Chapter 3

Methodology

Chapter overview

I Compilers and evm theory

A. Introduction to compilers

Compiler construction is a sophisticated field in computer science, involving the creation of software that translates source code written in a high-level programming language into a lower-level language, often machine code. The process of compiler construction is typically divided into several distinct phases, each with its specialized function:

1. Lexical Analysis

    - Token Generation: The compiler's first phase involves breaking down the source code into tokens, which are the basic building blocks of the language (like keywords, operators, identifiers, literals).

- Regular Expressions: Lexical analyzers often use regular expressions to recognize these tokens, simplifying the source code into a stream of tokens for further processing.

2. Syntax Analysis

   - Parsing: This phase involves analyzing the token sequence from lexical analysis to ensure that the expressions and statements conform to the rules of the programming language's grammar.
   - Syntax Trees: The result is often a syntax tree, which represents the syntactic structure of the source code in a hierarchical tree format.

3. Semantic Analysis

   - Type Checking: The compiler checks whether the parsed code makes logical sense, adhering to the language's semantic rules. This includes type checking, where the compiler verifies that each operation is compatible with its operands.
   - Symbol Table: The compiler uses a symbol table to keep track of variable names, their types, and other relevant information to ensure correct interpretation and context.

4. Intermediate Code Generation

   - Abstract Representation: The compiler translates the source code into an intermediate representation, which is a lower-level representation of the program that retains its logical structure.

- Platform-Independent Code: This intermediate code is typically platform-independent, serving as a bridge between the high-level source code and the machine code.

5. Optimization

   - Performance Improvement: The optimization phase refines the intermediate code to improve efficiency and performance. This can include eliminating redundant calculations, optimizing loops, and more.
   - Target-Dependent Optimizations: These optimizations often vary depending on the target machine's architecture and capabilities.

6. Code Generation

   - Machine Code: The final phase involves translating the optimized intermediate code into machine code specific to the target machine CPU architecture or virtual machine instruction set.
   - Assembly Language: Sometimes, the output is in the form of assembly language, which is then further translated into machine code by an assembler.

7. Linking and Loading

   - Linking: If the program consists of multiple source code files or external libraries, the linking process combines these into a single executable.
   - Loading: The loader then places the executable into memory for execution by the machine.

In the context of Elixireum, these compiler construction principles must be adapted to suit the specific needs of the Ethereum Virtual Machine (EVM). This includes considerations for smart contract compilation, optimization for gas efficiency, and adhering to the deterministic execution model of the EVM. Understanding these compiler construction phases is essential for Elixireum development, as each stage plays a critical role in transforming high-level Elixireum code into efficient, secure, and EVM-compatible bytecode.

B. Ethereum Virtual Machine - A Deeper Insight

EVM is a core component of the Ethereum blockchain, pivotal in maintaining its decentralized and secure nature. It serves as an execution environment for smart contracts, enabling the remarkable functionality that Ethereum offers. Understanding the EVM is crucial for developing Elixireum, as it provides the foundational framework within which this new language operates.

1. Execution Environment

    - Isolated System: The EVM operates as an isolated environment, completely sandboxed from the Ethereum network. This isolation ensures that code execution within the EVM does not directly affect the network or the filesystem of the host computer.
    - Deterministic Execution: The EVM is designed to be deterministic. This means that executing the same smart contract with the same input will always produce the same output across all nodes in the Ethereum network. This characteristic is essential for maintaining consistency and agreement (consensus) across the decentralized network.

2. EVM's Role in Smart Contracts

    - Smart Contract Lifecycle: Developers write smart contracts in high-level languages like Solidity or Vyper. These contracts are then compiled into EVM bytecode. When a contract is deployed on the Ethereum blockchain, it's executed by the EVM to ensure it behaves as expected.
    - Gas Mechanism: The EVM uses a gas mechanism to measure and limit the resources (computation and storage) a contract can use. Each operation in the EVM has a gas cost associated with it, preventing inefficient or malicious code from overloading the network.

3. State Machine

    - State Transitions: The EVM can be viewed as a state machine, where transactions transition the Ethereum blockchain from one state to another. Smart contracts, through their executions, modify the state stored in the Ethereum blockchain.
    - EVM Bytecode: The EVM interprets a series of bytes (bytecode) which instruct the machine on the operations to perform. This bytecode is the compiled version of high-level contract code.

4. Storage and Memory

    - Storage: The EVM has a long-term storage model, where data is stored persistently on the blockchain. This storage is expensive but necessary for maintaining the state across transactions.

- Memory: The EVM also includes a volatile memory space, cleared after each transaction, used for temporary storage during contract execution.

5. Challenges and Considerations for Elixireum

- Gas Optimization: Given the cost implications of EVM operations, Elixireum must focus on gas optimization during the compilation process.
- Security and Determinism: Security vulnerabilities in smart contracts can be costly. Hence, Elixireum needs to ensure robust security measures. Additionally, the deterministic nature of the EVM requires that Elixireum-generated bytecode behaves consistently across all executions.

II Problems (no researches, more hacking way to research)

Given the unique syntax borrowed from Elixir for the language we developed, our journey in crafting Elixireum often led us to delve into the implementation of Elixir itself. Due to Elixir's niche popularity, resources on writing a compiler in Elixir are scarce, and at times, completely nonexistent on the internet. Consequently, we found ourselves independently sourcing this information, relying heavily on the original codebase, experimentation, and our accumulated expertise. Similarly, our encounter with the intermediate Yul language presented its own set of challenges. The entirety of its documentation

seems to be condensed into a single chapter within the Solidity documentation, leading to a notable dearth of practical usage examples and detailed explanations of certain concepts. Faced with these constraints, we embarked on a quest for knowledge through hands-on experimentation and meticulous analysis of the opcodes found within the bytecodes of various examples. This approach allowed us to uncover and understand the intricacies of Yul, despite the limited resources at our disposal.

III Research design (methods: techonology stack, compiler stages, decisions like why we compile and not interpret)

IV Limitations (elixirish distribution is not applicable to evm, so it is cannot be used in smart contracts, libraries are out of the scope, external smart contract interactions, need to spec types for public functions, need of solidity compiler)

NOT LIMITATIONS BUT CONSTRAINTS (OR FROM LEVS PAPER MB REQUEIREMENTS) While this study provides valuable insights into the application of elixirish distribution in Ethereum Virtual Machine (EVM) environments, it is important to acknowledge several limitations that may impact the generalizability and scope of the findings.

While we

Inapplicability to EVM: The elixirish distribution, a key focus of this research, is not directly applicable to the Ethereum Virtual Machine (EVM). As a result, its usage in smart contracts within the EVM environment is not feasible, limiting the direct implementation of elixirish distribution in Ethereum-based decentralized applications.

Exclusion of Libraries: The scope of this study does not encompass the incorporation of external libraries. The exclusion of libraries may affect the versatility of the proposed methods, as the integration of additional functionalities from external sources is not considered in the current framework.

External Smart Contract Interactions: Interactions with external smart contracts fall outside the purview of this research. The study primarily focuses on the internal dynamics and distribution mechanisms within a single smart contract. External interactions, if any, are not explored in this context.

Specification Requirements for Public Functions: The necessity to specify types for public functions is a limitation inherent in the proposed methodology. While this ensures a higher degree of predictability and security, it imposes additional constraints on developers, requiring explicit type specifications for public functions to ensure compatibility with elixirish distribution.

Dependency on Solidity Compiler: The reliance on the Solidity compiler is another limitation of the methods presented in this study. Changes in the Solidity language or compiler versions may impact the applicability and stability of the elixirish distribution approach, necessitating careful consideration and potential adjustments in future developments.

It is crucial for future researchers and practitioners to be aware of these limitations when applying or extending the proposed methods in different con-

texts. Addressing these constraints may pave the way for more comprehensive and robust solutions in the evolving landscape of smart contract development and blockchain technologies.

V architecture and modules overview

VI other languages comparison

VII Conclusion

...

Referencing other chapters ??, 3, ??, ?? and ??

TABLE I
Simulation Parameters

A	B
Parameter	Value
Number of vehicles	$\mid \mathcal{V} \mid$
Number of RSUs	$\mid \mathcal{U} \mid$
RSU coverage radius	150 m
V2V communication radius	30 m
Smart vehicle antenna height	1.5 m
RSU antenna height	25 m
Smart vehicle maximum speed	v_{max} m/s
Smart vehicle minimum speed	v_{min} m/s
Common smart vehicle cache capacities	$[50, 100, 150, 200, 250]$ mb

A	B
Common RSU cache capacities	$[5000, 1000, 1500, 2000, 2500]$ mb
Common backhaul rates	$[75, 100, 150]$ mb/s

Fig. 1. One kernel at x_s (dotted kernel) or two kernels at x_i and x_j (left and right) lead to the same summed estimate at x_s. This shows a figure consisting of different types of lines. Elements of the figure described in the caption should be set in italics, in parentheses, as shown in this sample caption.

...