, or Vyper [24], an experimental language. However, the vast majority of smart contracts in the Ethereum blockchain are coded in one particular language, Solidity. According to Etherscan, the reference analytics platform for Ethereum, this language is used by more than 99% of the contracts used in the blockchain[7].

In this project, we aim our optimizations to reach the largest number of smart contracts possible. Therefore, we focused on the Solidity language, its compilation into EVM bytecode, and its intermediate representation, Yul.

1.3.1. Solidity

Solidity is an object-oriented language that targets the Ethereum Virtual Machine (EVM). It was proposed in 2014 by Gavin James Wood and developed by members of the Ethereum Foundation led by Christian Reitwiessner. It is mainly influenced by C++, but it also has borrowed concepts from other languages such as Python and JavaScript. This similarity with other popular languages has made Solidity an easy language to adopt for developers who want to develop smart contracts.

Solidity is the most popular programming language in the Ethereum blockchain. It is used for developing smart contracts that can be compiled into EVM bytecode to be deployed in the network. In Solidity, four types of programs can be developed[8]:

[7] These and more statistics about smart contracts can be consulted at `https://etherscan.io/dashboards/contract-statistics`.

[8] Read more about contracts, interfaces, and libraries in Solidity at `https://docs.soliditylang.org/en/v0.8.19/contracts.html`.

- **Contracts.** They can be seen as Java or C++ classes. They represent the smart contract deployed at the network with its state variables, functions, and constructor. They are defined using the `contract` keyword, and, as shown in Figure 1.2, they usually have the following structure:

  - Declaration of its state variables that may include a default value (Lines 2 and 3).
  - Constructor, a function declared with the `constructor` keyword only executed when the contract is created (Lines 5 to 7). If not declared, the contract has an implicit constructor with no parameters.
  - Functions of the contract (Lines 9 to 11).

```
1  contract C {
2     uint size;
3     uint[] a;
4
5     constructor(uint _size) {
6        size = _size;
7     }
8
9     function getFirst() public view returns (uint) {
10       return a[0];
11    }
12 }
```

Figure 1.2: Example of Solidity smart contract

- **Abstract contracts.** Contracts that are used as the base to implement other contracts. They contain at least one function that is not implemented, and they cannot be deployed. They are declared using the `abstract` keyword before the `contract` keyword.
- **Interfaces.** Similar to abstract contracts, but cannot provide the implementation of the functions defined. They are declared using the `interface` keyword.
- **Libraries.** Libraries are a special kind of contract that contains reusable code. They are usually deployed only once on the blockchain, and their code is used by other contracts using function calls. They are declared using the `library` keyword.

On Solidity, the following value types can be used:

- **Booleans** with two possible values: `true` or `false`.

- **Integers** that can be signed or unsigned and up to 32 bytes (`uint8`, `uint64`, `uint128`, `uint256`).
- **Fixed point numbers** signed and unsigned and with different sizes[9].
- **Address** containing a 20-byte address of an Ethereum account and some members such as `balance` and `send`.
- **Fixed-size byte arrays** containing up to 32 bytes (`bytes1`, `bytes2`, ..., `bytes32`).
- **Enums** user-defined data types with up to 256 values.
- **Contract types** implicitly defined when a contract is defined.

Additionally, the following reference types can be used:

- **Arrays** containing sequences of values of a specific type. They can have a fixed or dynamic size. Strings and dynamic-size byte arrays are considered special cases of arrays.
- **Array slices**, which are a view of a portion of an array.
- **Structs** user-created containing variables of different data types.

Finally, the **mapping** data type stores and maps key-value pairs. They are defined as `mapping(KeyType => ValueType)`. Mappings are particularly useful in Solidity smart contracts and a great alternative to arrays when storing unordered data that is frequently accessed.

All the mentioned data types, as well as function types, can be used by developers to define their own custom types.

Since Solidity is the language used for developing almost all of the smart contracts for the Ethereum blockchain, across this project, Ethereum smart contracts and Solidity smart contracts will be used as if they were the same thing and as they are in practice.

1.3.2. Yul

Yul is an intermediate language for Ethereum that can be compiled to executable EVM bytecode. It was developed by the Ethereum Foundation to be an intermedi-

[9] According to the Solidity documentation [10], fixed point numbers are not fully supported by Solidity yet.

ate representation of Solidity and other high-level languages that target the EVM. Figure 1.3 shows an example of a Yul program.

```
{
   function power(base:u256, exponent:u256) -> result:u256
   {
      switch exponent
      case 0:u256 { result := 1:u256 }
      case 1:u256 { result := base }
      default
      {
         result := power(mul(base, base), div(exponent, 2:u256))
         switch mod(exponent, 2:u256)
         case 1:u256 { result := mul(base, result) }
      }
   }
}
```

Figure 1.3: Example of Yul program from the Yul documentation [13]

Yul language was designed to be human-readable, provide a simple control flow that eases its analysis, verification, and optimization, and have a straightforward translation into EVM opcodes. Consequently, it is a small language where the only high-level constructions are for-loops, if statements, and switch statements. Furthermore, Yul has a single supported type, the 256-bit type of the EVM (`u256`), and it has no built-in operations, only the operations directly supported by the EVM (e.g. `add`, `mul`, `sstore`). However, it has different dialects[10], as Yul+ [16], that add to the language other types and instructions to target specific platforms.

Yul language is used as an intermediate representation by the Solidity compiler, which uses it to perform high-level optimizations on the code of compiled smart contracts. Additionally, it can be used inside Solidity programs, in the *inline assembly* blocks [11] to have more fine-grained control of the code being executed on the EVM.

This language will have a vital role in our optimization goals as it is currently the target of most compiler optimizations in Solidity. Moreover, Yul allows us to generate optimizations in the code at a low enough level to control what exactly is going to be executed in the EVM but, at the same time, at a high enough level to avoid dealing with the complexity of the stack machine.

[10]Read more about Yul dialects and extensions in `https://dev.to/shlok2740/syntax-and-semantics-of-yul-22pi`

1.4. Data storage

The Ethereum Virtual Machine has three available memory spaces while executing Smart Contracts: stack, memory, and storage.

Stack. The stack is a last-in-first-out memory region used as a working memory for performing all computations. This space, as represented in Figure 1.4, has a maximum size of 1024 256-bit words that store value-type local variables and jump address destinations. When an instruction is executed, the EVM consumes the necessary operands from the top elements of the stack and pushes the result back onto the stack. In addition, the EVM manages the values in the stack using the following instructions:

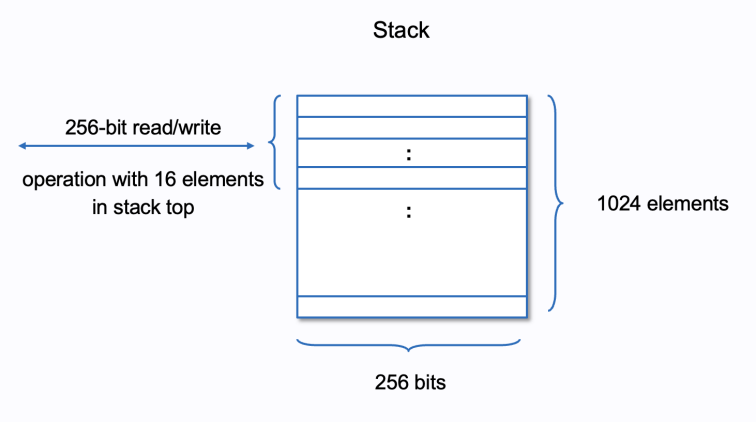

Figure 1.4: Representation of the EVM stack from *EVM Illustrated* [23]

- The `POP` instruction, used to discard and remove the element at the top of the stack.
- The `PUSH` instruction, used to push a value onto the stack. With this instruction, the EVM can push values between 1 and 32 bytes (`PUSH1` - `PUSH32`).
- The `DUP` instruction, used to clone one stack element into the top of the stack. Only the first 16 elements of the stack can be targeted (`DUP1` - `DUP16`).
- The `SWAP` instruction, used to swap one stack element with the top of the stack. Only the first 16 elements of the stack can be targeted (`SWAP1` - `SWAP16`).

All these instructions have a low gas cost of 3 units, except the `POP` instruction which has a gas cost of 2 units. This makes the stack the memory space with the cheapest cost when accessing or modifying its content.

Memory. It is a volatile memory space laid out as a word-addressing byte array that stores reference-type variables. Memory space is created and erased before and after each transaction or call to the contract. On creation, all the array elements have an initial value of 0. Values from memory can be accessed at byte or word (32 bytes) level.

The memory space stores the contents of reference-type variables (e.g., structs, arrays, and strings). Local variables of these types are references to a location in memory and must be defined using the keyword `memory`. Moreover, memory can allocate complex types used as function arguments as well as return values if specified using the same keyword.

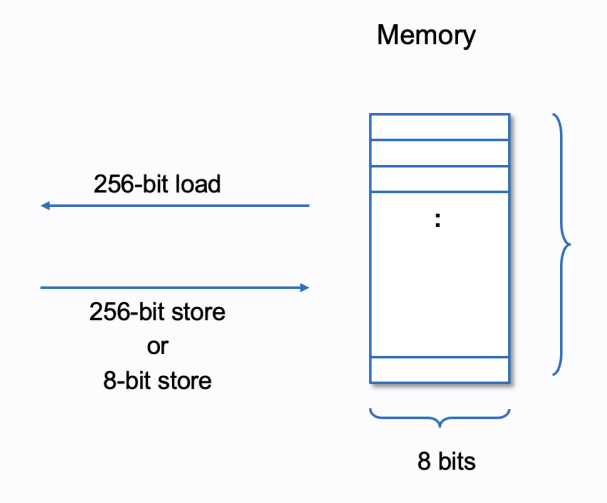

Figure 1.5: Representation of the EVM memory from *EVM Illustrated* [23]

In order to create a new value in memory, the `new` keyword must be used to allocate the corresponding space. In memory, all values are allocated consecutively starting on the address `0x80`[11]. Therefore, only static-sized values can be allocated in memory. All these values disappear when the function call ends.

Regarding gas consumption, the base cost of reading and writing to a memory address is low, with only three gas units. However, there is an extra cost associated with memory expansion. The EVM tracks the highest referenced memory address up to the moment for each transaction. When an instruction expands the memory, i.e., references an address higher than the highest referenced memory address, the corresponding instruction has an extra gas cost proportional to the difference between the old highest referenced address and the new one.

Storage. It is a non-volatile memory space maintained as part of the system state. Each smart contract has an associated storage space where its state information (i.e.,

[11] Memory words `0x00` and `0x20` are reserved for performing hash operations, word `0x40` contains the pointer to the first free memory address, and word `0x60` contains a permanent zero value.

state variables) is stored permanently. Storage data of every smart contract is stored in a state trie, a Merkle tree-like[12] data structure, that represents the state of the entire Ethereum network.

Storage space is created at contract deployment with all the slots set to 0. It is word addressable by 256-bit wide addresses. Therefore, smart contracts can theoretically store up to 2^{256} words (slots) of 256-bits, 2^{261} bytes. Since it is impossible to allocate such an amount of memory to each contract, it is implemented as a sparse array, a key-value store that maps 256-bit words to 256-bit words, as represented in Figure 1.6.

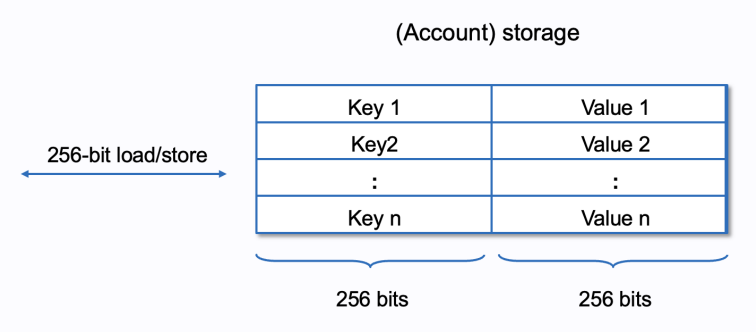

Figure 1.6: Representation of the EVM storage from from *EVM Illustrated* [23]

The slots allocation differs between fixed-sized values and dynamically-sized values:

- Fixed-sized values are allocated in consecutive locations reserved when the contract is deployed. Therefore, if three integer contract state variables `a`, `b` and `c` are defined, they are assigned the slots `0x0`, `0x1` and `0x2`.

- Dynamically-sized values cannot be allocated at contract deployment. Since their size is unpredictable, they can not be allocated consecutively at contract deployment. For these values, the EVM reserves a single slot as if they were statically sized values but saves its contents starting at a different slot, calculated using the Keccak-256 hash of the initially reserved slot address. The originally allocated slot in the case of dynamic-sized arrays is used to store the array length, while in mappings, it is unused.

Figure 1.7 illustrates the storage allocation in a smart contract with three variables defined in the following order: an unsigned integer `A`, an array `arr`, and another unsigned integer `B`.

[12]See more about the Merkle Patricia Tree at: `https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/`.

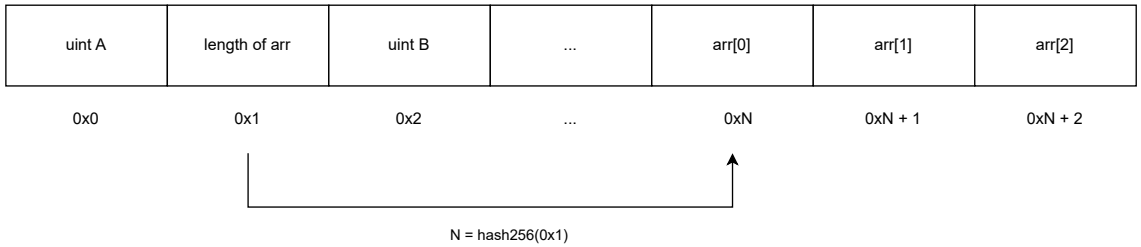

Figure 1.7: Storage allocation of smart contract state variables

It is also important to mention that it is possible to declare local pointers to storage values inside function bodies using the keyword `storage`, similar to how we do memory pointers.

Reading (`SLOAD`) and writing (`SSTORE`) to storage have a very high gas cost. In the Berlin fork, according to EIP-2929 [8], `SLOAD` has a cost of 2100 units on the first access to the slot in the current transaction and 100 units otherwise. In the case of `SSTORE`, the gas consumption is more complex. Its cost depends on whether the slot has been previously accessed, the previous value, and the new value. It can vary from 22100 gas units in the worst case scenario to even a refund of 19900 gas units[13]. The model penalizes when a slot is set from a zero to a non-zero value because it has to be maintained in the storage and broadcast over the blockchain network. Conversely, the model rewards setting slots to zero as it reduces the number of slots that need to be persistently stored.

Calldata. Finally, there is a fourth memory space that is not explicitly mentioned on the *Yellow paper* [25] but is described as the *Execution Enviroment*. This memory space is called the calldata, and it is a read-only and byte-addressable space that contains the parameters of the call or transaction being executed as well as environment information (e.g., address of the called, ether sent, and available gas). Additionally, this space can store the return parameters of the function calls. From this immutable space, the EVM can consult its size with instruction `CALLDATASIZE`, copy a single value using `CALLDATALOAD` to the stack and copy a region to memory using `CALLDATACOPY`. These read operations have a small gas cost comparable to the stack gas cost.

1.5. Arrays access in Ethereum

In this work, we will focus on optimizing array accesses in Solidity smart contracts for the Ethereum blockchain. Therefore, knowing how arrays work in Ethereum, and specifically in the Solidity language, is essential.

[13]See EIP-2929 [8] for more details.

The array is a reference type containing elements of a specific type. In Solidity, programmers can define arrays of any value, reference, mapping, or function type. They can be located in memory, calldata, or storage and can have a static or dynamic size.

```
1  contract C {
2     uint[4] arrayA;  // Fixed-size array
3     uint[] arrayB = [1,2];  // Fixed-size array
4     uint[] arrayC;  // Dynamic array
5
6     function f(uint size, uint[] calldata callArray, uint[5] calldata
          callArrayB) public {
7        uint[5] memory memArrayA;
8        int[] memory memArrayB = new int[](size);
9        bool[] memory memArrayC;
10       memArrayC = new bool[](size);
11
12       for (uint i = 0; i < size; i++)
13       {
14          arrayC.push(i);
15          arrayC.pop();
16       }
17
18       uint length = arrayC.length;
19       value = arrayC[0];
20       uint[] memory slice = callArray[1:4];
21    }
22 }
```

Figure 1.8: Example of arrays in Solidity

Memory. Arrays in memory are created inside the body of a function using a local variable pointer. They are declared in two different ways, depending on how their size is set. In function `f` (Lines 6 to 21 of Figure 1.8), we can observe the different ways of declaring a memory array. Arrays allocated in memory always have a fixed size that can be set statically (Line 7) or dynamically (Line 8 or Lines 9 and 10). Like any memory value, they are allocated on creation, starting at the first free memory position, where the length is stored, and consecutively storing all the array values. In the case of dynamic size arrays, their local variable (Line 9) initially points to the zero memory slot (`0x60`) while they are not initialized (Line 10).

Calldata. Arrays in calldata are only readable and are used as arguments or return values of contract functions. Calldata arrays can have a static or dynamic immutable

size. If they have a dynamic size, their lengths are calculated based on the range of addresses it comprises in the calldata region. Line 6 of Figure 1.8 shows the declaration of a dynamic and a static calldata arrays as function arguments `f`.

Storage. Arrays allocated in storage are created as contract state variables with a fixed or dynamic size. In the contract of Figure 1.8, we can observe the two ways of declaring a fixed-size array in storage: without initializing its values (Line 2) or initializing its values (Line 3). Additionally, at Line 4, we can see the declaration of a dynamic array. Dynamic arrays can only be declared in storage and can perform the `push` and `pop` operations in order to append or remove a value at the last position of the array.

In Solidity, arrays can be accessed in three different ways:

- **Length access.** The number of elements of the array can be consulted using the `length` member of the array pointer. At Line 18 of Figure 1.8, we can observe an example of length access to an array.
- **Index access.** The elements of the array can be accessed using their index (`<array base>[index]`), as shown at Line 19 of Figure 1.8.
- **Index Range access.** A subset of continuous elements of the array can be obtained using their index range (`<array base>[from: to]`), as shown at Line 20 of Figure 1.8. Index range accesses are only available for dynamic calldata arrays.

In Solidity smart contracts, an out-of-bounds check is always performed when accessing an array by index. This check compares the index being accessed with the array length, raising an exception if the index is not lower. Out-of-bounds checking helps to provide safer code, avoiding overflows on array accesses. However, they also have a negative impact on gas consumption. When accessing memory or storage arrays with a dynamically set length (dynamic arrays or fixed-size arrays with size set at runtime), their length must be loaded in order to perform the bounds check. Therefore, on each access, two loads have to be done: the length of the array and the value being accessed. This makes index accesses very expensive, especially in the case of storage arrays. The length load is not required for fixed-size arrays with statically set sizes since their length is known at compile time. There is also a gas consumption increase for all array indexes accessed due to all the other instructions performed during the check.

In this project, we will propose two different optimizations for the array index access to reduce its gas consumption by avoiding the length load needed for the out-of-bounds checks.

Chapter 2

The Solidity Compiler

In order to define a compiler, we will quote the definition given by the classic reference book on compiler technology, *Compilers: principles, techniques, and tools* [1] popularly known as the *Dragon Book*:

> *A compiler is a program that can read a program in one language, the source language, and translate it into an equivalent program in another language, the target language.* [1, p. 1]

The compilation of a program is a complex process composed of different phases. The traditional phases of the compilation process, as presented in the *Dragon Book* and shown in Figure 2.1, are:

- **Lexical Analysis.** It is the first phase of the compilation process. During this phase, the compiler processes the source program as a stream of characters, grouping them into lexemes stored as tokens.

- **Syntax Analysis or Parsing.** The compiler generates a tree structure representing the structure of the obtained tokens following the source language grammar. This structure is called the *syntax tree*.

- **Semantic Analysis.** The compiler checks that the code complies with the semantic rules of the source language.

- **Intermediate Code Generation.** This optional phase is where the compiler may generate an intermediate representation (IR) from the *syntax tree*.

- **Code Optimization.** During this phase, the compiler transforms the intermediate code in order to be able to generate a more efficient target code. The optimizations in this phase are machine-independent since they transform the intermediate representations without involving specific details of the targeted

machine as registers or memory allocation.

- **Code Generation.** The compiler maps the intermediate representation (or the *syntax tree* in case the IR was not generated) into the target code. During this phase, the compiler is in charge of the register allocation of the generated variables or their position on the stack in the case of stack-based machines (as the EVM).

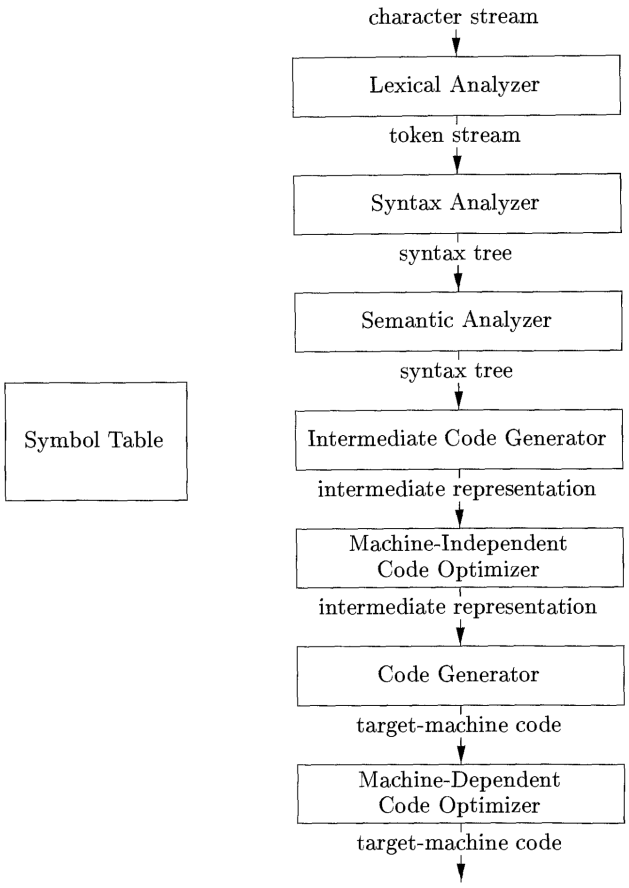

Figure 2.1: Phases of the compilation process from the *Dragon Book* [1]

Finally, the target code generated may be subject to machine-dependent optimizations that transform the code to achieve a better performance based on features specific to the targeted machine.

In this chapter, we will introduce the compilation of Solidity smart contracts into the final bytecode executed by the *Ethereum Virtual Machine*. As we will see, the compiler process resembles the traditional phase division previously presented, with minimal variations. This explanation of the Solidity compiler will mainly focus on the compiler features and phases that are modified or extended in the implementation of our proposed optimizations and which will be mentioned in later chapters.

2.1. Phases of the Solidity Compiler

The Solidity compiler is the software that reads a program in Solidity and translates it into an equivalent program in EVM bytecode. The compiler was released in 2014 with the publication of the Solidity language by the Ethereum Foundation. It is an open-source compiler that can be consulted in its GitHub repository [15], where community contributors help to develop and improve it, led by the Ethereum Foundation official developers.

The Solidity compiler is constantly evolving to adapt itself to the changes in the EVM and the Solidity language. At the moment of this project, the Solidity compiler is in version 0.8.19.

The compiler, written in C++, is a complex program consisting of several modules that handle different phases of the compilation process. Figure 2.2 illustrates the main phases of a Solidity smart contract compilation.

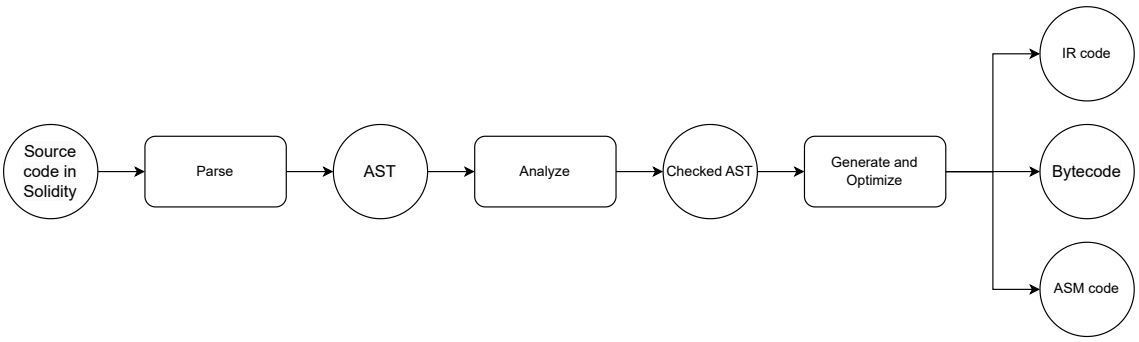

Figure 2.2: Phases of the Solidity compiler

In this section, we will review the most relevant compiler phases and modules, the understanding of which is critical to the optimizations we will perform in both the Solidity language and the compiler.

2.1.1. Parsing

Parsing is the first phase of the compilation process of a Solidity smart contract. During this phase, the compiler generates an internal representation of the code containing all the information included in the source files. This internal representation, called Abstract Syntax Tree (AST), will be used in the following phases to interpret, analyze and generate the EVM bytecode.

2.1.1.1. Abstract Syntax Tree

The Abstract Syntax Tree (AST) is a hierarchical representation of the code containing the information needed to analyze the source code and generate the machine-readable code. During this phase, the compiler generates an AST for each source file.

```
1 pragma solidity >=0.8.4;
2
3 contract C {
4    uint[] a;
5
6    function f() public view returns (uint) {
7       return a[0] + 2;
8    }
9 }
```

Figure 2.3: Example of Solidity smart contract

We will use the minimal code shown in Figure 2.3 as a running example for this section. The AST generated by the compiler when compiling the code in Figure 2.3 is represented in Figure 2.5.

Every AST node of the Solidity compiler has the following attributes:

- An identifier of the AST node that is unique.
- The location in the source file.
- An `Annotation` object that contains information about the object type and other annotations.

Apart from those attributes, all AST nodes have the following methods:

- The `==` and `!=` operator functions that check if two nodes have the same id.
- An `accept` function to be traversed by a visitor (checkers or code generators).

The AST of a given source file can be exported as a JSON using the *–ast-compact-json* command line option of the current compiler. In Figure 2.4, we can see part of the generated JSON representing AST of the add expression in the function defined in Figure 2.3.

```
"expression":{
 "commonType":{"typeIdentifier":"t_uint256","typeString":"uint256"},
 "id":15,
 "isConstant":false, "isLValue":false, "isPure":false, "
    lValueRequested":false,
 "leftExpression":{
 "baseExpression":{
 "id":11, "name":"a",
 "nodeType":"Identifier",
 "src":"154:1:0",
 "typeDescriptions":{"typeIdentifier":"
    t_array$_t_uint256_$dyn_storage","typeString":"uint256[] storage
     ref"}
 },
 "id":13,
 "indexExpression":{
 "id":12, "name":"i",
 "nodeType":"Identifier",
 "src":"156:1:0",
 "typeDescriptions":{"typeIdentifier":"t_uint256","typeString":"
    uint256"}
 },
 "isConstant":false, "isLValue":true, "isPure":false,
 "lValueRequested":false,
 "nodeType":"IndexAccess",
 "src":"154:4:0",
 "typeDescriptions":{"typeIdentifier":"t_uint256","typeString":"
    uint256"}
 },
 "nodeType":"BinaryOperation",
 "operator":"+",
 "rightExpression":{
 "id":14,
 "isConstant":false, "isLValue":false, "isPure":true,
 "kind":"number", "lValueRequested":false,
 "nodeType":"Literal",
 "src":"161:1:0",
 "typeDescriptions":{"typeIdentifier":"t_rational_2_by_1","typeString
    ":"int_const 2"},
 "value":"2"
 },
 "src":"154:8:0",
 "typeDescriptions":{"typeIdentifier":"t_uint256","typeString":"
    uint256"}
}
```

Figure 2.4: Example of AST Expression node represented as JSON

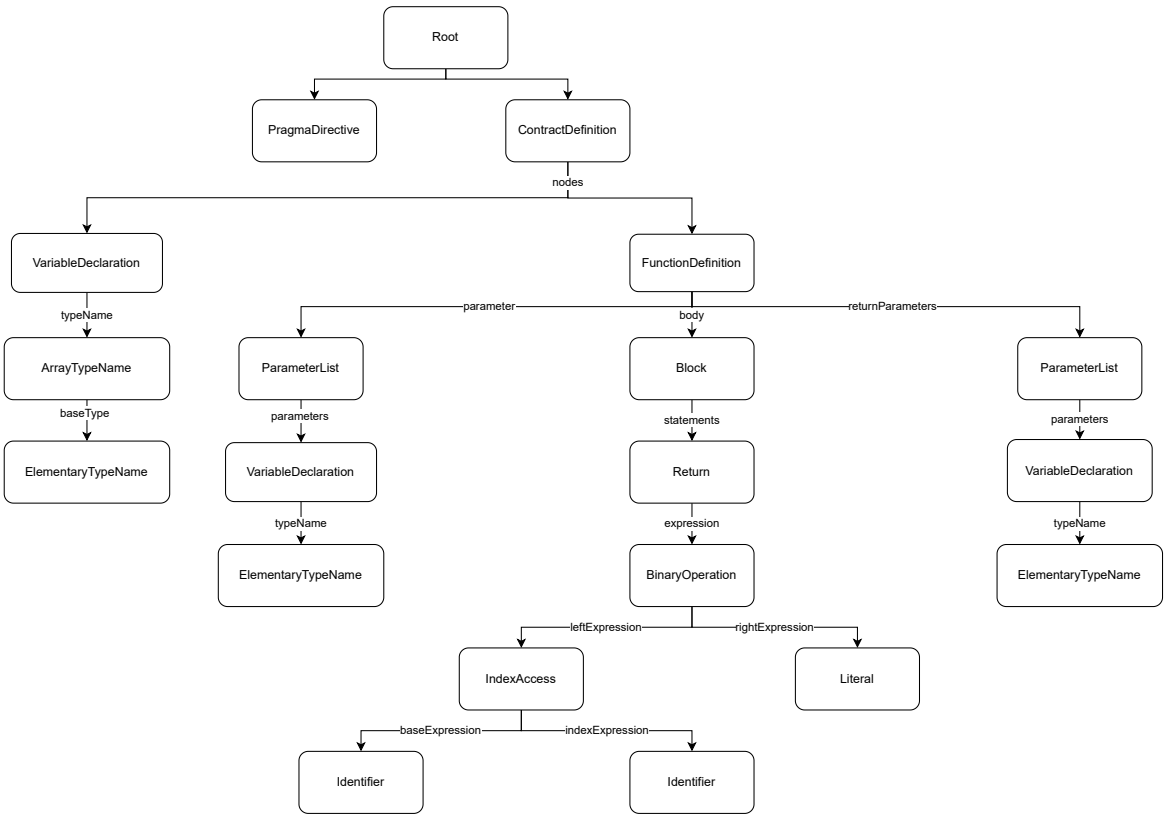

Figure 2.5: Example of the AST of a contract

Visitor pattern. The compiler implements the Visitor Pattern [17] in order to allow the different analyzers and code generators to traverse the AST. In order to implement this pattern, classes `ASTVisitor` and `ASTConstVisitor`[1] are defined, providing a default implementation of the methods to visit each kind of AST Node, as shown in Figure 2.6.

The compiler analyzers and code generators, which will be discussed in later sections, inherit from either the `ASTVisitor` or `ASTConstVisitor` classes. Each of them overwrites the `visit` methods to perform its corresponding actions over the source code elements represented by each AST node. The use of this pattern makes the AST fully modular and simplifies the extension and modification of the AST nodes, as well as the extension of every module that works over the AST (i.e., type checking or IR generation).

[1]Source code available at `https://github.com/javierSande/solidity/blob/develop/libsolidity/ast/ASTVisitor.h`

```
virtual bool visit(Block& _node) { return visitNode(_node); }
virtual void endVisit(Block& _node) { endVisitNode(_node); }

/// Generic function called by default for each node, to be
   overridden by derived classes
/// if behavior unspecific to a node type is desired.
virtual bool visitNode(ASTNode&) { return true; }
/// Generic function called by default for each node, to be
   overridden by derived classes
/// if behavior unspecific to a node type is desired.
virtual void endVisitNode(ASTNode&) { }
```

Figure 2.6: Example of default methods used by the AST visitors

Relevant AST nodes. While over 60 diverse types of AST nodes are used to represent different syntactic elements of source code, it is beyond the scope of this document to describe all of them in detail. Instead, we will focus on the most pertinent types for our compiler modifications.

- **Statement node.** The AST `Statement` node is one of the most basic types of nodes in the Solidity AST. It serves as a base to represent a wide range of different code statements, such as variable declarations, function calls, or control flow statements (if-else or loop constructions). Each possible code statement is represented by a corresponding AST node that inherits from the `Statement` node. Figure 2.7 shows an example of a statement that represents a return statement (`Return` class).

```
contract C {
   uint[] a;

   function f() public view returns (uint) {
      return a[0] + 2;
   }
}
```

Figure 2.7: Example of statement

- **Block.** The AST `Block` node is one of the most relevant AST nodes of the compiler in the context of this project. A block is a type of statement containing a group of zero or more code statements enclosed within curly braces. It represents diverse structures such as the body of contracts, functions, if-else statements, or loops. Additionally, blocks can also contain other nested blocks.

In addition to the common attributes shared by all AST nodes, blocks contain a list of child nodes representing code statements. Moreover, they contain information pertinent to a particular type of block, the `UncheckedBlock`, which is explained in Section 3.1 of Chapter 3. Figure 2.8 shows an example of a block forming the body of a function.

```
1 contract C {
2    uint[] a;
3
4    function f() public view returns (uint){
5       return a[0] + 2;
6    }
7 }
```

Figure 2.8: Example of block

- **Expression.** The AST `Expression` node is another basic node in the Solidity AST. It represents a code statement that can be evaluated to produce a value. There are many types of expressions in Solidity. Some of them are simple and cannot be divided (known as `PrimaryExpressions`), such as literals or identifiers, and others are more complex and are composed of other expressions. That is the case of unary or binary operations, function calls, tuples, or index accesses. Each of them is represented by a corresponding AST node that inherits from the `Expression` node and contains all the information about the particular type of expression they represent. Figure 2.9 shows an example of an expression that performs a binary operation between two values (`BinaryOperation` class).

```
1 contract C {
2    uint[] a;
3
4    function f() public view returns (uint) {
5       return a[0] + 2;
6    }
7 }
```

Figure 2.9: Example of expression

- **Index Access.** The AST `IndexAccess` node represents an index access expression in Solidity, commonly used to retrieve values from mappings and arrays. The index access node is a straightforward node containing two child expression nodes in addition to the attributes common to all AST nodes. These two child nodes represent the array or mapping base being accessed and the

index of the value to be retrieved. Understanding how this node is generated in later compiler phases is essential for achieving our array access optimization goals. Figure 2.10 shows an example of index access over a storage array in Solidity.

```
1 contract C {
2    uint[] a;
3
4    function f() public view returns (uint) {
5       return a[0] + 2;
6    }
7 }
```

Figure 2.10: Example of index access

- **Inline Assembly.** The AST `InlineAssembly` node represents a block of Yul code nested inside a Solidity code block, as seen in Figure 2.11. Inline assembly blocks give programmers a more fine-grained control of the code being executed on the EVM, allowing them, for example, to point direct slots on storage or memory or to bypass security checks. As a consequence of this higher control, the use of Yul code allows developers to create code with reduced gas consumption while requiring them to create more complex code. This is especially interesting when developing libraries since other developers frequently reuse them. Figure 2.11 shows the usage of an inline assembly block in a simple library.

```
1  library GetCode {
2     function at(address addr) public view returns (bytes memory code)
          {
3        assembly {
4           let size := extcodesize(addr)
5           code := mload(0x40)
6           mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f),
7           not(0x1f))))
8           mstore(code, size)
9           extcodecopy(addr, add(code, 0x20), 0, size)
10       }
11    }
12 }
```

Figure 2.11: Example of inline assembly block from Solidity documentation [11]

2.1.2. Analysis

Once the parser creates the AST, the compiler analyzes the syntax tree to get more information (i.e., types of expressions, inheritance, or external calls) and checks that the syntax and types used in every statement of the contract are correct. In this phase, different modules of the compiler take place, each of them in charge of checking different aspects of the code represented in the AST. Figure 2.12 shows the different analysis and checking phases.

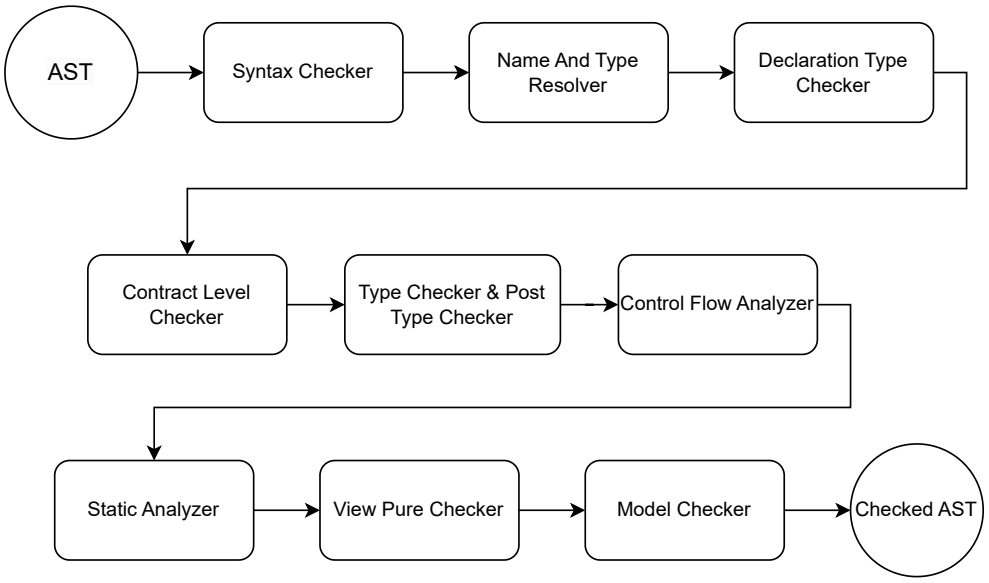

Figure 2.12: Analysis phases of the Solidity compiler

Syntax Checking. The `SyntaxChecker` module is in charge of performing the syntax checking of the AST. This module checks that the AST and all its nodes follow the syntax rules of the Solidity language. For example, it checks that `continue/break` statements are placed inside a loop or that the definition of a struct is not empty. Additionally, during this phase, the compiler issues warnings about the usage of language features that are deprecated or experimental.

Type Checking. In the Solidity compiler, the type checking of the AST is performed in different phases. Each phase is in charge of checking the types at different levels in the code, and it is performed by a different module. The four main modules in charge of the type checking in Solidity are:

- `NameAndTypeResolver`: registers all the variable declarations in each scope and resolves the name references. For each variable reference, it checks if the variable exists in the current scope and adds a reference to the variable declaration to the AST node.

- `DeclarationTypeChecker`: assigns types to declarations.
- `TypeChecker`: checks the applicability of operations based on the operand types and issues errors for invalid operations.
- `PostTypeChecker`: performs checks that can only be done when all types of all AST nodes are known, such as circular references in constant variables.

Contract Level Checking. The `ContractLevelChecker` verifies overloads, abstract contracts, function clashes, and other checks at contract or function level. This module checks, for example, that there are no duplicated functions, that the contract constructor is correctly provided, or that the library is defined correctly (it does not inherit and has no constant state variables).

Control Flow Analyzer. The `ControlFlowAnalyzer` module generates and analyzes the Control Flow Graph (CFG) of the code. This module looks for unreachable code and uninitialized variable accesses.

Static Analysis. The `StaticAnalyzer` module performs a static analysis of the code, issuing warnings that can help programmers write clean code. These warnings include unused local variables, statements with no effect, or division by zero operations.

Mutability Checking. In Solidity, functions are defined with a state mutability modifier that defines how the function interacts with the data stored in the contract. The `ViewPureChecker` module checks that the state mutability of each function is correct. The four possible state mutability identifiers, ordered from most to least restrictive, are :

- **Pure**: functions that do not read or modify the state of the blockchain. These functions can only call pure functions and cannot read from or write to state variables. Pure functions are declared with the `pure` keyword.
- **View**: read-only functions that do not modify the state of the blockchain. They cannot contain code that modifies state variables, receive or send ether, nor call functions that are not pure or view. View functions are declared with the `view` keyword.
- **Non-Payable**: default function state mutability, assumed by the compiler for every function not explicitly defined as pure, view, or payable. Non-payable functions can read and modify state variables but cannot send or receive ether.

- **Payable**: functions that can send or receive ether. Payable functions are declared with the `payable` keyword. If a function is not declared as payable, any attempt to send or receive ether will be rejected.

During this phase, the `ViewPureChecker` module issues warnings in case the state mutability of the function can be restricted (i.e., there is no write to storage and can be restricted to `view`) or errors when the declared state mutability is not compatible (i.e., a state variable is accessed inside a `pure` function).

Model Checking. The `ModelChecker` module is the entry point to the SMT solvers [12]. As explained in its documentation, during this phase, the module automatically tries to prove that the code satisfies the specification given by `require` and `assert` statements. This optional phase is activated by selecting the solver engine (BMC or CHC) using the *–model-checker-engine* command line option.

2.1.3. Code generation

During the code generation phase, the compiler generates the binaries of the smart contract that will be executed by the EVM both to deploy and execute contracts in the blockchain. This phase is divided into smaller steps that generate different representations of the code at different levels, being the bytecode the lowest level representation of the binaries in hexadecimal. Additionally, some optimization steps that will be explained in Section 2.2.1 can be performed at different representation levels in order to achieve an optimized final bytecode. Figure 2.13 represents the code generation process with the different modules and optimization options.

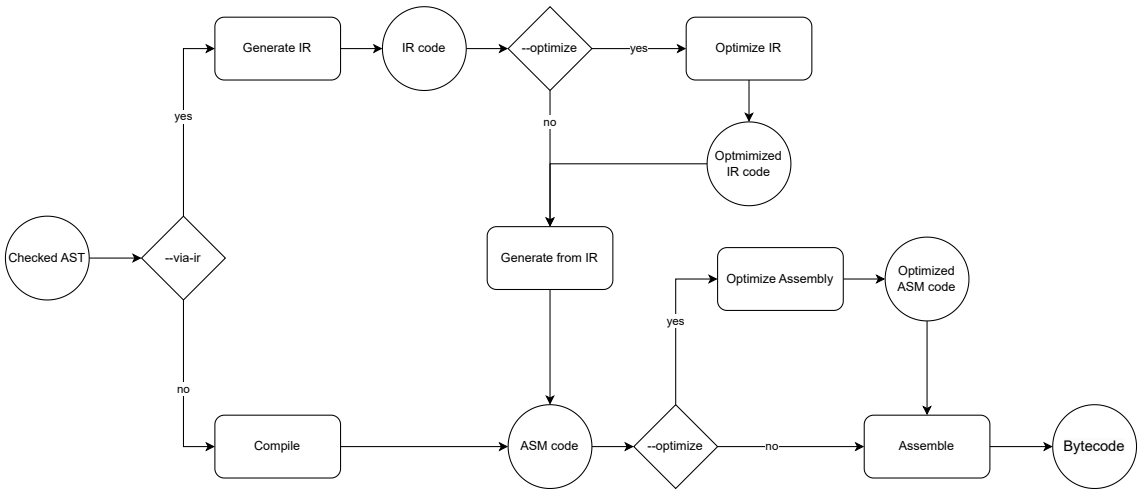

Figure 2.13: Code generation phases of the Solidity compiler

All of the different representations of the code can be obtained from the compiler using different command line arguments[2], being *–bin* the flag needed to generate the contract binaries (bytecode). For each of the possible representations, the compiler generates a contract creation code, containing the code needed to deploy the contract on the blockchain, and a runtime code, containing only the code that is stored on-chain and executed on each call to the deployed smart contract.

2.1.3.1. IR generation

The Solidity compiler can generate an Intermediate Representation (IR) of the code. This intermediate representation uses the Yul language and provides a low-level representation of the original Solidity code but at a higher level than the EVM assembly code (ASM). The IR is only generated when requested (*–ir* or *–ir-optimized* flags) or when the compiler generates the ASM, and consequently, the bytecode from the generated IR (*–via-ir* flag).

In order to generate the IR code, the compiler uses the `IRGenerator` class, which translates the Solidity code, represented by the AST, into Yul code. This class, and the `IRGeneratorForStatements` at statement level and below, translates the Solidity code using templates of Yul functions. Figure 2.14 shows an example of a function template used to transform a memory array index access into Yul.

```
return m_functionCollector.createFunction(functionName, [&]() {
   return Whiskers(R"(
      function <functionName>(baseRef, index) -> addr {
         if iszero(lt(index, <arrayLen>(baseRef))) { <panic>() }
         let offset := mul(index, <stride>)
         <?dynamicallySized>
         offset := add(offset, 32)
         </dynamicallySized>
         addr := add(baseRef, offset)
      }
   )")
   ("functionName", functionName)
   ("panic", panicFunction(PanicCode::ArrayOutOfBounds))
   ("arrayLen", arrayLengthFunction(_type))
   ("stride", to_string(_type.memoryStride()))
   ("dynamicallySized", _type.isDynamicallySized())
   .render();
});
```

Figure 2.14: Example of template of Yul function

[2]Read more about the Solidity output options in `https://docs.soliditylang.org/en/v0.7.4/using-the-compiler.html`

All the generated code is streamed into a single string variable that is later parsed into a Yul AST[3] to be analyzed. Finally, if the *–optimize* flag is set, the IR code is optimized using the Yul optimization module explained in Section 2.2.1.2.

2.1.3.2. EVM Assembly generation

The generation of EVM assembly code, or ASM code, is the previous step to the final binaries generation. The assembly code is a human-readable low-level representation of the instructions that the EVM will execute. ASM code can be generated directly from the Solidity source code or from the generated IR code (*–via-ir*).

Generate from source code. By default, the assembly code of the smart contracts is generated directly from the Solidity source code, using the AST generated at the parsing phase and completed during the analysis phase. In order to transform the code structures into assembly code, the compiler uses two different methods depending on the complexity and usage frequency of the structure:

- Transform statements directly to assembly code, as shown in Figure 2.15.

```
1  m_context << dupInstruction(1 + _stackDepth);
2  switch (_arrayType.location())
3  {
4  case DataLocation::CallData:
5     // length is stored on the stack
6     break;
7  case DataLocation::Memory:
8     m_context << Instruction::MLOAD;
9     break;
10 case DataLocation::Storage:
11    m_context << Instruction::SLOAD;
12    if (_arrayType.isByteArrayOrString())
13       m_context.callYulFunction(m_context.utilFunctions().
            extractByteArrayLengthFunction(), 1, 1);
14    break;
15 }
```

Figure 2.15: Compiler code excerpt that illustrates the direct transformation to assembly

[3]The Yul Abstract Syntax Tree is a syntax representation of the Yul code similar to the one used for Solidity code, but much simpler with only 18 types of nodes. See `https://github.com/ethereum/solidity/blob/28593839d9913a740e9c8514d2ba607241d54398/libyul/AST.h` for more information

- Use predefined function templates written in Yul, as in the IR generation, and generate the corresponding assembly code. This method is used to transform complex internal operations such as copying an array from storage to memory, as in the example shown in Figure 2.16. The predefined Yul templates are also used for functions in charge of ABI[4] encoding, decoding, and type conversions.

```
m_context.callYulFunction(m_context.utilFunctions().
    copyByteArrayToStorageFunction(_sourceType, _targetType), 3, 0);
```

Figure 2.16: Compiler code excerpt that illustrates the usage of predefined Yul function

Generate from IR. The EVM assembly generation from the IR code (Yul) is relatively straightforward. Since both representations operate at a similarly low level, each Yul statement can be translated into only a few assembly instructions. However, when generating the assembly code, the compiler has to track the evolution of the stack size and the position of each variable on it, making this process highly complex.

In order to perform the code transformation, the compiler uses the previously generated AST of the IR, visiting each node and appending the transformations into an `Assembly` object containing the final assembly code. Figure 2.17 shows an example of a function used to transform an `if` structure in Yul to assembly code.

```
void CodeTransform::operator()(If const& _if)
{
   visitExpression(*_if.condition);
   m_assembly.setSourceLocation(originLocationOf(_if));
   m_assembly.appendInstruction(evmasm::Instruction::ISZERO);
   AbstractAssembly::LabelID end = m_assembly.newLabelId();
   m_assembly.appendJumpToIf(end);
   (*this)(_if.body);
   m_assembly.setSourceLocation(originLocationOf(_if));
   m_assembly.appendLabel(end);
}
```

Figure 2.17: Compiler code excerpt that illustrates the transformation from IR to assembly

[4]See more about the ABI specification at `https://docs.soliditylang.org/en/v0.8.19/abi-spec.html`

2.1.3.3. Bytecode generation

The bytecode generation is the last phase of code generation in the compiler. In this phase, the assembly code is linked and assembled into the creation and runtime bytecode. The generated bytecode is the lowest-level language produced by the compiler, the hexadecimal representation of the EVM instructions, as the example shown in Figure 2.18.

6080604052348015600f57600080fd5b5060043610602857600035 60e01c806326121ff014
602d575b600080fd5b60336047565b604051603e91906072565b60405180910390f35b60
006002600546056919060ba565b905090565b60008190509190505565b606c81605b565b
82525050565b60006020820190506085600083018460655 65b92915050565b7f4e487b71
0000000000000000000000000000000000000000000000000000000000006000526011600452
60246000fd5b600060c382605b565b915060cc83605b565b925082820190508082111560
e15760e0608b565b5b9291505056fea2646970667358221220148429967791ea8a3f39c34
3c7247dc5662860f0c6e71d1c1440a899503ee0ac64736f6c637827302e382e32302d63692
e323032322e31302e382b636f6d6d69742e6537643036366534 2e6d6f640058

Figure 2.18: Runtime bytecode of the contract of Figure 2.3

In addition to generating EVM bytecode as a sequence of hexadecimal opcodes, the Solidity compiler can also output a human-readable representation of the EVM instructions. This output can be generated using the command-line option *–opcodes*. Figure 2.19 shows an example of the opcode representation of the bytecode.

```
1 PUSH 80
2 PUSH 40
3 MSTORE
4 CALLVALUE
5 DUP1
6 ISZERO
7 PUSH 0x0f
8 JUMPI
9 PUSH 0
```

Figure 2.19: Opcodes representation of the first instructions on Figure 2.18

2.2. Code optimizations

During the code generation, compilers try to generate a target code as efficient as possible. In order to do that, the compiler transforms the code into a new, more efficient code. However code optimization is not a trivial task, as explained on the following paragraph extracted from the *Dragon Book* [1]:

> *The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.* [1, p. 505]

There is a great variety of different code optimizations that compilers can perform, targeting different levels of abstraction and targeting different objectives, such as time efficiency, memory efficiency, or target code size. We can distinguish two main types of code optimizations on compilers:

- **Machine-independent code.** These optimizations are performed at an optimization phase prior to the target code generation, where the compiler transforms the IR into a new IR code from which a more efficient code can be generated. At this level, the optimizations are not tied to specific features of the targeted architecture, such as register or memory allocation.

- **Machine-dependent code.** These optimizations are performed over the target code at the end of the code generation process. At this level, optimizations are tied to the specific features of the targeted architecture, such as the stack allocation on the EVM.

In the Ethereum blockchain, the efficiency of smart contracts is critical. The execution of each contract call has an associated gas cost which translates directly into an economic cost for the user. In a network where hundreds of thousands of contract calls are performed daily, reducing the execution cost of contract calls is crucial for developers, contract owners, and clients, especially when dealing with large-scale smart contracts. Accordingly, there is a high interest in generating the most efficient code possible regarding gas consumption, and the code optimization process for any code targeting the EVM is subject to intense research.

In this section, we will discuss the state of the art on code optimizations in Ethereum. On the one hand, we will introduce the two optimization modules in the Solidity compiler, which respectively perform transformations on the IR and generated code in order to output the most efficient code possible. On the other

hand, we will cover some proposed work on pre and post-generation optimizations for code targeting the EVM.

2.2.1. Solidity compiler optimizations

The Solidity compiler has two different optimizer modules that optimize the generated code in order to make it more efficient in terms of execution cost and code size. The optimizer modules of the compiler operate at two different levels. The 'old' module operates at the opcode level and focuses on performing small transformations on the generated code in order to improve its efficiency. In contrast, the 'new' optimizer operates at the IR code level and plays the role of the machine-independent code optimization module transforming the IR during the code generation process.

On the current compiler version, both optimization modules are disabled by default and can be activated using the command line parameter *–optimize*. Additionally, we can use the parameter *–optimize-runs* to indicate an approximate number of times the contract is expected to be executed across its lifetime. This expected number of executions allows developers to establish on the optimizer a tradeoff between the code size, which affects the deployment cost, and the execution gas cost once deployed. In a contract that will be used only a few times, the compiler will prioritize producing a shorter code over the execution cost reduction. In contrast, for contracts that will be executed many times, the optimizer will generate more efficient code without caring about the length of the final optimized code. However, an optimized contract will probably consume less gas for deployment as well as for function calls.

2.2.1.1. Opcode optimizer

The opcode optimizer was the first code optimizer implemented in the Solidity compiler. It operates at the opcode (bytecode) level applying simplification rules and removing unused and duplicated code. The opcode optimization takes place at the end of the code generation process, transforming the generated bytecode into a more efficient bytecode. Therefore, this module fits into the definition of a machine-dependent code optimizer since it works directly on the target code instead of the IR.

The optimizer divides the sequence of instructions on blocks delimited by JUMP instructions. Then it analyzes the instructions inside those blocks, keeping track of the stack, memory, or storage modifications. Finally, it applies several optimization phases in a loop until no optimization is possible. The optimizations applied are:

- `FullInliner`: replaces jumps to blocks containing simple instructions with a copy of the instructions in the block.

- `JumpdestRemover`: removes unused `JUMPDEST` instructions and their referenced tags.
- `PeepholeOptimiser`: optimizes small windows of instructions, replacing them with a more efficient sequence of instructions that produces the same result.[5]
- `BlockDeduplicator`: unifies duplicated blocks.
- `CommonSubexpressionEliminator`: finds and combines equal expressions.
- `ConstantOptimiser`: replaces constant expressions by their computed values at compile time.

The opcode optimizer module can be very effective at applying simple optimizations to the bytecode, reducing the gas consumption of a contract. However, it has some limitations. Because it operates on a very low level, it has limited information and understanding of the code. It can optimize small sets of bytecode instructions but cannot perform optimizations for high-level structures. Furthermore, because the low-level code is exceptionally complex to interpret, implementing new optimizations and proving its correctness is very challenging. For our purpose, it would be almost impossible to develop an optimization step for array accesses, as it would be very difficult to specifically target the bytecode sections corresponding to optimizable array accesses, and every change on the bytecode to optimize the accesses would have side-effects on the stack of the EVM.

2.2.1.2. Yul optimizer

The Yul optimizer [14] was introduced in version 0.4.20 of the compiler [7]. It is an optimization module that operates on Yul code (the IR) and serves the role of the machine-independent code optimizer transforming the IR code during the code generation process in a way it leads to a more efficient generated bytecode.

This module is much more powerful than the opcode optimizer. Since it operates at a higher level of abstraction, it can perform more sophisticated optimizations taking advantage of the semantics of the contract code. Furthermore, because there is no possibility of performing arbitrary jumps in Yul, the optimizer can compute the side effects of each function call. This allows the optimizer to perform sophisticated optimizations, such as code reordering or even function call removal. Finally, because it operates at a higher level of abstraction, it is easier for developers to understand, maintain and extend the optimizer. Now, developers do not have to figure out how to target optimizable patterns on the EVM assembly code nor deal with the side effects of the optimizations on the stack. This module generates an optimized IR code that the compiler will transform into ASM code. Consequently, if the optimized

[5]Read more about the peephole optimization in the *Dragon Book* [1, p. 549]

IR code is equivalent to the original IR code, the generated ASM code is guaranteed to be equivalent to the ASM code that would be generated from the original code.

The Yul optimizer performs a predefined sequence of optimization steps to the AST of the generated IR, transforming it to optimize the code or to allow further optimizations. This set of steps can be personalized by the developer using the *–yul-optimizations* command-line parameter. These are some of the most relevant optimization steps detailed in the Yul optimizer documentation [14]:

- `LoadResolver`: replaces loads from memory or storage for its value if known.
- `DeadCodeEliminator`: removes unreachable code.
- `EqualStoreEliminator`: removes store instructions to memory or storage if there is an identical call without any changes on the parameter values in between.
- `LoopInvariantCodeMotion`: moves variable declarations outside the loop if such variables remain constant during the loop and have only read side effects or no side effects.
- `ForLoopConditionIntoBody`: moves the condition expression of the loop into the body.
- `ExpressionInliner` and `FullInliner`: replace function calls with a copy of the corresponding function body.

When the optimization option on the compiler is activated, the Yul optimization can take place in two different phases of the compilation process.

- **Default ASM generation.** In the case the *–via-ir* flag is not set, the EVM assembly code is directly generated from the original Solidity code, as seen in section 2.1.3.2. However, as explained in the previously mentioned section, this code generation process uses, in some cases, predefined function templates coded in Yul. When the optimizations are activated, the generated Yul functions from those templates and the Yul code inside the inline assembly blocks will be optimized by the Yul module before being transformed into EVM assembly code. In this scenario, the Yul optimization has a limited effect on the final ASM code and, consequently, the bytecode. This is because it only optimizes small independent sections of the contract but does not optimize the contract as a whole.
- **ASM generation via IR.** If the *–via-ir* flag is set, the IR code is used to generate the EVM assembly code. Therefore, when the optimizations are enabled, the Yul optimizer module will optimize the IR code, and this optimized IR code will be used to generate the EVM assembly code. Consequently, in

this scenario, the Yul optimizer has a much more significant impact on the code because all the code will be optimized as a whole, being able to capture relations between all the elements of the code.

2.2.2. Related work on code optimizations

In addition to the compiler optimization modules, developers can find a great variety of external optimization tools focused on improving efficiency in Ethereum smart contracts.

A relevant example of external optimizations for EVM code is *GASPER*, an optimization tool proposed in the article *Under-Optimized Smart Contracts Devour Your Money* [6]. In this article, a group of researchers from different Chinese universities described seven costly code patterns not being optimized by the Solidity compiler. Those patterns were separated into two categories. The *useless code-related patterns* category includes situations where a nested conditional evaluates to true or to false under all circumstances due to its relation with the condition they are enclosed into. Besides, the *loop-related patterns* collect simple loop patterns where expensive operations can be moved outside the loop or duplicated operations can be combined or removed. Finally, they implemented *GASPER*, a tool that automatically identifies the code-related patterns and expensive operations on loops, giving the developer valuable information to optimize the code.

Another relevant work on the same area was presented in *Characterizing Efficiency Optimizations in Solidity Smart Contracts* [5], where a group of researchers from the Vienna University of Technology analyze the applicability of 25 optimization strategies for Solidity smart contracts. Those strategies are divided into:

- ***Time-for-Space Rules*** where memory and storage usage is reduced by not storing any value that can be computed when needed, which on the other hand, increases the execution time.

- ***Space-for-Time Rules*** where execution time is reduced by storing precomputed or frequently used data, which on the other hand, increases the use of memory and storage.

- ***Loop Rules*** that describe strategies to move code out of the loop, to reduce the number of conditional expressions inside the loop body, and to fusion loops.

- ***Logic Rules*** related to logic evaluations. These rules exploit identity properties, reorder evaluations, precompute conditions, and replace boolean variables with condition expressions.

- ***Procedure Rules*** that reduce the number of functions by performing inlining,

transforming iterative functions into recursive functions or

- ***Expression Rules*** that exploit identities remove common subexpressions and combine expressions.

They concluded that while not all of the strategies discussed could be applied to programs targeting EVM or providing gas cost reduction, most of them, 21 out of 25, have direct applicability to smart contracts and have the potential to reduce gas consumption.

Among these examples of research on smart contracts optimization, it is mandatory to mention the research carried out by the Costa Group[6], a research group of the Complutense University of Madrid, in which this work has been developed. This group, dedicated to the research of optimization, verification, and understanding of programs, has carried out relevant publications related to smart contracts optimization in recent years.

An example of their research work is *GASOL* (Gas AnalysiS and Optimization tooL) [2], a gas analyzer and optimizer for smart contracts. GASOL is a tool able to analyze Solidity functions according to different cost models that the programmer can select. It infers the gas cost associated with the targeted program as well as the number of EVM instructions that will require. Moreover, this tool detects optimizable patterns related to storage usage and optionally generates an optimized version of the Solidity code. Optimizations consist of substituting multiple accesses to the same storage value, which are expensive, by accesses to a copy of the value stored in memory, which is considerably cheaper. However, this transformation is only viable when the cost of creating the variable copy and updating the original variable with the final value is paid off by the saved gas on the memory accesses. Thus, it uses the cost analysis performed over the code to detect code sections where this transformation reduces gas costs.

The analysis and optimization capabilities offered by GASOL make it an extremely powerful tool for developers seeking to create efficient Solidity smart contracts.

The article *Inferring Needless Write Memory Accesses on Ethereum Bytecode* [3] is another example of external optimization developed by this group. This article describes a static analyzer that detects unnecessary memory write instructions on the EVM bytecode. The described analyzer identifies memory slot allocation, reads and writes, and detects memory write instructions to access a memory slot that is not being read afterward. This post-compilation optimization has proved to be useful in detecting optimization opportunities on real smart contracts, according to the results provided in the mentioned article.

[6]See more about Costa Group at their website: `https://costa.fdi.ucm.es/web/`.

It is worth noting that the mentioned proposals from the Costa Group focus on reducing the execution cost by optimizing the usage of the memory layout (storage or memory) since the cost of loading or storing values from storage or memory often causes a significant portion of the total gas expense. In line with this shared motivation, the following chapters will introduce two new optimization proposals to reduce gas consumption on array accesses.

Chapter 3

Optional Checking

As explained in the first chapter, index access on storage arrays is one of the most expensive accesses to a value in a contract memory space. This high gas cost is the result of the two loads performed on each index access: one to retrieve the array length and perform the bounds checks and another to retrieve the value at the given index of the array. Therefore, a possible way of reducing the gas cost of arrays accesses is by disabling the out-of-bounds checks. By doing that, we save the gas derived from the comparison and from the load of the array length. With the current version of the compiler (v. 0.8.19), it is only possible to bypass those checks by coding the array accesses using a block of assembly code (Yul), as shown in Figure 3.1. However, Yul programming requires very skilled developers and results in very complex code.

```
1  function sumElements(uint256[] storage array) internal view
       returns(uint256 sum) {
2      assembly {
3          mstore(0x60, array.slot)
4          for { let i := 0 }
5          lt(i, sload(array.slot))
6          { i := add(i, 1) } {
7              sum := add(sload(add(keccak256(0x60, 0x20), i)), sum)
8          }
9      }
10 }
```

Figure 3.1: Example of adding the elements of an array using Yul block

Our idea is to provide a more straightforward solution to bypass out-of-bounds checks using Solidity code. A new 'tool' in the Solidity language that allows programmers to reduce the gas consumption of the code by disabling array bounds

checking on code areas with array accesses that the programmers consider safe (e.g., sequential array accesses).

Based on already existing solutions, such as the `unchecked` block for saving gas on arithmetic operations, and ideas that have been discussed by different compiler developers and Solidity programmers [1], we have implemented a new type of block where out-of-bounds checks are disabled on array index accesses performed inside.

3.1. Unchecked Math

To implement this solution, we inspired ourselves in the `unchecked` block offered by Solidity.

Prior to Solidity 0.8.0, integer overflows and underflows were important sources of bugs and vulnerabilities in Ethereum smart contracts. Since the EVM does not perform underflow or overflow checks on the results of arithmetic operations, many Ethereum smart contracts have experienced unexpected behaviors that have caused critical failures or have been exploited by attackers. To solve this issue, programmers needed to wrap those operations on checks to revert the execution in case of underflow or overflow. Another option was to use libraries such as the `SafeMath` library [21], which introduced additional checks on the operations. Nonetheless, it required adapting the source code for each arithmetic operation to be protected against overflows and underflows.

After Solidity 0.8.0 [9], the compiler automatically includes such checks on the generated code. These checks increase the security of the contracts and free the developers from coding them. However, it also increases the cost of all arithmetic operations as the code now will always execute the checks, even in some situations when underflow and overflow cannot occur. Therefore, we now have a safer default program behavior but also a greater gas cost that may be unnecessary in some scenarios. This is the same problem we are facing with out-of-bounds checking on array index accesses.

In order to solve that situation, the Solidity language provides developers with the `unchecked` block. For all the statements inside that block, no underflow or underflow check is made, allowing developers to save gas on operations they consider safe. Figure 3.2 shows an example of the use of an `unchecked` block inside a smart contract.

[1] Original discussion about the possibility of implementing a similar solution to the `unchecked` block for array accesses can be found at `https://github.com/ethereum/solidity/issues/9117`

```
1  // SPDX-License-Identifier: BSD-4-Clause
2  pragma solidity >=0.8.4;
3
4  contract C {
5     function f(uint a, uint b) pure public returns (uint) {
6        // This subtraction will wrap on underflow.
7        unchecked { return a - b; }
8     }
9
10    function g(uint a, uint b) pure public returns (uint) {
11       // This subtraction will revert on underflow.
12       return a - b;
13    }
14 }
```

Figure 3.2: Example of usage of `unchecked` block

It is also important to note that the use of `unchecked` blocks presents some constraints. Knowing them is useful to design our new block as by establishing similar rules on its use, the new type of block will be familiar to developers. The constraints when using the `unchecked` block are:

- `unchecked` blocks can only be used inside regular blocks.
- To avoid ambiguity, the use of _; is not allowed inside an `unchecked` block.
- `unchecked` blocks cannot be nested.
- Functions called within an `unchecked` block do not inherit the property.
- `unchecked` block does not disable division by zero or modulo by zero checks.

The `unchecked` block is an example of how Solidity provides developers with a tool to reduce the gas cost on the operations they consider safe. With this block, programmers can optimize their contracts by disabling certain checks without using complex low-level code (Yul).

3.2. Unchecked Array

Based on the `unchecked` block for arithmetic operations, we have created a new block in the Solidity language where the out-of-bounds checks are disabled on any array index access inside the block. With this block, developers can reduce the gas

consumption in the sections of the code they consider safe from out-of-bounds array accesses. We call it the `uncheckedArray` block.

```
1  // SPDX-License-Identifier: BSD-4-Clause
2  pragma solidity >=0.8.4;
3
4  contract C {
5     uint256[] arr;
6
7     function f(uint idx) pure public returns (uint) {
8        // This access will not check out-of-bounds.
9        uncheckedArray { return arr[idx]; }
10    }
11
12    function g(uint idx) pure public returns (uint) {
13       // This access will revert on out-of-bounds.
14       return arr[idx];
15    }
16 }
```

Figure 3.3: Example of usage of `uncheckedArray` block

This new block, shown in Figure 3.3, makes it easier for developers to improve the efficiency of their code without affecting its simplicity and maintainability, two of the drawbacks of using inline assembly code to solve this problem.

3.2.1. Constraints

The use of the `uncheckedArray` block presents some constraints to avoid ambiguity and to provide a similar interface to the `unchecked` block for arithmetic operations.

- `UncheckedArray` blocks can only be used inside regular blocks.
- The use of _; is not allowed inside an `uncheckedArray` block.
- `uncheckedArray` blocks cannot be nested.
- Functions called within an `uncheckedArray` block do not inherit the property.
- `uncheckedArray` block does not affect byte arrays or Strings.

3.2.2. Implementation

In order to implement this new kind of block in the Solidity language, we have modified different phases of the compilation process[2]. Since a new variant of the original block structure was created, we needed to modify how it is parsed, analyzed, and transformed into the final code executed by the EVM.

3.2.2.1. AST Representation

To implement the `uncheckedArray` block, we have extended the AST node structure representing the block. A `Boolean` variable has been added to indicate whether the block is an `uncheckedArray` block as well as its corresponding getter, and additionally, all the methods to import and export the AST node in JSON format have been extended to include the new information.

3.2.2.2. Parsing

To implement the `uncheckedArray` block, the compiler parser has to be extended in order to recognize this new language construct.

First, we registered the block identifier token, using 'uncheckedArray' as an identifier since it is descriptive and follows the same convention as the unchecked arithmetic block ('unchecked'). Then, we modified the grammar of Solidity by adding this new kind of block. Figure 3.4 shows the final grammar representing block parsing in Solidity language.

```
1 /**
2  * A curly-braced block of statements. Opens its own scope.
3  */
4 block:
5   LBrace ( statement | uncheckedBlock | uncheckedArrayBlock )*
        RBrace;
6
7 uncheckedBlock: Unchecked block;
8
9 uncheckedArrayBlock: UncheckedArray block;
```

Figure 3.4: Grammar to parse solidity blocks

[2]All changes performed to the original compiler in order to implement this version of the `uncheckedArray` block the can be found at `https://github.com/ethereum/solidity/compare/develop...javierSande:solidity:uncheckedArray`.

Finally, we adapted the block parsing function of the compiler to the new grammar. To do so, we added a new step, shown in Figure 3.5, to check if the `uncheckedArray` token precedes the brackets that open the block being processed.

```
1 bool const uncheckedArrayBlock = m_scanner->currentToken() == Token::
    UncheckedArray;
```

Figure 3.5: Modification on the parse of the `uncheckedArray` block

It is important to note that we have added a new parser error in this phase, which is issued when an `uncheckedArray` block is found outside a regular block. Figure 3.6 shows the introduced parsing error.

```
1 if (!_allowUncheckedArrayBlock)
2         parserError(5297_error, "\" uncheckedArray\" blocks can only
              be used inside regular blocks.");
3       advance();
```

Figure 3.6: New parse error

3.2.2.3. Syntax checking

In order to ensure the correct syntax of `uncheckedArray` blocks, the compiler must check that they do not appear nested in the code. To do so, we have extended the `SyntaxChecker` with a variable to track when it is inside an `uncheckedArray` block. This variable is updated when an `uncheckedArray` block is accessed and exited during the syntax checking (performed using the visitor pattern) and used when an `uncheckedArray` block is accessed to check if the accessed block is inside another `uncheckedArray` block. Additionally, we have added a check in the block visit function to guarantee that the visited `uncheckedArray` block is not inside another one.

3.2.2.4. Type checking

The `uncheckedArray` block does not impact how types must be checked within the block. Therefore, no modification is required.

3.2.2.5. Code generation

Finally, the compiler has been modified to generate the correct code for index accesses performed inside an `uncheckedArray` block. Bounds checks on the array index array accesses are generated at an IR or ASM level, depending on whether the IR code is used to generate the ASM code. Consequently, we have only modified how the array index accesses are generated in the IR and ASM code. As seen in Chapter 2, the Solidity compiler has two modules in charge of generating these representations: the IR generator and the ASM generator.

IR Code generation. The IR code generation for array index accesses uses predefined Yul util functions. Therefore, new util functions have been defined to generate the array accesses to each type of memory location, as the one being shown in Figure 3.7.

```
1  function <functionName>(array, index) -> slot, offset {
2     <?multipleItemsPerSlot>
3        <?isBytesArray>
4           switch lt(arrayLength, 0x20)
5              case 0 {
6                 slot, offset := <indexAccessNoChecks>(array, index)
7              }
8              default {
9                 offset := sub(31, mod(index, 0x20))
10                slot := array
11             }
12       <!isBytesArray>
13          let dataArea := <dataAreaFunc>(array)
14          slot := add(dataArea, div(index, <itemsPerSlot>))
15          offset := mul(mod(index, <itemsPerSlot>), <storageBytes>)
16       </isBytesArray>
17       <!multipleItemsPerSlot>
18          let dataArea := <dataAreaFunc>(array)
19          slot := add(dataArea, mul(index, <storageSize>))
20          offset := 0
21       </multipleItemsPerSlot>
22 }
```

Figure 3.7: Predefined Yul util function for a storage index access inside an `uncheckedArray` block

Since those new functions are inserted in the resulting code by the `IRCodeGenerator`, using the information from the AST node and the `IRGenerationContext`, we have

extended the context so the generator can determine whether array access must include bounds checks. The solution is to include a new variable with two possible enum values: `Checked` or `Unchecked`. This variable stores the value `Unchecked` while the generator traverses the nodes inside an `uncheckedArray` block, and the value `Checked` otherwise. Then, the generator uses that information from its context to decide which predefined array access function to insert, as shown in Figure 3.8.

```
1 m_context.uncheckedArrays() ?
2   m_utils.storageUncheckedArrayIndexAccessFunction(arrayType) :
3   m_utils.storageArrayIndexAccessFunction(arrayType))
```

Figure 3.8: Call to generate a storage array access in IR code

EVM assembly code generation. The EVM assembly code generation is highly complex because of how local variables are treated on the limited stack of the EVM (Section 1.4 Chapter 1). Fortunately, the generated code of most of the basic operations is predefined, as it is on the IR code generation explained in the previous paragraph. This is the case of array operations such as `push`, `pop`, or length accesses, whose generated assembly code is defined in the `ArrayUtils` class.

The modifications needed here are minimal as the function in charge of generating the array index access code already contemplated the possibility of not doing bounds checks over the array length. The official compiler version uses this option when the access is part of other larger operations, such as a push or an assignment of an array from memory to storage (copy of the array), where the soundness of the array accesses is guaranteed by construction. Therefore, we have extended the corresponding compiler context, as we did with the IR generation context, so the `ExpressionCompiler` can determine whether to add the bounds checks to the EVM assembly code. Figure 3.9 shows the modification made on the generator code in order to enable or disable the checks on array accesses based on the information of the compiler context.

```
1 checkAccess = !m_context.uncheckedArrays();
2 ArrayUtils(m_context).accessIndex(arrayType, checkAccess);
```

Figure 3.9: Call to generate the EVM assembly code of a storage array access

3.3. Targeted Unchecked Array

To increase the power of this new gas-saving mechanism, we want it to support targeting specific arrays inside the block. With this slight improvement, developers

can include at the opening of the `uncheckedArray` block a list of the array bases that should not be checked on index access. The list is optional, and if it is not provided, none of the arrays accessed inside the block will perform bounds checks. In Figure 3.10, we can see how this new feature of the `uncheckedArray` block is used.

```
1  // SPDX-License-Identifier: BSD-4-Clause
2  pragma solidity >=0.8.4;
3
4  contract C {
5     uint256[] arrA;
6     uint256[] arrB;
7
8     function f(uint idx) pure public returns (uint) {
9        // arrA access will not check out-of-bounds.
10       uncheckedArray(arrA) {
11          return arrA[idx] + arrB[idx];
12       }
13    }
14 }
```

Figure 3.10: Example of usage of targeted `uncheckedArray` block

3.3.1. Constraints

In addition to the `uncheckedArray` block constraints (Section 3.2.1), there is a significant limitation when targeting array accesses. Since expressions cannot be evaluated at compilation time, array bases have to be literally compared. Therefore, array base expressions need to be transformed into strings to be compared.

To avoid possible misunderstandings, we have extended the compiler with a new warning, shown in Figure 3.11. This warning rises when an expression different from an identifier is listed as a targeted array base.

```
1  Warning: The array accesses performed over a base listed here will
      not perform index out-of-bounds checks. Comparison between array
      bases is literal. Only in those accesses with the same literal
      base the uncehckedArray will take effect.
2   --> c.sol:11:22:
3    |
4  11 | uncheckedArray(matrix[i]) {
5    | ^^^^^^^^^
```

Figure 3.11: Warning about literal comparisons of the targeted array bases

3.3.2. Implementation

The implementation of the optional target list on the `uncheckedArray` block required to extend most of the compilation phases originally modified[3]. We had to modify the AST block node and the parsing in order to retrieve and store the targets list, the different analyzers to check that each element in the list is valid and an array base, and finally, the code generator to only disable the bounds checks on the accesses to targeted arrays when specified.

3.3.2.1. AST Representation

In addition to the original changes to the block node, we needed to extend the block node to store the list of the targeted array bases when provided. In order to do that, a vector of expressions has been added to the block attributes. Those provided expressions became then children of the block node in the AST. Therefore, the `accept` function (visitor pattern) of the block has also been modified to allow the visitors to access the array base list. This extension is fundamental to ensure that all the compiler checks are performed over the elements of the targets list.

Finally, a method `nodeToString` has been defined for all the expression type nodes, so they can be literally compared between them (see 3.3.1). Figure 3.12 shows an example of the `nodeToString()` methods to represent member accesses.

```
1 ASTString const MemberAccess::nodeToString() const {
2   return expression().nodeToString() + TokenTraits::toString(Token::
      Period) + memberName();
3 }
```

Figure 3.12: Member access `nodeToString` method

3.3.2.2. Parsing

The only difference when parsing this new version of the block is that the `uncheckedArray` block can now receive a list of parameters between the identifier and the opening braces of the block. Therefore, we have extended the grammar with this new feature, as shown in Figure 3.13, and modified the parser phase. Now, when the parser finds the `uncheckedArray` keyword, it looks for an opening parenthesis to parse the target list. If it founds an opening brace instead, it will treat

[3]All changes performed to the original compiler in order to implement this version of the `uncheckedArray` block the can be found at `https://github.com/ethereum/solidity/compare/develop...javierSande:solidity:targetedUncheckedArray`.

the block as an `uncheckedArray` block where the bounds checks are disabled on all array index accesses.

```
1  /**
2   * A curly-braced block of statements. Opens its own scope.
3   */
4  block:
5    LBrace ( statement | uncheckedBlock | uncheckedArrayBlock )*
         RBrace;
6
7  uncheckedBlock: Unchecked block;
8
9  uncheckedArrayBlock:
10   UncheckedArray block | UncheckedArray LParen (expression? ( Comma
         expression?)* ) RParen block;
```

Figure 3.13: Grammar to parse blocks

3.3.2.3. Syntax checking

This new feature of the `uncheckedArray` block has no other effect on the syntax checking than extending the checking over the expressions on the targets list. However, this was already solved when we adapted the `accept` function of the block node.

3.3.2.4. Type checking

When a list of targets is specified in an `uncheckedArray` block, the compiler must perform type-checking on the expressions on that list. As with syntax checking, this was solved when we adapted the `accept` function of the block node. However, we also needed to implement a new type check over the parameter list to guarantee that all the parameters conform to valid array bases. In order to do that, we have modified the type checker so it traverses the list of bases, checking that its type belongs to the `Array` category and is not a string or byte array. Finally, we have modified the checker so it raises the warning described in Section 3.3.1 (Figure 3.11) whenever it finds on the targets list an array base expression that is not an identifier.

3.3.2.5. Code generation

The only change in the code generation is how the IR and EVM assembly code generators decide whether array access must include the bounds checks. Now, we

have three possibilities:

- The `uncheckedArray` block has no targets, so the bounds of all the arrays are unchecked.
- The `uncheckedArray` block has a list of targets, so only the bounds of targeted arrays are unchecked.
- We are outside any `uncheckedArray` block, so the bounds of all the array accesses are checked.

Consequently, we have modified the generation contexts to be able to store the required information to identify these three scenarios. We keep the previously added variable, indicating if we are inside an `uncheckedArray` block that affects all the arrays (`Unchecked`) or not (`Checked`), and a new vector has been created in order to store the targeted bases, if any.

Now, the code generator will query its context whether the array access must bypass bounds checks, as shown in Figure 3.14. With our modifications, the context will now answer affirmatively if the code is into an `uncheckedArray` block without targets (where all accesses are unchecked) or if the array base is literally equal to one of the targets stored in the context.

```
1 checkAccess = !m_context.isArrayUnchecked(baseExpression);
2 ArrayUtils(m_context).accessIndex(arrayType, checkAccess);
```

Figure 3.14: Call to generate a storage array access in EVM assembly code

3.4. Results and experiments

As explained at the beginning of this chapter, skipping the out-of-bounds checking on array index accesses reduces the consumed gas. By doing this, we save the cost of the length retrieval, the comparison, and other instructions that take part in the bounds check.

Once we implemented the `uncheckedArray` block to bypass such bounds checks, we wanted to quantify the gas savings on array accesses. In order to do that, for each kind of access (in storage, memory, and calldata), we have performed a study on the bounds check bytecode, its instructions, and cost, quantifying the theoretical gas saving of removing the check. Finally, we have executed different smart contracts to measure the real impact of the `uncheckedArray` block.

3.4.1. Storage gas saving

Since accessing storage has the highest gas cost among all the possible memory accesses, we expect the `uncheckedArray` block to have the most important gas reduction when applied to storage array accesses. In Figure 3.15, we can observe the main part of the bounds check in the EVM assembly code:

```
1 DUP2
2 SLOAD // Load length
3 DUP2
4 LT  // Compare
5 PUSH2
6 0x75
7 JUMPI // Jump to panic function
```

Figure 3.15: Example of EVM assembly code for bounds checks on storage arrays

From the observed code and according to the current gas costs published by the Ethereum foundation at EIP-2929 [8], by bypassing that check, we will save gas by not executing the following instructions:

- Two `DUP2` instructions, with a cost of 3 gas units each.
- A load from storage (`SLOAD`), with a cost of 2100 units on the first access to the address and of 100 in later accesses.
- A comparison (`LT`), with a cost of 3 gas units.
- A `PUSH2` instruction, with a cost of 3 gas units.
- A `JUMPI` instruction, with a cost of 10 gas units.

This results in an estimated gas save of 122 units. It is a theoretical result, and this gas-saving can be slightly different in practice since other instructions may be avoided or introduced to keep track of variables in the stack of the EVM, and the structure of EVM bytecode may be different, resulting in a different division of the code into blocks and, consequently, a different number of jump operations. Additionally, if the array length has not been previously accessed, we will save 2100 gas units on the first array access.

Experimental results. To measure the gas savings using the `uncheckedArray` block, we have developed a simple benchmark. It comprises a smart contract with an array in storage and a function that iterates that array and computes the sum

of its elements. This function has two versions: one that wraps the loop in an `uncheckedArray` block (Figure 3.16) and another without the `uncheckedArray` block.

```
1 function accessStorage() public returns (uint) {
2    uint sum = 0;
3    uncheckedArray(array) {
4       for(uint256 i = 0; i < array.length; i++)
5          sum += array[i];
6    }
7    return sum;
8 }
```

Figure 3.16: Tested function

The benchmark has been executed several times with different array lengths, showing the following results:

Iterations	Original Gas	Unchecked Gas	Diff	Diff per Iteration
1	26380	26279	101	101.00
10	51220	50012	1208	120.80
100	299620	287342	12278	122.78
1000	2783620	2660642	122978	122.98

Table 3.1: Gas savings on storage accesses

In Table 3.1, we can see the results of executing the code shown in Figure 3.16 with and without the `uncheckedArray` block over arrays of 1, 10, 100, and 1000 elements. Results show that as we increase the array size and, consequently, the iterations to access the array, the saved gas per iteration tends to be 123 gas units. This saving is one unit higher than expected, and most probably, it is because bypassing the bounds check means we are avoiding an extra `JUMP` instruction (cost of 1 unit of gas) to exit from the block containing such a check.

However, when executed on an array with only one array, the improvement is smaller than expected. If we only perform one iteration, we save 21 units less than expected (22 if we take 123 units as the new reference). This difference is because, as explained, removing the bounds checks, and therefore, some blocks of the code, may generate a redistribution of the bytecode. This redistribution can have side effects on the execution cost of the rest of the contract, increasing or decreasing the gas consumed. Looking at these results, we can interpret that in this specific case, we reduce the gas consumption by 123 units per iteration, but the rest of the function increases its consumption by 21 units. Therefore, if the function only performs one iteration, the gas saving will be 101 units, but as we increase the number of

iterations, this extra cost is distributed between all the iterations, getting close to the 123 units of gas saved per iteration.

Again, this is valid for this particular case. On other functions, the effect on the cost execution unrelated to the array index accesses can be different, even causing a reduction. Nonetheless, this side effect on the contract gas cost is minimal compared to the potential savings of the `uncheckedArray` block, especially when performing multiple accesses.

3.4.2. Memory gas saving

In the case of arrays in memory, we expect the `uncheckedArray` block to have a lower impact on the gas cost. In Figure 3.17, we can observe the main part of the bounds check in the EVM assembly code:

```
1 DUP2
2 MLOAD // Load length
3 DUP2
4 LT  // Compare
5 PUSH2
6 0x75
7 JUMPI // Jump to panic function
```

Figure 3.17: Example of EVM assembly code for bounds checks on memory arrays

From the observed code, according to EIP-2929 [8], we can conclude that by bypassing that check, we will save gas by not executing the following instructions:

- Two `DUP2` instructions, with a cost of 3 gas units each.
- A load from memory (`MLOAD`), with a cost of 3 units.
- A comparison (`LT`), with a cost of 3 gas units.
- A `PUSH2` instruction, with a cost of 3 gas units.
- A `JUMPI` instruction, with a cost of 10 gas units.

This results in a theoretical gas save of 25 units. However, as in the case of storage accesses, this number may vary slightly depending on the compiled contract.

Experimental results. To measure the gas savings using the `uncheckedArray` block, we have developed a benchmark similar to the one used to test the storage array accesses. The only difference is that the function computes the sum of the elements of an array stored in memory. As in the previous benchmark, this function has two versions: wrapping the for loop in an `uncheckedArray` block (Figure 3.18) and another without using an `uncheckedArray` block.

```
1 function accessMemory(uint[] memory array) public returns (uint) {
2    uint sum = 0;
3    uncheckedArray(array) {
4       for(uint256 i = 0; i < array.length; i++)
5          sum += array[i];
6    }
7    return sum;
8 }
```

Figure 3.18: Tested function

The benchmark has been executed several times with different array lengths, showing the following results:

Iterations	Original Gas	Unchecked Gas	Diff	Diff per Iteration
1	23237	23201	36	36.00
10	30365	30095	270	27.00
100	101666	99056	2610	26.10
1000	816433	790423	26010	26.01

Table 3.2: Gas savings on memory accesses

On Table 3.2, we can see the results of executing the code shown in Figure 3.18 with and without the `uncheckedArray` block over arrays of 1, 10, 100, and 1000 elements. As we observed and explained in the previous experiment, the resulting gas saving tends to be one unit higher than expected. Contrary to what we observed in the experiment over storage arrays, the side effect on the rest of the function execution cost is a saving of 10 gas units. Therefore, this extra saving is distributed as we increase the number of iterations, resulting in a downward trend to 26 units of gas saving per iteration.

3.4.3. Calldata gas saving

Regarding arrays in calldata, we expect the `uncheckedArray` block to have a lower impact on the gas cost than with arrays in storage or memory. In Figure 3.19, we can observe the main part of the bounds check in the EVM assembly code:

```
DUP2
DUP2
LT  // Compare
PUSH2
0x75
JUMPI // Jump to panic function
```

Figure 3.19: Example of EVM assembly code for bounds checks on calldata arrays

Therefore, by bypassing those checks, and according to the current gas costs published at EIP-2929 [8], we save gas by not executing:

- Two `DUP2` instructions, with a cost of 3 gas units each.
- A comparison (`LT`), with a cost of 3 gas units.
- A `PUSH2` instruction, with a cost of 3 gas units.
- A `JUMPI` instruction, with a cost of 10 gas units.

This results in a theoretical gas save of 22 units. However, as in the previous cases, this number may vary slightly depending on the compiled contract.

Experimental results. To measure the gas savings using the `uncheckedArray` block, we have developed an identical benchmark to the one used to test the memory array accesses. The only difference is that the function receives an array from calldata.

```
function accessMemory(uint[] calldata array) public returns (uint) {
   uint sum = 0;
   uncheckedArray(array) {
      for(uint256 i = 0; i < array.length; i++)
         sum += array[i];
   }
   return sum;
}
```

Figure 3.20: Tested function

The benchmark has been executed several times with different array lengths, showing the following results:

Iterations	Original Gas	Unchecked Gas	Diff	Diff per Iteration
1	22679	22634	45	45.00
10	27791	27539	252	25.20
100	78911	76589	2322	23.22
1000	590123	567101	23022	23.02

Table 3.3: Gas savings on calldata accesses

On Table 3.3, we can see the results of executing the code shown in Figure 3.20 with and without the `uncheckedArray` block over arrays of 1, 10, 100, and 1000 elements. In this case, the gas savings is also one unit higher than expected, probably for the same reasons as in the previous experiments. We can observe a trend of 23 gas units saving per iteration. As seen with the experiment on the memory arrays, the side effects on the execution cost produce an extra saving of 12 gas units, which is distributed as we increase the number of iterations.

3.5. Conclusions

This chapter aims to introduce a new tool in the Solidity language to reduce gas consumption on array accesses. In order to do that, we have proposed the `uncheckedArray` block, which reduces gas cost by bypassing the out-of-bounds checks on array accesses. First, we developed a basic version where checks were disabled in every array access performed inside the block. Then, we extended this version, allowing developers to target specific arrays inside the block and consequently giving programmers an even more fine-grained control of the gas consumption of array index accesses.

As proved by experimental results, this new block produces a gas cost reduction as effective as theoretically expected. It is able to save gas on array accesses without a negative impact on the code performance, resulting in a significant reduction in the gas consumed, especially when dealing with storage arrays.

The main advantage of this block, when compared to the current solutions available to optimize accesses, is its simplicity and easy use, requiring minimal changes to the source code that does not affect code readability or maintainability. In addition, another advantage of this new block is that it resembles an existing solution in the Solidity language, especially the basic version, whose operation is identical to that of the `unchecked` block for arithmetic operations. This resemblance makes it easier for this solution to be adopted in the near future, as it is based on the same principles that the developer community has accepted and currently employs with the `unchecked` block.

Finally, it is necessary to note that this optimization tool has two main drawbacks. Firstly, it generates security breaches if not used correctly by developers. With the `uncheckedArray` block, we reduce the gas consumption at the cost of security, and, although used correctly, it is harmless, it can generate important vulnerabilities when used by inexperienced programmers. Lastly, we need to modify the code in order to apply the optimization provided by this new block. Having to add this block to existing source code is a considerable inconvenience for libraries developed in the past that programmers frequently use to develop new smart contracts.

These two weak points of the `uncheckedArray` block motivate the new optimization mechanism proposed in the following chapter, which will reduce the gas cost of array accesses without risking security nor modifying the source code of smart contracts or libraries.

Chapter 4

Compiler Optimizations

Although the `uncheckedArray` block allows us to reduce the cost of accessing an array, it has two main drawbacks. Firstly, developers have to modify the smart contract code to optimize the desired array accesses. And in second place, bypassing the bounds check can result in failures on the contract if not used correctly. Therefore, in this project phase, we want to create a new optimization of array accesses without the mentioned cons. We want to introduce a new optimization step at compile time that reduces the gas cost of array accesses without affecting the soundness and safety of the resulting code.

To find possible optimizations, we focus on how arrays are usually accessed. In most smart contracts, as in other programs written in different programming languages, arrays are generally accessed inside loops. Generating efficient loops is crucial to maintain a low execution cost. Since the instructions inside the loop will presumably be executed several times, it is essential to keep as many instructions as possible outside the loop.

```
1 function search(uint x) view public returns (uint, bool) {
2   for (uint i = 0; i < arr.length; i++) {
3      if (arr[i] == x)
4         return (i,true);
5   }
6   return (0, false);
7 }
```

Figure 4.1: Example of sequential search in Solidity

In Figure 4.1, we can observe a basic example of for loop used to perform a sequential search on an array. This function looks simple and optimal. However, there is an equivalent way of writing this loop that will save a significant amount

of gas per iteration. Since accessing values from storage has a high cost, when the array size remains constant inside the loop, it is highly convenient to store the length of the array in a local variable (stack) outside the loop. This variable can then be used inside the loop condition, saving a storage load per iteration. As shown in Figure 4.2, it is a simple change on the code that can make us save around 100 gas units per iteration (cost of load from storage). Nonetheless, developers sometimes do not perform this optimization due to oversight or ignorance.

```
1 function search(uint x) view public returns (uint, bool) {
2    uint len = arr.length;
3    for (uint i = 0; i < len; i++) {
4       if (arr[i] == x)
5          return (i,true);
6    }
7    return (0, false);
8 }
```

Figure 4.2: Example of optimized sequential search in Solidity

There is another possible optimization for this loop, similar to the previous one, but which is only possible to perform in the source code using an inline assembly block. As we know from previous chapters, when accessing the elements in storage or memory arrays, the array length must be loaded to perform the bounds checking. This optimization aims to store such length on a local variable outside the loop and use it to check bounds on each array index access performed inside the loop.

4.1. Current Loop Optimizations

Once the two optimization goals to implement in this phase of the project are set, we must study how the current optimizer modules of the compiler treat array length and index accesses inside loops.

In the case of the opcode optimizer (see Section 2.2.1.1), since it works at a very low level, it can only optimize array accesses in very specific cases. Using the `ConstantOptimizer`, the module can replace the array length loads of static-sized arrays with the computed length at compile time. This has a gas-saving effect on index and length accesses over arrays with a predefined fixed size. However, this module performs no optimization on dynamic-sized arrays.

On its side, the Yul optimization module (see Section 2.2.1.2), in very specific situations, is able to perform the optimizations mentioned in the introduction of this chapter on dynamic-sized arrays. By using the `LoopInvariantCodeMotion` step

together with function inlining, the Yul module is able to identify some situations where the array length load can be performed outside the loop, as we see in the next section.

4.1.1. Loop Invariant Code Motion

The Loop Invariant Code Motion is an optimization step of the Yul optimizer module, introduced in Section 2.2.1.2 of Chapter 2. This step analyzes the body of the loop and moves variable declarations outside the loop if such variables remain constant during the loop and have only read side effects (e.g., a load from storage or memory) or no side effects. This optimization is very powerful because it can prevent the program from computing expressions with constant results on each iteration.

However, since this module works at a relatively low level, it presents two significant limitations that restrict the situations where the optimization can be applied. Foremost, it works only at the top level in the loop body and post block, i.e., variable declarations inside conditional branches will not be considered for moving. And second, it cannot reason about fine-grained storage or memory locations. Consequently, if the code writes to any location in the same memory region (storage or memory) inside the loop body, the compiler is not able to determine whether the location written corresponds to the length of the array being cached or to any other location in the storage or memory, and, consequently, the optimization is not applied.

4.1.2. Real case analysis

In order to completely analyze how this optimization works on both index and length array accesses inside the loop, we let us take the function shown in Figure 4.3 as an example. This function is one of the particular situations where the Yul optimizer is able to optimize the array length access and the array index access by extracting the array length loads from the loop.

In the following explanation of how this loop is optimized, we will only focus on the steps that directly affect the array length and index access. Since the IR code has high complexity and the complete optimization process makes the result very difficult to interpret, the code shown to support the explanation is just a representation of how the real IR code would be modified if we only apply the mentioned optimization steps. Therefore, many optimization steps have been left aside, some expressions have been simplified, and certain variables have been conveniently renamed or deleted.

```
1 function sum() view public returns (uint) {
2    uint s = 0;
3    for (uint i = 0; i < arr.length; i++)
4       s += arr[i];
5    return s;
6 }
```

Figure 4.3: Function to add the elements of an array

The Yul code shown in Figure 4.4 corresponds to the IR generated by the compiler for the function of Figure 4.3. Observe that the loop condition has been moved to the loop body (Lines 10-13). This code reordering is performed by the optimization step `ForLoopConditionIntoBody`, which moves the condition expression of the loop into the body. This optimization step is applied by default when generating the IR code of loops, even if the optimizations are disabled. It is important to note that the generated code uses Yul functions for performing basic operations on the array at Lines 12, 17, and 18. We will focus on the calls highlighted at Lines 12 and 17, which code is shown in Figure 4.5.

Once the compiler has generated the IR code, it will start with the optimization process. The first relevant steps of the loop optimization performed by the compiler are related to function inlining. During this process, the optimizer module will try to replace function calls in the code with the body of the called function. This process is performed in two different steps, the `ExpressionInliner` and the `FullInliner`, which target different kinds of function calls:

- **Expression Inliner.** The expression inliner step inlines functions to replace calls inside functional expressions. This optimization step is applied when the following conditions hold:
  - The expression returns a single value.
  - It is on the left side of a variable assignment.
  - The expression has only movable[1] arguments.
  - The expression has arguments that are small constants or that are referenced less than twice in the function body.

  Therefore, in the case of array access optimizations, the expression inliner will exclusively affect the call to the auxiliary length loading function at Line 12 as it complies with all the conditions. In contrast, the auxiliary function in

[1] According to the compiler documentation, an expression is considered movable "if it is side-effect free and its evaluation only depends on the values of variables and the call-constant state of the environment".

```
1  function fun_sum_34() -> var__7 {
2     var__7 := zero_value_for_split_t_uint256()
3     let var_s_10 := convert_t_rational_0_by_1_to_t_uint256(0x00)
4
5     for {
6        let var_i := convert_t_rational_0_by_1_to_t_uint256(0x00)
7     } 1 {
8        var_i := increment_t_uint256(var_i)
9     } {
10       // Loop condition: i < arr.length
11       let _slot := 0x00
12       let expr_19 := array_length_t_array$_t_uint256_$dyn_storage(
             _slot)
13       if iszero( lt(var_i, expr_19) ) { break }
14
15       // Load arr[i]
16       let _1_slot := 0x00
17       let _8, _9 :=
            storage_array_index_access_t_array$_t_uint256_$dyn_storage(
            _1_slot, var_i)
18       let _10 := read_from_storage_split_dynamic_t_uint256(_8, _9)
19
20       // sum += arr[i]
21       var_s_10 := checked_add_t_uint256(var_s_10, _10)
22    }
23
24    var__7 := var_s_10
25    leave
26 }
```

Figure 4.4: IR code from Figure 4.3

charge of the array access at Line 17 does not comply with the first condition, as it returns two variables.

- **Full Inliner.** The full inliner step performs function inlining if the transformation does not lead to a larger code. Therefore, it inlines functions only if the called function is very small or if it is called only a few times in the entire code.

  Consequently, the array length getter would also be inlined by this optimization step because its body comprises a single instruction. Nevertheless, since the auxiliary function to perform the array index access is considered a large function, it will only be inlined into large functions if used only a few times in the code. Our experiments detected that the optimizer does not inline the

function call with more than three index accesses expressions on the code. Additionally, if several functions contain array accesses, the inlining does not occur. We cannot establish an exact heuristic since this behavior varies depending on the code size, how many accesses are produced, and where they are produced. However, we can establish that this is a major limitation because in most of the cases tried, with experimental and real contracts, this inlining is not produced, and, without this inlining, the following optimization steps over the array access are not possible.

```
1  function storage_array_index_access_t_array$_t_uint256_$dyn_storage(
       array, index) -> slot, offset {
2     //Bounds check
3     let arrayLength := array_length_t_array$_t_uint256_$dyn_storage(
          array)
4     if iszero(lt(index, arrayLength)) {
5        panic_error_0x32()
6     }
7     // Compute the storage slot of the element in the array
8     let dataArea := array_dataslot_t_array$_t_uint256_$dyn_storage(
          array)
9     slot := add(dataArea, mul(index, 1))
10    offset := 0
11 }
12
13 function array_length_t_array$_t_uint256_$dyn_storage(value) ->
       length {
14    length := sload(value)
15 }
```

Figure 4.5: IR auxiliar functions used in Figure 4.4

Since our code contains a single array access, the inlining process will successfully replace the function calls to the array length getter and the array index access auxiliary functions, shown in Figure 4.5. In Figure 4.6, we can observe the code resulting from this optimization process.

Finally, the loop invariant code motion step will try to move outside the loop the declaration of variables that remain constant inside the loop. Since there is no side-effect on loading the array length and the length remains constant (there is no writing to storage), the optimizer will be able to move both array length loads, the one for the loop condition and the one for the loop access. Here is where having the array index access inlined is critical. Because it is inlined, the array length access for the bounds checks (Line 11 in Figure 4.6) can be identified as a constant expression by the loop invariant code motion. If it were not inlined, the length load would be performed inside the array index access auxiliary function that is called inside

the loop with variable arguments (the index being accessed) and, consequently, a non-constant expression.

```
function fun_sum_34() -> var__7 {
   var__7 := zero_value_for_split_t_uint256()
   let var_s_10 := convert_t_rational_0_by_1_to_t_uint256(0x00)

   for {
      let var_i := convert_t_rational_0_by_1_to_t_uint256(0x00)
   } 1 {
      var_i := increment_t_uint256(var_i)
   } {
      // Loop condition: i < arr.length
      let expr_19 := sload(0x00)
      if iszero( lt(var_i, expr_19) ) { break }

      // Load arr[i]
      let _1_slot := 0x00

      //Load array length
      let arrayLength := sload(_1_slot)

      //Bounds check
      if iszero(lt(var_i, arrayLength)) { panic_error_0x32() }

      // Compute the storage slot of the element in the array
      let dataArea := array_dataslot_t_array$_t_uint256_$dyn_storage(
         _1_slot)
      slot := add(dataArea, mul(var_i, 1))
      offset := 0

      // Load value from storage
      let _10 := extract_from_storage_value_dynamict_uint256(sload(
         _1_slot), offset)

      // sum += arr[i]
      var_s_10 := checked_add_t_uint256(var_s_10, _10)
   }

   var__7 := var_s_10
   leave
}
```

Figure 4.6: Optimized IR code after applying `ExpressionInliner` and `FullInliner` steps to the code in Figure 4.4

In this particular case, the optimizer will even detect that the loop condition expression (Line 12 in Figure 4.6) and the condition expression of the bounds check on the array access (Line 21 in Figure 4.6) are the same. Since the first condition leads to an exit of the loop when reached, the second condition always evaluates to true, so the optimizer will remove the bounds check on the array access. Figure 4.7 shows the IR code after optimizing the array length and access loads. In this Figure, we can see that the code reading the length of the array has been moved out of the loop to Line 4. If the conditions were not equal, the bounds check would have remained on the code, but since both load expressions (Line 11 and 18 in Figure 4.6) are equivalent, the optimizer will remove one of them and use the same variable for both conditional expressions.

```
1  function fun_sum_34() -> var__7 {
2     var__7 := zero_value_for_split_t_uint256()
3     let var_s_10 := convert_t_rational_0_by_1_to_t_uint256(0x00)
4     let arrayLength := sload(0x00)
5
6     for {
7        let var_i := convert_t_rational_0_by_1_to_t_uint256(0x00)
8     } 1 {
9        var_i := increment_t_uint256(var_i)
10    } {
11       // Loop condition: i < arr.length
12       if iszero( lt(var_i, arrayLength) ) { break }
13
14       // Load arr[i]
15       // Compute the storage slot of the element in the array
16       mstore(0, ptr)
17       let dataArea := keccak256(0, 0x20)
18
19       slot := add(dataArea, mul(var_i, 1))
20       let _10 := extract_from_storage_value_dynamict_uint256(sload(
            slot), 0)
21
22       // sum += arr[i]
23       var_s_10 := checked_add_t_uint256(var_s_10, _10)
24    }
25
26    var__7 := var_s_10
27    leave
28 }
```

Figure 4.7: Optimized IR code after applying `LoopInvariantCodeMotion` step to the code in Figure 4.6

To summarize, after all these optimization steps are applied, some unnecessary variables are removed, and the generated code will produce two fewer loads from storage per iteration than the original one, resulting in a great improvement in contract efficiency. If the array were stored in memory, the optimization process would be the same, but the amount of gas saved would be much lower because the cost of loading values from memory is only three gas units in most cases. In the case of an array in calldata, the optimization would not have any effect since the length of the array would already be stored in the stack.

However, this is an ideal case. As we have seen during the process, the optimization module has several important limitations:

- Regarding array length accesses inside the loop, the following conditions must hold:
  - There is no write to storage inside the loop.
  - The array length access must be at the loop body top level and not inside any other structure, such as if-then expressions or nested loops.
- Regarding array index accesses, an additional very strong limitation is required:
  - The code can only have a few array index accesses in the entire contract, usually less than four[2].

This last constraint makes it almost impossible to take advantage of this optimization module on array accesses on smart contracts.

In this analysis of the current optimizations on arrays inside loops, we have seen that even though they exist and they are quite powerful, they can be used in very limited situations, especially in the case of array index accesses. Consequently, it would be interesting to create a new array access optimization, based on the same principles, that overcomes most of the current limitations and, therefore, covers a much wider range of situations.

4.2. A new optimization phase

Knowing the limitations of the current optimizer modules when dealing with loops and array accesses, we decided to implement a new optimization phase at a higher level of abstraction. This new optimization phase overcomes most of the

[2]This number may vary slightly depending on the size of the contract and function where the accesses are performed.

limitations faced by the Yul optimizer. We propose to directly generate optimized array access in IR code, using all the information we can extract from the AST of the Solidity code. This extra information gives the optimizer a significant advantage compared to the current modules, allowing us to optimize array accesses, both to their length and by index, in a broader range of situations. Additionally, this new optimization phase opens a new window to further complex optimizations that could not be implemented on the existing modules.

For example, this optimization allows storage modifications that do not affect the array length inside the loop. At the Yul level, the optimizer can only see a `SSTORE` instruction to a dynamically computed storage slot, and it can not know if that slot is the one containing the length of the array. But now, at the Solidity level, the optimizer can distinguish between a storage modification that affects the array length, such as a push or pop operation, and a modification of the value of an array index that does not have any side effect on its length.

4.2.1. Applicability constraints

The first constraint of this new optimization is that it is only performed over storage arrays. The reason is that while the gas cost of loading a value from storage is high (100 gas units according to EIP-2929 [8]), the load from memory is cheap (3 gas units). Therefore, when optimizing memory arrays, the margin is so low that only the new instructions needed at EVM bytecode level to keep the array lengths on the stack during the whole loop may be more expensive than loading it when required. We leave as a future work the study of an optimization that can also obtain gains with memory arrays. In the case of calldata arrays, this optimization has no sense as they do not need their length to be loaded when being accessed.

A fundamental part of this optimization is to set the conditions that guarantee that the generated code is equivalent to the original. Since we are moving the array length load outside the loop, we need to guarantee that the length remains constant during the execution of the loop. In order to do that, we have first to identify all the ways the array length of a storage array can be modified in Solidity code and then to establish the constraints that guarantee a safe application of the optimization:

- Doing a push operation on the array.
- Doing a pop operation on the array.
- Assigning a new array to the array storage variable.
- Using an inline assembly block.
- Calling a function that performs any previously mentioned operation.

Push and Pop. Push and pop operations increase or decrease by one unit the length of the array they are applied to. This operation can be performed over the dynamic storage arrays using the contract state variable as well as a memory pointer that points to the array, which makes it very difficult to keep track of which storage array is being modified. Therefore, we have established as a constraint that the optimization only is performed if there is no push nor pop operation over any array inside the loop.

Array copy. In Solidity, assignments between storage and memory and between state variables create an independent copy. Therefore, any assignment of an array (in storage, memory, or calldata) to a state variable results in the array being copied to the state variable and, consequently, changing its length. In order to solve this situation, the optimizer does not optimize array accesses where its array base is assigned with another value inside the loop.

Inline Assembly. Inline assembly blocks perform low-level operations using the Yul language, such as direct accesses to storage using `SLOAD` or `SSTORE` bytecode instructions. Therefore, when analyzing inline assembly blocks, we face the same limitations as the Yul optimizer: we cannot identify if a `SSTORE` instruction is modifying the length of an array being used inside the loop. Consequently, the optimization is not performed if there is an inline assembly block inside the loop.

Function calls. Finally, the length of an array can be modified by calling a function that performs any of the previous operations. Therefore, we must guarantee that no modification to an array length is done in the called function or any function called from it. This would require elaborating and analyzing the function call graph, as well as complex reasoning and soundness proof. However, this is out of the scope of this project phase. In order to simplify the check of this condition, we have established a more restrictive constraint that guarantees the previous one: the called function does not modify the state of the contract in any way (there are no writes to storage). We can easily identify if a function modifies the contract state using the state modifiers (see mutability checking in Section 2.1.2 of Chapter 2). These modifiers guarantee that all functions transitively reachable must have declared the same or a more restrictive modifier, and thus we do not need to perform any traversal of the function call graph. Using them, we can establish that the optimization will not take place if there is a call to a function that is not `view` or `pure`.

Another important constraint for this new optimizer is that it only operates on accesses where the base expression is a variable identifier. Therefore, we can only optimize accesses to one-dimensional arrays or to the first dimension of multidimensional arrays. If we wanted to optimize accesses to the second and following dimensions of the array, we would need to store outside the loop the length of each array accessed in each dimension. That solution would not be practical for two main

reasons. First, we cannot easily evaluate at compile time what lengths are needed to optimize the accesses on the second and following dimensions. And second, it would be very complex to generate the code so that each length can be used when accessing its corresponding array since there is a non-static base expression for each array on each dimension.

Finally, another essential constraint that must be guaranteed is that only array variables defined outside the loop can be optimized. This is trivial for contract state variables since they are declared outside the function. Nonetheless, if the array is accessed through a pointer, we can only optimize it if the pointer is declared before entering the loop.

To sum up, we have established the following conditions to establish if we can perform the optimization on a body of a loop:

- There is no push nor pop executed over any array inside the loop.
- There is no inline assembly block inside the loop.
- If there is a function call, this function must be a `view` or `pure` function.

Additionally, we have established three conditions to optimize array accesses inside the loop:

- The array variable needs to be declared before entering the loop.
- The array access base needs to be an identifier. Therefore, only accesses to the first dimension of the array are optimized.
- There cannot be any assignment to the storage variable that points to the array.
- Only accesses to the first dimension of the array are optimized.

Some of the mentioned constraints could be relaxed by performing a deeper code analysis at compile time, such as not allowing any inline assembly block only if it writes to storage or analyzing if the called functions modify the length of an array. However, this is beyond the scope of this project, and it is one of the most interesting lines of future work.

4.2.2. Implementation

Since we want the new optimization phase to take place at a higher level of abstraction and use all the information contained in the AST of the Solidity code, we have implemented it at the same level as the IR code generation[3]. We have defined a new optimization module that the compiler can use while generating the IR code in order to generate an optimized code instead of optimizing the code after its generation.

Therefore, as shown in Figure 4.8, when generating loops from the original code, the `IRGenerator` will use the new optimization module to identify and optimize all the optimizable array accesses performed inside the loop.

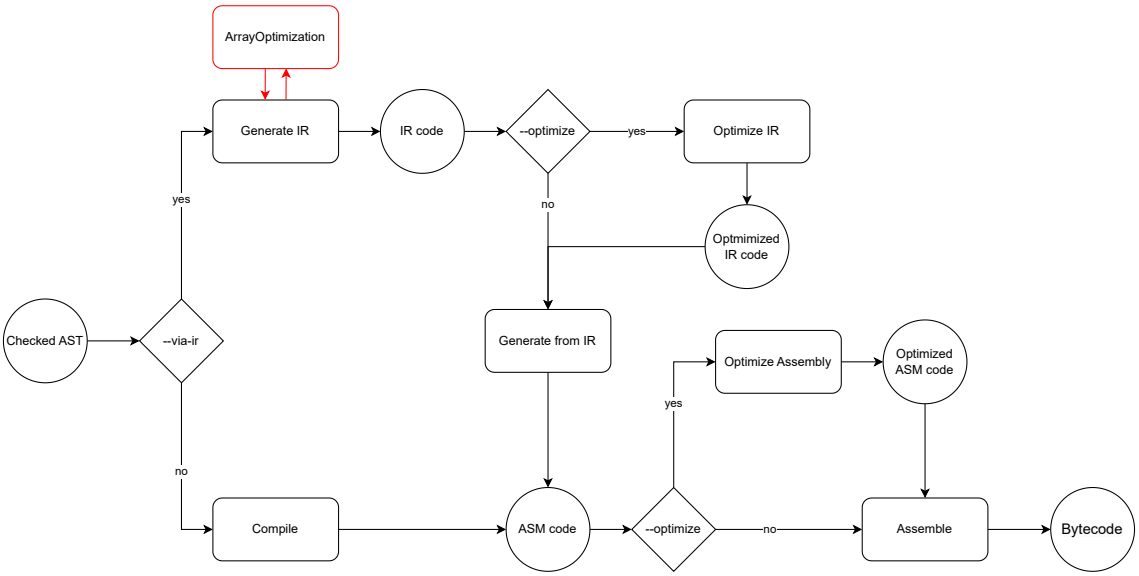

Figure 4.8: Code generation phases of the Solidity compiler with the new optimization phase

This new module comprises three different classes that work together with the IR generation classes to generate optimized array accesses inside the generated loops. In Figure 4.9, we can see how, when generating a loop in IR code, the `ViewArrayChecker` first checks if the loop complies with the constraints to be optimized. Then, the `ArrayLoopOptimiser` will look for and store optimizable array accesses inside the loop. Finally, the `ArrayLoopOptimizedAccessGenerator` generates the variable declarations with the length of the optimized arrays on the IR code before `IRGeneratorForStatements` starts to generate the loop, using the previously declared array lengths variables to replace the length loads both for array length and index accesses.

[3]All changes performed to the original compiler in order to implement this new optimization phase can be found at `https://github.com/ethereum/solidity/compare/develop...javierSande:solidity:arrayLoopOptimization`.

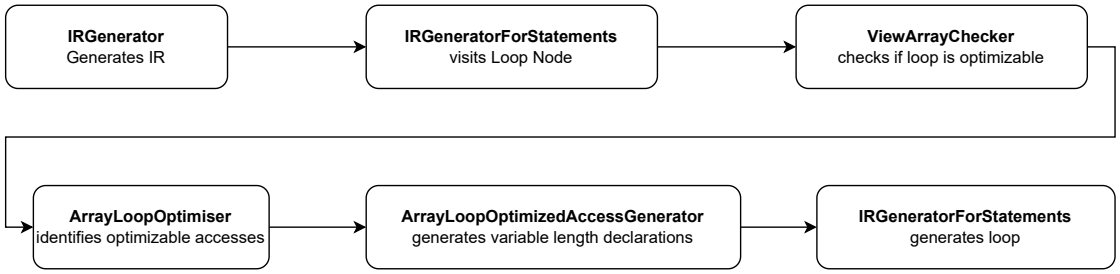

Figure 4.9: Loop generation phases for IR code with the new optimization phase

We can see in Figure 4.10 how the compiler replaces the generation of the length access expression by a call to the previously generated length variable. Moreover, Figure 4.11 shows the generation of an array index access in IR code. Here the length value is passed as a parameter to the auxiliary index access function used to perform the bounds checks.

```
1  if (dynamic_cast<ArrayType const*>(memberAccess.expression().
       annotation().type) &&
2      memberAccess.memberName() == "length" &&
3      m_arrayOptimiser.isOptimizable(memberAccess.expression())
4  ) {
5      define(memberAccess) <<
6          m_arrayOptimiser.getLengthVariable(memberAccess.expression());
7      return;
8  }
```

Figure 4.10: Generation of the IR code for an array length access

```
1  if (m_arrayOptimiser.isOptimizable(baseExpression))
2  {
3      appendCode() << Whiskers(R"(
4          let <slot>, <offset> := <indexFunc>(<array>, <index>, <length>)
5      )")
6      ("slot", slot)
7      ("offset", offset)
8      ("indexFunc", m_utils.storageArrayIndexAccessWithLengthFunction(
           arrayType))
9      ("array", m_arrayOptimiser.getSlotVariable(baseExpression))
10     ("index", IRVariable(*_indexAccess.indexExpression()).name())
11     ("length", m_arrayOptimiser.getLengthVariable(baseExpression))
12     .render();
13 }
```

Figure 4.11: Generation of the IR code for an array index access

View Array Checker. The `ViewArrayChecker` class is in charge of checking that no array length is modified inside the loop that is going to be optimized. This class visits every statement inside the loop body as well as the condition expression and post expression (in the case of a for loop), checking that no statement can change the size of an array according to the rules established in the previous section.

Array Loop Optimized Access Generator. Variables storing the lengths of the arrays in the IR code are generated by the `ArrayLoopOptimizedAccessGenerator` class. This class, called before generating the optimized loop, registers a new Yul variable for each array base stored by the `ArrayLoopOptimiser`. Then, it creates its declaration code where each variable receives as a value the result of the auxiliary length getter function calls, as shown in Figure 4.12. Finally, it returns a map between the optimized arrays and its generated length variable.

```
1 _generator.appendCode() <<
2   "let " << arrayLengthVar << " := " <<
3   _utils.arrayLengthFunction(arrayType) <<
4   "(" << IRVariable(*arrayBase).commaSeparatedList() << ")" << "\n";
```

Figure 4.12: Generation of a length variable in Yul

Array Loop Optimiser. The `ArrayLoopOptimiser` is the main class in charge of the optimization. It calls the other two optimization classes and processes their result, apart from identifying and storing all the optimizable array bases being accessed inside the loop. After `ViewArrayChecker` checks that a loop is optimizable, the `ArrayLoopOptimiser` class visits every array index access and array length access statement of the loop body, loop condition, and post expression, storing the accessed bases. It also visits every assignment statement to check that base arrays on the optimizable list are not modified.

The `ArrayLoopOptimiser` is composed of the two variables storing the information got during the search of optimizable array accesses:

- The `m_accessedArraysSet` is a set storing all the array bases being optimized. The compiler uses this set to check if a given array access must be optimized. Since we only optimize array accesses where the base expression is an identifier, this set contains the variable identifiers represented as strings.

- The `m_accessedArrays` is a vector of the base expressions of each array being optimized. These array base expressions are necessary as they contain information needed to generate the length load outside the loop.

And two other variables generated during the array length variable generation that contains the information needed to use the generated variables to optimize

array accesses:

- The `m_accessedArrayLengthMap` is a map between the variable identifier of all optimized arrays at the generated loop and the identifier of the Yul variables where their lengths are stored in the generated code.

- The `m_accessedArraySlotMap` is a map between the variable identifier of all optimized arrays at the generated loop and the identifier of the Yul variables where their slots are stored. These variables are created during the array length variable generation and need to be kept to be used during the array index access of the array inside the loop.

Finally, this class also implements a mechanism to manage the optimizations on nested loops. During the optimization, all the body loop statements are visited, looking for array accesses, so the optimizer will also detect all the array accesses performed inside nested loops. Therefore, the optimizer will first extract the load of the length to the entry of the most external loop, which may cause duplication conflicts when generating the interior loops. In order to solve that, a straightforward solution would be to disable the optimization on internal loops. However, this may result in optimizations not being performed. If we only optimize the most external loop, when a length load cannot be moved to the entry of the external loop but can be extracted to the entrance of an internal loop, we will be missing an optimization possibility. This is the case of the example shown in Figure 4.13, where the length access of the array a cannot be optimized on the external loop (because of the push at Line 3) but can be optimized in the internal loop.

```
1 for(uint256 i = 0; i < a.length; i++)
2 {
3    a.push(b[i]);
4    for(uint256 j = 0; j < a.length; j++)
5       a[i] += c[j];
6    a[i] += d[j];
7 }
```

Figure 4.13: Example of nested loop

Consequently, we opt for a second and slightly more complex solution. We have implemented the `ArrayLoopOptimiser` to keep track of the arrays being optimized at each level of the nested loops, creating optimization scopes with the arrays being optimized at each loop level. In order to do that, we have extended the class with two stack attributes:

- The `m_scopes` stack with the sets of the arrays being optimized on each nested loop, where the top vector corresponds to the deepest loop.

- The `m_accessedArraysInScope` stack contains the vectors of the base expressions of each array being optimized on each loop, where the top vector corresponds to the deepest loop.

Both stacks are implemented using standard vectors, allowing the compiler to track which arrays must be optimized when entering and leaving a loop.

With the creation of the `m_accessedArraysInScope`, the `m_accessedArrays` vector was deleted. Since the information is used to generate the length variables before the loop generation, all the needed information is contained at the top of the stack for each nested loop. However, the `m_accessedArraysSet` set is conserved to contain all the array bases being optimized at each moment, i.e., it includes all the array bases stored at any scope vector at the `m_scopes` stack, allowing the compiler to check if the array access must be optimized in constant time. If we removed the `m_accessedArraysSet`, the compiler would need to check if the array base is in any stack vector, resulting in poor efficiency. When the optimizer and generator leave a loop, the scope vector at the top of the `m_scopes` stack removes from the set the arrays that could only be optimized inside the loop walked out.

4.2.3. Usage

This new array access optimization is performed during the IR generation, resulting in an IR code where the array accesses inside loops are already optimized. Because of that, we need the compiler to use the IR code to generate the EVM assembly code and, consequently, the final bytecode (see Section 2.1.3 in Chapter 2). In order to do that, we have to set the flag *–via-ir* to force the code generation through IR code. However, if we generate the final code using the IR without being optimized by the Yul module, the resulting code will likely have a higher gas consumption even if our optimization is applied. This is because, since the IR generation must be as straightforward as possible, the compiler is designed to create one variable per computed expression resulting in many unnecessary variable declarations that increase gas consumption. Consequently, when using the compilation via IR, activating the Yul optimizer module is mandatory to get a code without redundant variable declarations that produce a higher cost. In fact, as seen in the code optimization section in Chapter 2, using the Yul optimizer with the compilation via IR produces the most optimized code the current compiler can produce. Therefore, when applying our optimization, we must force the compiler to use the Yul optimization module (*–optimize* flag), so we get the most optimized code that the compiler can produce, including now our array access optimization.

In order to make it easier for the developers to configure the compiler to use this new optimization, we have created the *–optimize-arrays* command line option. This new flag has the same effect as using *–via-ir* and *–optimize* flags together. Figure 4.14 shows the optimization command line options of the compiler, including

the new *–optimize-arrays* flag.

```
Optimizer Options:
 --optimize Enable bytecode optimizer.
 --optimize-runs n (=200)
 The number of runs specifies roughly how often each
 opcode of the deployed code will be executed across the
 lifetime of the contract. Lower values will optimize
 more for initial deployment cost, higher values will
 optimize more for high-frequency usage.
 --optimize-yul Legacy option, ignored. Use the general --optimize to
 enable Yul optimizer.
 --no-optimize-yul Disable Yul optimizer in Solidity.
 --optimize-arrays Enable array access optimizer in Solidity. Same
    effect as --optimize and --via-ir.
 --yul-optimizations steps
 Forces yul optimizer to use the specified sequence of
 optimization steps instead of the built-in one.
```

Figure 4.14: Optimization options of the compiler

4.3. Results and experiments

As explained at the beginning of this chapter, this new optimization stores the length of storage arrays being accessed (both to access its length or index) inside a loop on a local variable (stored in the stack). Then this variable is used to replace length loads on length accesses (e.g., condition of the loop) and bounds checks on index accesses. With this slight change in the code, we save the cost of loading a value from storage on each array access on each iteration of the loop. We only maintain the cost of the first length load, which is now performed previous to the start of the loop. Therefore, we are saving 100 gas units per access on each iteration [4] in each iteration except the first one.

However, since we are adding the array lengths to the stack, the generated bytecode may introduce new instructions to keep these values on the top of the stack during the loop execution. Consequently, the final saved amount may be slightly lower. Nevertheless, the difference between the cost of operations on the stack (e.g., `PUSH`, `POP`, `DUP`), which usually varies between 1 and 3 gas units, is much lower than a load from storage (100 gas units) so only on remote cases the optimization may not reduce the gas consumption because of those extra instructions.

[4] According to EIP-2929 [8], the cost of the first access to a storage slot is 2100 gas units. However, since we maintain this access, this does not affect the saving computation.

Additionally, there is a specific case where the optimization increases the gas consumption: if the loop is not accessed. Since with the optimization, the array lengths are loaded before entering the loop, if the loop is not accessed, we will produce extra loads from storage. This issue can be easily solved in future work by wrapping the array loads on a conditional statement with the same condition as the array loop.

4.3.1. Simple experiment

In order to prove the theoretical gas saving per access, we used a simple function similar to the one used in the previous chapter to test the `uncheckedArray` block.

Figure 4.15 shows the tested function, which saves the sum of two arrays in one array, all in storage.

```
1 function sumArrays() public {
2   for(uint256 i = 0; i < size; i++)
3      a[i] = b[i] + c[i];
4 }
```

Figure 4.15: Tested function

The execution of this simple benchmark on arrays of different lengths (1, 10, and 100 elements) produced the results shown in Table 4.1.

Iterations	Original	New Optimization	Diff	Diff per Iteration
1	36305	36376	-71	-71.00
10	278387	275965	2422	242.20
100	2699207	2671855	27352	273.52

Table 4.1: Gas cost difference between using the original optimization and the new optimization

We observe that our new optimization increases gas consumption when executing the code over an array with a single element. This consumption increase is because, as explained, the optimization takes the length loads out of the loop, but they are still performed once. Consequently, since the optimization cannot reduce the number of loads but still introduces an extra cost by creating new local variables, the final gas cost is slightly increased (less than an 0.2%).

However, as we increase the length of the array and consequently the number of iterations, the optimization becomes more effective, increasing the gas saves per

iteration to an amount close to the expected 300 gas units (3 length loads from storage) for this case.

4.3.2. Real life experiments

The main goal during this phase is to improve the current array access optimizations providing a new optimization that covers a much more comprehensive range of scenarios and that can be used on real code without any needed modification, as required with the `uncheckedArray` block. Therefore, we want to measure the efficiency of this new optimization on the code of contracts currently being used in the Ethereum blockchain. In order to do that, we have selected public smart contracts and libraries that developers frequently use to manage arrays and matrices.

Using such libraries and contracts, we have developed a benchmark [5] to measure the gas saving when applying our new optimization. This benchmark is composed of three different test suites that use three real libraries to perform different operations over storage arrays and matrixes.

We have slightly modified the code of some of the original libraries to adapt them to the current compiler version and only use storage arrays.

In order to evaluate the gas savings produced with this new optimization, we have used the same optimization setup for both the current compiler and the compiler with our new optimization. We have set up the *–optimize* and *–via-ir* flags, which generate the most optimized code the current optimizations can generate.

Array Library. The Array test suite uses an array management library from the Solidity Standard Library [4]. This library repository provides two libraries, `UintArray` and `IntArray`, to manage arrays of unsigned and signed integers. Both libraries are implemented as contracts that store the integer array in a state variable (in storage), allowing the calling contract to perform different operations over the array.

In order to develop our test suite, we have selected three methods of the `UintArray` library (maximum value getter, the minimum value getter, and the sum of the elements of the array), and we have created three different tests where we create an array, call the corresponding method and verify the correctness of the returned value. Moreover, since the library was developed with a target to the 0.4.0 version of the compiler, we had to modify some function declarations in order to adapt themselves to the current compiler version restrictions, such as the constructor definition or the memory space declaration on function reference parameters.

[5] The used benchmarks can be consulted at `https://github.com/javierSande/solidity-benchmarks.git`.

We have computed the gas savings by executing each test three times over arrays of 100 elements sorted in ascending order, descending order, and randomly generated. Table 4.2 and Figure 4.16 show the average gas consumption of each test with and without our optimization.

Method	Original Gas	Optimized Gas	Diff	Percentage
Max	299687	280597	19090	6.37%
Min	296619	278543	18076	6.09%
Sum	300269	282369	17900	5.96%

Table 4.2: Gas savings on Array Library tests

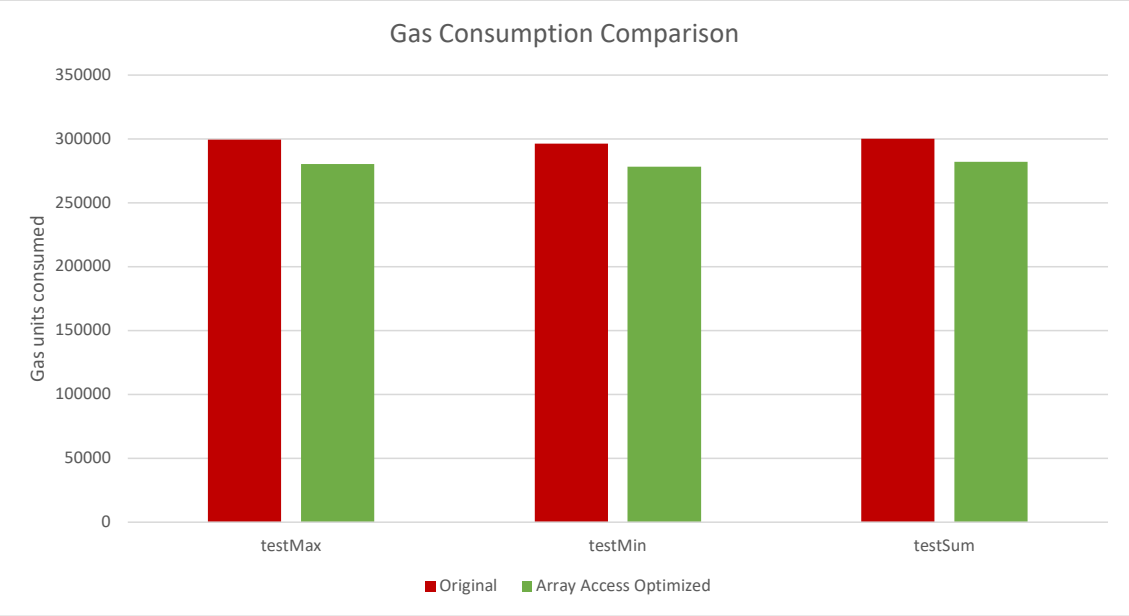

Figure 4.16: Comparison of the gas cost between original code and optimized code

The results show a gas reduction of around 6% on each test. We can observe that the *sum* test has a slightly lower saving than the other two tests because, on this test, the array is accessed only once per iteration, while on the other tests, it is accessed twice whenever local minimums or maximums are found. Additionally, we can observe s a slight difference in gas consumption and saving between the maximum and minimum getter tests, which are technically identical in terms of performance, which is explained because of the randomness of the array. Probably, the array sorted randomly produces more accesses when looking for the maximum (more local maximums), which explains the higher average gas cost and the higher gas saving on the *max* test over the *min* test.

From these results, we conclude that the optimization produces a significant gas saving even in functions where arrays are accessed only a few times per iteration, where the potential gas reduction is lower.

Matrix Library. The Matrix test suite has been developed using the SolMATe libraries [20] for floating-point computation, array manipulation, and linear algebra. From it, we took the `VectorUtils` and `MatrixUtils` libraries to develop several tests on array manipulation. With these test suites, we want to measure the efficiency of our optimization when manipulating matrices, even when it can only optimize accesses to the first dimension of a matrix.

We have developed six tests to add, multiply, and transpose matrices, add or multiply a matrix by a number, and compute the diagonal of a matrix. Since both libraries only supported memory arrays, they have been modified to be able to operate over storage arrays. Additionally, some unused functions were removed. Each test calls a library function that receives storage matrices as parameters and returns a new matrix or vector (diagonal) stored in memory. Therefore, our new optimization will only reduce gas from the array read accesses, limiting its potential even more.

Method	Original Gas	Optimized Gas	Diff	Percentage
Add Matrix	833995	751526	82469	9.89%
Add Number	511443	441897	69546	13.60%
Diagonal	100008	97978	2030	2.03%
Dot	1868660	1751896	116764	6.25%
Multiply Number	515487	445941	69546	13.49%
Transpose	462439	417172	45267	9.79%

Table 4.3: Gas savings on Matrix Library tests

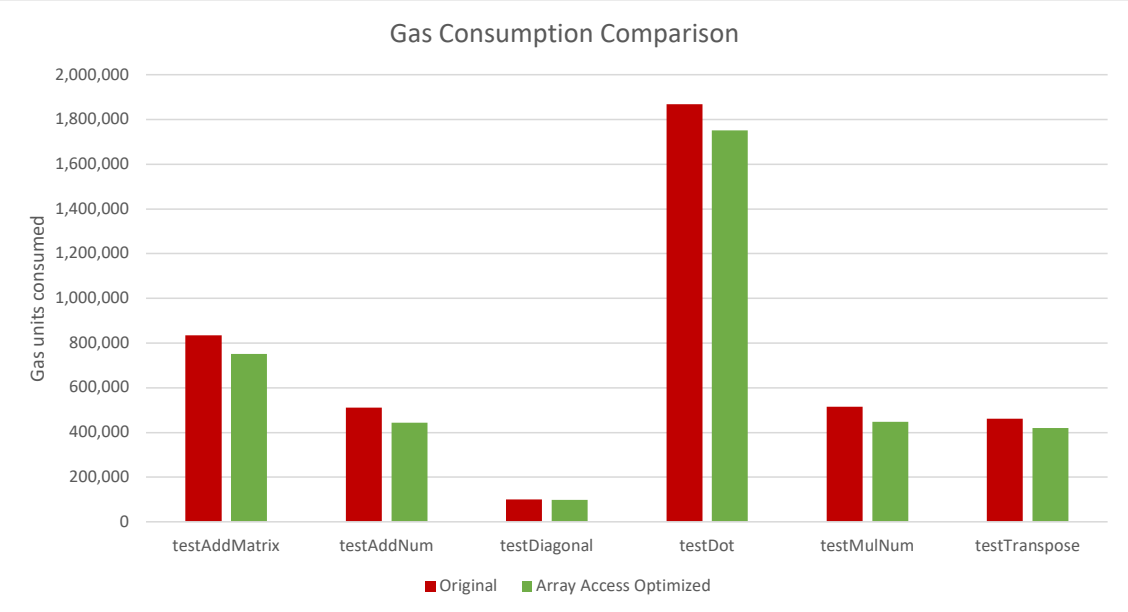

Figure 4.17: Comparison of the gas cost between original code and optimized code

Tests have been executed over a storage 10 x 10 matrix with random integer values, using both the current compiler and the modified compiler containing our new optimization.

Despite the limitations, with our optimization, we reduce the gas consumption between 9% and 14% in most tests. Table 4.3 and Figure 4.17 show a great reduction in tests with many array accesses per iteration, such as additions and multiplications, where all the array elements are accessed.

Nonetheless, we also observe a very low gas reduction on the *diagonal* test, which performs much fewer array accesses than other tests. As shown in Figure 4.18 in a 10 x 10 matrix, the function only performs ten iterations with two array index accesses (1 to the rows and 1 to the columns) per iteration, where we only optimize the row access (first dimension). The test performs an identical operation to check that the diagonal values are correct, so we have to double the number of iterations. From those 20 iterations, we are getting a saving of 2030 gas units, which means we are saving around 100 units per iteration. Knowing that the loop complies with the current compiler constraints to optimize the array length access on the condition, we can conclude that our optimizer is saving this extra gas from each array index access in the loop. Therefore, our optimization is using its maximum potential, and the only reason the saving percentage is low is that most of the gas consumption on the function is produced by other operations, such as creating the memory vector[6].

```
1 function diag(int256[][] storage a) internal view returns (int256[]
    memory) {
2    int256[] memory diagonal_vector = new int256[](a.length);
3    for (uint i=0; i<a.length; i++) {
4       diagonal_vector[i] = a[i][i];
5    }
6    return diagonal_vector;
7 }
```

Figure 4.18: Diagonal function of the `MatrixUtils` library

The case of the *dot* test is similar. Although the dot operation requires accessing every value in the array since it is a complex function with three nested loops, it has a high gas cost derived from other operations, and consequently, the percentage of the saved gas may be lower but still significant (more than 100,000 gas units).

Those results prove that our new optimization, despite its limitations on multidimensional arrays, performs well when optimizing matrix accesses. It is also probable that as we increase the number of dimensions of an array, this performance decreases,

[6]The creation of vector in memory has relatively high cost due to memory expansion (EIP-2929 [8]).

but the use of storage arrays of 3 or more dimensions is not frequent in Solidity smart contracts since the cost of manipulating them is extremely high.

Sorting Library. The Sorting tests suite is a collection of the most common sorting methods in programming adapted to Solidity, contained in the `SortLib` library. Sorting methods are one of the most access intense manipulations of arrays, performing several array index accesses per iteration in order to read, compare and reorder values. Consequently, they are a great benchmark to measure the real potential of our optimization on a complex array manipulation.

On the `SortLib` library, we have included several integer sorting methods with different complexities: the selection, insertion, and bubble (standard and optimized) sort methods, which have a n^2 time complexity, and the heap sort method with a $nlogn$ time complexity. Using the mentioned library, we have developed several tests to call the corresponding sorting method and check the result. Each test has been executed three times with an array with 100 integers sorted in ascending (best-case scenario), descending (worst-case scenario), and random order to guarantee reliable average gas consumption results.

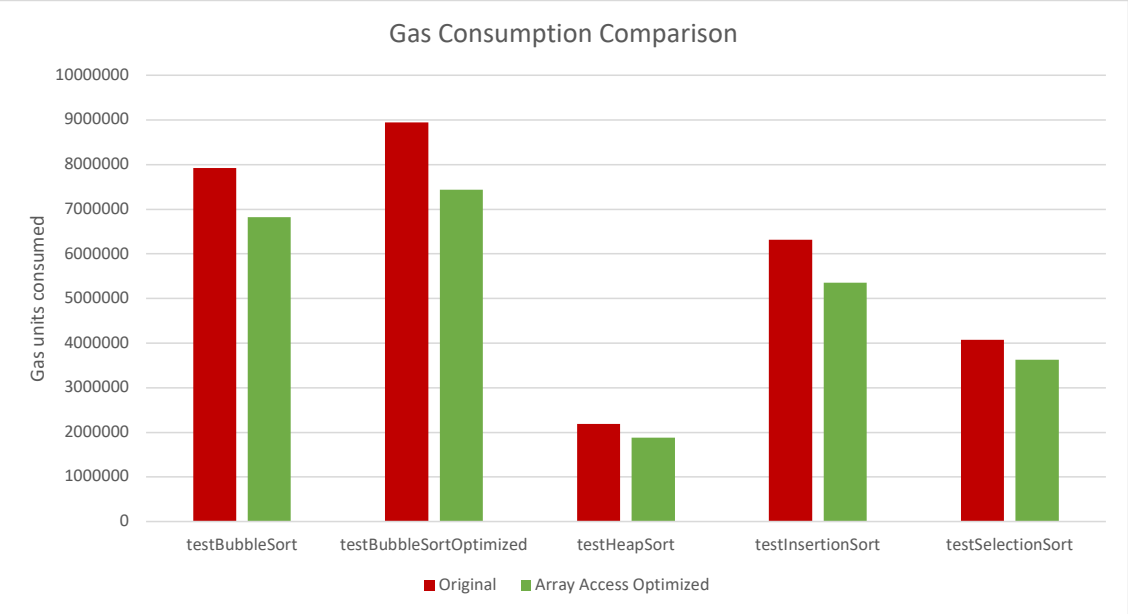

Figure 4.19: Comparison of the gas cost between original code and optimized code

Table 4.4 and Figure 4.19 show the average gas consumption on each test for the code compiled with and without our new optimization. These results indicate that our optimization produces a remarkable reduction of gas consumption, between 10% and 17% of the total gas cost, independently of the order of complexity of the sorting method.

Method	Original Gas	Optimized Gas	Diff	Percentage
Bubble Sort	7928770	6828488	1100282	13.88%
Bubble Sort Optim.	8948553	7441954	1506599	16.84%
Heap Sort	2184631	1876938	307693	14.08%
Insertion Sort	6314264	5356155	958109	15.17%
Selection Sort	4069664	3623238	446426	10.97%

Table 4.4: Gas savings on Sort Library tests

4.4. Conclusions

In this chapter, we have presented a new optimization for array accesses performed inside loops. As explained, this is an optimization already being performed by the Yul optimizer module of the official compiler. However, since this optimization is being performed at Yul level, the compiler lacks much important information about the code behavior. Consequently, the amount of optimized Yul code is limited and can only target particular cases. As a solution to the significant limitations of the Yul optimizer, we have proposed a new optimization phase at a higher level. This new phase works with information from the source code and its AST, having much more information on the effects of the code on the program state and, therefore, overcoming most of the limitations the Yul optimizer faces.

Then, we implemented a new array optimization in this new phase using the same idea as the original optimization. This optimization identifies accesses to storage arrays inside loops, analyzes if their length remains constant on each iteration, and moves the load of their lengths outside the loop when possible, avoiding loading the length on each iteration.

In order to prove the effectiveness of this new optimization, we have compared the gas consumption of different smart contracts when only applying the current optimizations and when adding our optimization. Firstly, we have compared the consumption of a simple contract to identify the origin of the gas savings easily. Results show us that the gas savings are as expected and that, even in simple contracts, the proposed optimization goes further than the current optimizer. Finally, we have compared the gas consumption over real libraries used by developers to manage arrays in their smart contracts. The results have shown a more than considerable reduction in gas consumption, exceeding the 10% in most cases.

This new compiler optimization has proved to be almost as efficient as the `uncheckedArray` block when reducing the gas cost of accessing storage arrays. Additionally, it has two advantages over the mentioned block: it preserves safety, and we do not need to modify code to get optimized accesses. On the downside, it only optimizes array accesses inside loops and in specific situations where the compiler can guarantee that the array length remains constant.

Conclusions and Future Work

We started this project with the aim of optimizing array accesses in Solidity smart contracts. In order to do so, we focused on reducing the overhead produced by bounds checks.

The initial study of the Solidity language, its compiler, and the optimization strategies used to reduce gas consumption on smart contracts allowed us to present and implement two optimization proposals to accomplish our initial goal.

The first proposed solution was to allow programmers to disable bounds checks on index accesses. In order to do so, we based ourselves on a solution currently used in the Solidity language that disables underflow and overflow checks on arithmetic operations, the `unchecked` block. From this idea, we came up with a new language construct, the `uncheckerArray` block, that disables bounds checks on any array access enclosed in the block, and which has proven its effectiveness on experimental results, reducing gas consumption in memory, calldata, and, most significantly, storage arrays.

However, this solution is not perfect. It puts safety in the hands of programmers, requires to modify the code, and is not able to reduce gas consumption of accesses where the bounds checks are needed. All these drawbacks of the first solution motivated a completely different one: an optimization at compile time.

In this second solution, we wanted to create a new compiler optimization on array accesses. From the study of the current compiler, we discover that the optimizations being performed are limited by the low-level information about the code they have. Accordingly, we implemented a new optimization phase that takes place during the IR generation, using the information from the Solidity source code and its AST. Using this new phase, we implemented a new optimization that targets storage array accesses inside loops, reducing its gas cost derived from the length load on bounds checks.

This new optimization has proven a remarkable efficacy, showing substantial gas

reductions when applied to real smart contracts and demonstrating its ability to optimize scenarios beyond the reach of the existing compiler optimizations.

We sincerely believe both proposals have great potential to be implemented in a future version of the official compiler, and it is worth continuing to work on them.

On the one hand, the `unchecked` block is the perfect candidate because of its proven effectiveness and its similarity, both in behavior and usage, with the `unchecked` block for arithmetic instructions. It is a solution that fulfills its purpose and, at the same time, is based on an idea that is currently present in the production compiler.

On the other hand, the new optimization phase requires more future work in order to release a version that can be implemented on a production compiler. Nevertheless, its proven impact on gas reduction and all the potential new optimizations that can be derived from this new phase make this proposal an exciting development path.

Future work

The optimization proposals introduced in this project open the door for vastly future work. Both the `uncheckedArray` block and the new compiler phase require additional work in order to be ready for production.

As previously mentioned, the `uncheckedArray` block is the proposal with the higher possibilities to be implemented in a future version of the official compiler. However, in order to make this proposal succeed, there are several steps to complete:

1. Formally propose the new `uncheckedArray` block to the developer community, including the Ethereum Foundation, and study the feedback received.

2. Study the viability of the target list of optimizable arrays since the proposed implementation may confuse developers. The literal comparison of the targeted array bases is quite limiting and confusing. Therefore, exploring other ways of comparing array base expressions would be extremely interesting.

3. Write the corresponding documentation about the block, its usage, constraints, and effects on the compiler documentation.

4. Develop unit tests according to the compiler standards, including compilation and gas consumption tests.

5. Prepare a *pull request* with the implementation, tests, and documentation in the official repository of the Solidity compiler.

Regarding the new compiler optimization for array accesses, its implementation on a future version of the official compiler may involve a longer process than the `uncheckedArray` block. However, this proposal opens a wide range of lines of future work. The possible lines of work range from working on the formalization and improvement of the optimization to developing new optimizations on the foundations set by our proposal. The most relevant possibilities are:

- **Formalize and prove the correctness of the optimization.** Proving the correctness of the optimized code is fundamental to guarantee that the optimization is safe and has no undesired side effects.

- **Reduce constraints.** Most of the constraints used in Chapter 4 are general and straightforward in order to have a simple implementation that does not shift the focus from the basics of the optimization phase. However, as mentioned in Chapter 4, some constraints can be relaxed. There are other possible workarounds that take advantage of the high-level information of the AST of the source code. For example, the limitation of calls to `view` and `pure` functions can be relaxed by analyzing the called functions to guarantee that they do not modify the length of any array. We can also relax the restriction applied to inline assembly blocks by guaranteeing they do not write to storage.

- **Improve performance.** As mentioned in Chapter 4, the proposed implementation of the new optimization may perform poorly in some scenarios, such as loops not being accessed or accesses to arrays with more than two dimensions.

- **Generalize the optimization.** Our optimization targets array accesses inside loops. Since they are executed several times, the possible extra cost generated is amortized, and the amount of saved gas is high. However, it is also possible to target code sequences where an array is accessed several times without changing its length and perform a similar optimization where we save the array length at the beginning of the sequence to use it on each access to the array.

- **New compiler optimizations.** In order to implement our new array access optimization, we created a new optimization phase on the compiler at a higher level than the existing ones. The creation of this new phase lays the foundation for the creation of new high-level optimizations that were not possible until now. Array accesses are just an example of all the possible code structures that can be targeted and optimized in this new phase.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 2 edition, 2006.

[2] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts. In A. Biere and D. Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–125. Springer International Publishing, 2020.

[3] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio. Inferring needless write memory accesses on ethereum bytecode. In S. Sankaranarayanan and N. Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 448–466. Springer Nature Switzerland, 2023.

[4] alianse777. *Solidity Standard Library*. 2018. `https://github.com/alianse777/solidity-standard-library`.

[5] T. Brandstätter, S. Schulte, J. Cito, and M. Borkowski. Characterizing efficiency optimizations in solidity smart contracts. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 281–290, 2020.

[6] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017.

[7] Ethereum Foundation. Solidity v0.4.20 release announcement, 2018. `https://blog.soliditylang.org/2018/02/14/solidity-0.4.20-release-announcement/`.

[8] Ethereum Foundation. *EIP-2929*. 2020. `https://eips.ethereum.org/EIPS/eip-2929`.

[9] Ethereum Foundation. Solidity v0.8.0 release announcement, 2020. `https://blog.soliditylang.org/2020/12/16/solidity-v0.8.0-release-announcement/`.

[10] Ethereum Foundation. Solidity documentation, 2021. https://docs.soliditylang.org/en/latest/index.html.

[11] Ethereum Foundation. Inline asesembly documentation, 2022. https://docs.soliditylang.org/en/v0.8.19/assembly.html.

[12] Ethereum Foundation. Solidity smt checker documentation, 2022. https://docs.soliditylang.org/en/v0.8.19/smtchecker.html.

[13] Ethereum Foundation. Yul documentation, 2022. https://docs.soliditylang.org/en/v0.8.19/yul.html.

[14] Ethereum Foundation. Yul optimizer of solidity compiler, 2022. https://docs.soliditylang.org/en/v0.8.19/yul.html#yul-optimizer.

[15] Ethereum Foundation. Solidity repository, 2023. https://github.com/ethereum/solidity.

[16] FuelLabs. *Yul+*. 2020. https://github.com/FuelLabs/yulp.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[18] LLL. Lll documentation, 2014. https://lll-docs.readthedocs.io/en/latest/lll_introduction.html.

[19] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. https://bitcoin.org/bitcoin.pdf.

[20] NTA-Capital. *SolMATe Library*. 2018. https://github.com/NTA-Capital/SolMATe.

[21] OpenZeppelin. Safemath library, 2022. https://docs.openzeppelin.com/contracts/2.x/api/math#SafeMat.

[22] N. Szabo. Smart contracts: Building blocks for digital markets, 1996. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[23] T. Takenobu. *Ethereum EVM Illustrated*. 2018. https://github.com/takenobu-hs/ethereum-evm-illustrated.

[24] Vyper. Vyper documentation, 2020. https://docs.vyperlang.org/en/stable/.

[25] G. Wood. Ethereum: A secure decentralised generalised transaction ledger, 2019. https://ethereum.github.io/yellowpaper/paper.pdf.

Adios, ríos; adios, fontes;
adios, regatos pequenos;
adios, vista dos meus ollos:
non sei cando nos veremos.
Miña terra, miña terra,
terra donde me eu criei,
hortiña que quero tanto,
figueiriñas que prantei,
prados, ríos, arboredas,
pinares que move o vento,
paxariños piadores,
casiña do meu contento,
muíño dos castañares,
noites craras de luar,
campaniñas trimbadoras,
da igrexiña do lugar,
amoriñas das silveiras
que eu lle daba ó meu amor,
camiñiños antre o millo,
¡adios, para sempre adios!
¡Adios groria! ¡Adios contento!
¡Deixo a casa onde nacín,
deixo a aldea que conozo
por un mundo que non vin!
Deixo amigos por estraños,
deixo a veiga polo mar,
deixo, en fin, canto ben quero...
¡Quen pudera non deixar!...

Cantares Gallegos

Rosalía de Castro

Caminante, son tus huellas
el camino y nada más;
Caminante, no hay camino,
se hace camino al andar.
Al andar se hace el camino,
y al volver la vista atrás
se ve la senda que nunca
se ha de volver a pisar.
Caminante no hay camino
sino estelas en la mar.

Proverbios y cantares
Campos de Castilla
Antonio Machado